Visual Numerics®

# JMSL™

Numerical Library

# User's Guide

VOLUMES 1 and 2 of 3

VERSION 3.0

**Visual Numerics Corporate Headquarters**
12657 Alcosta Boulevard, Suite 450
San Ramon, CA 94583

**USA Contact Information**

| | |
|---|---|
| Toll Free: | 800.222.4675 |
| San Ramon, CA: | 925.415.8300 |
| Westminster, CO: | 303.379.3040 |
| Houston, TX: | 713.784.3131 |
| Fax: | 925.415.9500 |
| Email: | info@vni.com |
| Web site: | www.vni.com |

**Visual Numerics has Offices Worldwide**
USA • UK • France • Germany • Mexico • Japan
• Korea • Taiwan
For contact information, please visit
www.vni.com/contact

**For IMSL Support Contact Information**
www.vni.com/tech/imsl/phone.html




C, C#, Java™, and Fortran
Application Development Tools

# Contents

# Chapter 1

# Linear Systems

Classes



## Usage Notes

### Solving Systems of Linear Equations

A square system of linear equations has the form $Ax = b$, where $A$ is a user-specified $n$ x $n$ matrix, $b$ is a given right-hand side $n$ vector, and $x$ is the solution $n$ vector. Each entry of $A$ and $b$ must be specified by the user. The entire vector $x$ is returned as output.

When $A$ is invertible, a unique solution to $Ax = b$ exists. The most commonly used direct method for solving $Ax = b$ factors the matrix $A$ into a product of triangular matrices and solves the resulting triangular systems of linear equations. Functions that use direct methods for solving systems of linear equations all compute the solution to $Ax = b$.

### Matrix Factorizations

In some applications, it is desirable to just factor the $n$ x $n$ matrix $A$ into a product of two triangular matrices. This can be done by a constructor of a class for solving the system of linear equations $Ax = b$. The constructor of class `LU` computes the LU factorization of $A$.

Besides the basic matrix factorizations, such as $LU$ and $LL^T$, additional matrix factorizations also are provided. For a real matrix $A$, its $QR$ factorization can be computed using the class `QR`. The class for computing the singular value decomposition (SVD) of a matrix is discussed in a later section.

### Matrix Inversions

The inverse of an $n$ x $n$ nonsingular matrix can be obtained by using the method `inverse` in the classes for solving systems of linear equations. The inverse of a matrix need not be computed if the purpose is to *solve* one or more systems of linear equations. Even with multiple right-hand sides, solving a system of linear equations by computing the inverse and performing matrix multiplication is usually more expensive than the method discussed in the next section.

### Multiple Right-Hand Sides

Consider the case where a system of linear equations has more than one right-hand side vector. It is most economical to find the solution vectors by first factoring the coefficient matrix $A$ into products of triangular matrices. Then, the resulting triangular systems of linear equations are solved for each right-hand side. When $A$ is a real general matrix, access to the $LU$ factorization of $A$ is computed by a constructor of LU. The solution $x_k$ for the $k$-th right-hand side vector, $b_k$ is then found by two triangular solves, $Ly_k = b_k$ and $Ux_k = y_k$. The method `solve` in class `LU` is used to solve each right-hand side. These arguments are found in other functions for solving systems of linear equations.

## Least-Squares Solutions and QR Factorizations

Least-squares solutions are usually computed for an over-determined system of linear equations $A_{m \times n}x = b$, where $m > n$. A least-squares solution $x$ minimizes the Euclidean length of the residual vector $r = Ax - b$. The class QR computes a unique least-squares solution for $x$ when $A$ has full column rank. If $A$ is rank-deficient, then the *base* solution for some variables is computed. These variables consist of the resulting columns after the interchanges. The $QR$ decomposition, with column interchanges or pivoting, is computed such that $AP = QR$. Here, $Q$ is orthogonal, $R$ is upper-trapezoidal with its diagonal elements nonincreasing in magnitude, and $P$ is the permutation matrix determined by the pivoting. The base solution $x_B$ is obtained by solving $R(P^T)x = Q^Tb$ for the base variables. For details, see class `QR`. The QR factorization of a matrix $A$ such that $AP = QR$ with $P$ specified by the user can be computed using keywords.

## Singular Value Decompositions and Generalized Inverses

The SVD of an $m$ x $n$ matrix $A$ is a matrix decomposition $A = USV^T$. With $q = \min(m, n)$, the factors $U_{m \times q}$ and $V_{n \times q}$ are orthogonal matrices, and $S_{q \times q}$ is a nonnegative diagonal matrix with nonincreasing diagonal terms. The class `SVD` computes the singular values of $A$ by default. Part or all of the $U$ and $V$ matrices, an estimate of the rank of $A$, and the generalized inverse of $A$, also can be obtained.

## Ill-Conditioning and Singularity

An $m$ x $n$ matrix $A$, is mathematically singular if there is an $x \neq 0$ such that $Ax = 0$. In this case, the system of linear equations $Ax = b$ does not have a unique solution. On the other hand, a matrix $A$ is *numerically* singular if it is "close" to a mathematically singular matrix. Such problems are called *ill-conditioned*. If the numerical results with an ill-conditioned problem are unacceptable, users can either use more accuracy if it is available (for type *float accuracy* switch to *double*) or they can obtain an *approximate* solution to the system. One form of approximation can be obtained using the SVD of $A$: If $q = \min(m, n)$ and

$$A = \sum_{i=1}^{q} s_{i,i} u_i v_i^T$$

then the approximate solution is given by the following:

$$x_k = \sum_{i=1}^{k} t_{i,i} \left(b^T u_i\right) v_i$$

The scalars $t_{i,i}$ are defined below.

$$t_{i,i} = \begin{cases} s_{i,i}^{-1} & \text{if } \mathrm{s_{i,i}} \geq \mathrm{tol} > 0 \\ 0 & \text{otherwise} \end{cases}$$

The user specifies the value of *tol*. This value determines how "close" the given matrix is to a singular matrix. Further restrictions may apply to the number of terms in the sum, $k \leq q$. For example, there may be a value of $k \leq q$ such that the scalars $\left|b^T u_i\right|$, $i > k$ are smaller than the average uncertainty in the right-hand side $b$. This means that these scalars can be replaced by zero; and hence, $b$ is replaced by a vector that is within the stated uncertainty of the problem.

## *class* Matrix

Matrix manipulation functions.

### Declaration

public class com.imsl.math.Matrix
**extends** java.lang.Object

### Methods

---

- *add*
  `public static double[][]` **add**`( double[][] a, double[][] b )`
  - **Description**
    Add two rectangular arrays, a + b.
  - **Parameters**
    * `a` – a `double` rectangular array
    * `b` – a `double` rectangular array
  - **Returns** – a `double` rectangular array representing the matrix sum of the two arguments
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of either of the input matrices are not uniform, or (2) the matrices are not the same size.

---

- *checkMatrix*
  `public static void` **checkMatrix**`( double[][] a )`

– **Description**
  Check that all of the rows in the matrix have the same length.
– **Parameters**
  * `a` – a `double` matrix
– **Throws**
  * `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix are not uniform.

---

- *CheckMatrix*
  `public static void` **CheckMatrix**`( double[][] a )`

  **Deprecated**

  Check that all of the rows in the matrix have the same length.

  – **Parameters**
    * `a` – a `double` matrix
  – **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix are not uniform.

---

- *checkSquareMatrix*
  `public static void` **checkSquareMatrix**`( double[][] a )`

  – **Description**
    Check that the matrix is square.
  – **Parameters**
    * `a` – a `double` matrix
  – **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when the matrix is not square.

---

- *CheckSquareMatrix*
  `public static void` **CheckSquareMatrix**`( double[][] a )`

  **Deprecated**

  Check that the matrix is square.

  – **Parameters**
    * `a` – a `double` matrix
  – **Throws**

  * `java.lang.IllegalArgumentException` – This exception is thrown when the matrix is not square.

---

- *frobeniusNorm*
  `public static double` **frobeniusNorm**`( double[][]` **a** `)`
  - **Description**
    Return the Frobenius norm of a matrix.
  - **Parameters**
    * `a` – a `double` rectangular array
  - **Returns** – a `double` scalar value equal to the Frobenius norm of the matrix.

---

- *infinityNorm*
  `public static double` **infinityNorm**`( double[][]` **a** `)`
  - **Description**
    Return the infinity norm of a matrix.
  - **Parameters**
    * `a` – a `double` rectangular array
  - **Returns** – a `double` scalar value equal to the maximum of the row sums of the absolute values of the array elements

---

- *multiply*
  `public static double[]` **multiply**`( double[][]` **a**`, double[]` **x** `)`
  - **Description**
    Multiply the rectangular array a and the column array x.
  - **Parameters**
    * `a` – a `double` rectangular matrix
    * `x` – a `double` column array
  - **Returns** – a `double` vector representing the product of the arguments, a*x
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of the input matrix are not uniform, or (2) the number of columns in the input matrix is not equal to the number of elements in the input column vector.

---

- *multiply*
  `public static double[][]` **multiply**`( double[][]` **a**`, double[][]` **b** `)`
  - **Description**
    Multiply two rectangular arrays, a * b.
  - **Parameters**

* `a` – a `double` rectangular array
* `b` – a `double` rectangular array

– **Returns** – the `double` matrix product of a times b

– **Throws**

* `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of either of the input matrices are not uniform, or (2) the number of columns in a is not equal to the number of rows in b.

---

• *multiply*

`public static double[] **multiply**( double[] **x**, double[][] **a** )`

– **Description**
Return the product of the row array x and the rectangular array a.

– **Parameters**

* `x` – a `double` row array
* `a` – a `double` rectangular matrix

– **Returns** – a `double` matrix representing the product of the arguments, x*a.

– **Throws**

* `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of the input matrix are not uniform, or (2) the number of elements in the input vector is not equal to the number of rows of the matrix.

---

• *oneNorm*

`public static double **oneNorm**( double[][] **a** )`

– **Description**
Return the matrix one norm.

– **Parameters**

* `a` – a `double` rectangular array

– **Returns** – a `double` value equal to the maximum of the column sums of the absolute values of the array elements

---

• *subtract*

`public static double[][] **subtract**( double[][] **a**, double[][] **b** )`

– **Description**
Subtract two rectangular arrays, a - b.

– **Parameters**

* `a` – a `double` rectangular array
* `b` – a `double` rectangular array

– **Returns** – a `double` rectangular array representing the matrix difference of the two arguments

- **Throws**
  - `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of either of the input matrices are not uniform, or (2) the matrices are not the same size.

---

- *transpose*
  `public static double[][] transpose( double[][] a )`
  - **Description**
    Return the transpose of a matrix.
  - **Parameters**
    - `a` – a `double` matrix
  - **Returns** – a `double` matrix which is the transpose of the argument
  - **Throws**
    - `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix are not uniform.

### Example: Matrix and PrintMatrix

The 1 norm of a matrix is found using a method from the Matrix class. The matrix is printed using the PrintMatrix class.

```
import com.imsl.math.*;

public class  MatrixEx1 {
    public static void main(String args[]) {
        double nrm1;
        double a[][] = {
            {0., 1., 2., 3.},
            {4., 5., 6., 7.},
            {8., 9., 8., 1.},
            {6., 3., 4., 3.}
        };

        // Get the 1 norm of matrix a
        nrm1 = Matrix.oneNorm(a);

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Simple Matrix");

        // Print the matrix and its 1 norm
        p.print(a);
        System.out.println("The 1 norm of the matrix is "+nrm1);
```

```
    }
}
```

### Output

```
A Simple Matrix
   0  1  2  3
0  0  1  2  3
1  4  5  6  7
2  8  9  8  1
3  6  3  4  3

The 1 norm of the matrix is 20.0
```

## *class* ComplexMatrix

Complex matrix manipulation functions.

### Declaration

public class com.imsl.math.ComplexMatrix
**extends** java.lang.Object

### Methods

---

- *add*
  `public static Complex[][]` **add**`( Complex[][]` **a**`, Complex[][]` **b** `)`
  - **Description**
    Add two rectangular `Complex` arrays, a + b.
  - **Parameters**
    * `a` – a `Complex` rectangular array
    * `b` – a `Complex` rectangular array
  - **Returns** – the `Complex` matrix sum of the two arguments
  - **Throws**

    * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of either of the input matrices are not uniform, or (2) the matrices are not the same size.

---

- *checkMatrix*
  `public static void` **checkMatrix**`( Complex[][]` **a** `)`
  - **Description**
    Check that all of the rows in the `Complex` matrix have the same length.
  - **Parameters**
    * `a` – a `Complex` matrix
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix are not uniform.

---

- *CheckMatrix*
  `public static void` **CheckMatrix**`( Complex[][]` **a** `)`

  ### Deprecated

  Check that all of the rows in the `Complex` matrix have the same length.
  - **Parameters**
    * `a` – a `Complex` matrix
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix are not uniform.

---

- *checkSquareMatrix*
  `public static void` **checkSquareMatrix**`( Complex[][]` **a** `)`
  - **Description**
    Check that the `Complex` matrix is square.
  - **Parameters**
    * `a` – a `Complex` matrix
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when the matrix is not square..

---

- *CheckSquareMatrix*
  `public static void` **CheckSquareMatrix**`( Complex[][]` **a** `)`

**Deprecated**

Check that the `Complex` matrix is square.

- **Parameters**
  - `a` – a `Complex` matrix
- **Throws**
  - `java.lang.IllegalArgumentException` – This exception is thrown when the matrix is not square..

---

- *frobeniusNorm*
  `public static double` **frobeniusNorm**`( Complex[][] a )`
  - **Description**
    Return the Frobenius norm of a `Complex` matrix.
  - **Parameters**
    - `a` – a `Complex` rectangular matrix
  - **Returns** – a `double` value equal to the Frobenius norm of the matrix
  - **Throws**
    - `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix is not uniform.

---

- *infinityNorm*
  `public static double` **infinityNorm**`( Complex[][] a )`
  - **Description**
    Return the infinity norm of a `Complex` matrix.
  - **Parameters**
    - `a` – a `Complex` rectangular matrix
  - **Returns** – a `double` value equal to the maximum of the row sums of the absolute values of the array elements.
  - **Throws**
    - `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix is not uniform.

---

- *multiply*
  `public static Complex[]` **multiply**`( Complex[][] a, Complex[] x )`
  - **Description**
    Multiply the rectangular array a and the column vector x, both `Complex`.
  - **Parameters**
    - `a` – a `Complex` rectangular matrix
    - `x` – a `Complex` vector

- **Returns** – a `Complex` vector containing the product of the arguments, A*x
- **Throws**
  * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of the input matrix are not uniform, and (2) the number of columns in the input matrix is not equal to the number of elements in the input vector.

---

- *multiply*
  `public static Complex[][]` **multiply**`( Complex[][]` **a**`, Complex[][]` **b** `)`
  - **Description**
    Multiply two `Complex` rectangular arrays, a * b.
  - **Parameters**
    * `a` – a `Complex` rectangular array
    * `b` – a `Complex` rectangular array
  - **Returns** – the `Complex` matrix product of a times b
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of either of the input matrices are not uniform, or (2) the number of columns in a is not equal to the number of rows in b.

---

- *multiply*
  `public static Complex[]` **multiply**`( Complex[]` **x**`, Complex[][]` **a** `)`
  - **Description**
    Return the product of the row vector x and the rectangular array a, both `Complex`.
  - **Parameters**
    * `x` – a `Complex` row vector
    * `a` – a `Complex` rectangular matrix
  - **Returns** – a `Complex` vector containing the product of the arguments, x*A.
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of the input matrix are not uniform, or (2) the number of elements in the input vector is not equal to the number of rows of the matrix.

---

- *oneNorm*
  `public static double` **oneNorm**`( Complex[][]` **a** `)`
  - **Description**
    Return the `Complex` matrix one norm.
  - **Parameters**

- * `a` – a `Complex` rectangular array
- **Returns** – a `double` value equal to the maximum of the column sums of the absolute values of the array elements
- **Throws**
  - * `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix is not uniform.

---

- *subtract*
  `public static Complex[][] subtract( Complex[][] a, Complex[][] b )`
  - **Description**
    Subtract two `Complex` rectangular arrays, a - b.
  - **Parameters**
    - * `a` – a `Complex` rectangular array
    - * `b` – a `Complex` rectangular array
  - **Returns** – the `Complex` matrix difference of the two arguments.
  - **Throws**
    - * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of either of the input matrices are not uniform, or (2) the matrices are not the same size.

---

- *transpose*
  `public static Complex[][] transpose( Complex[][] a )`
  - **Description**
    Return the transpose of a `Complex` matrix.
  - **Parameters**
    - * `a` – a `Complex` matrix
  - **Returns** – the `Complex` matrix transpose of the argument
  - **Throws**
    - * `java.lang.IllegalArgumentException` – This exception is thrown when the lengths of the rows of the input matrix are not uniform.

### Example: Print a Complex Matrix

A Complex matrix is initialized and printed.

```
import com.imsl.math.*;

public class  ComplexMatrixEx1 {
    public static void main(String args[]) {
        Complex a[][] = {
            {new Complex(1,3), new Complex(3,5), new Complex(7,9)},
```

```
            {new Complex(8,7), new Complex(9,5), new Complex(1,9)},
            {new Complex(2,9), new Complex(6,9), new Complex(7,3)},
            {new Complex(5,4), new Complex(8,4), new Complex(5,9)}
        };

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Complex Matrix");

        // Print the matrix
        p.print(a);
    }
}
```

### Output

```
  A Complex Matrix
     0     1     2
0  1+3i  3+5i  7+9i
1  8+7i  9+5i  1+9i
2  2+9i  6+9i  7+3i
3  5+4i  8+4i  5+9i
```

## *class* **LU**

LU factorization of a matrix of type `double`.

`LU` performs an $LU$ factorization of a real general coefficient matrix. The `condition` method estimates the condition number of the matrix. The LU factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same infinity norm.

The $L_1$ condition number of the matrix $A$ is defined to be $\kappa(A) = ||A||_1||A^{-1}||_1$. Since it is expensive to compute $||A^{-1}||_1$, the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described in a paper by Cline et al. (1979).

An estimated condition number greater than $1/\epsilon$ (where $\epsilon$ is machine precision) indicates that very small changes in $A$ can cause very large changes in the solution $x$. Iterative refinement can sometimes find the solution to such a system.

`LU` fails if $U$, the upper triangular part of the factorization, has a zero diagonal element. This can occur only if $A$ either is singular or is very close to a singular matrix.

Use the `solve` method to solve systems of equations. The `determinant` method can be called to compute the determinant of the coefficient matrix.

`LU` is based on the LINPACK routine `SGECO`; see Dongarra et al. (1979). `SGECO` uses unscaled partial pivoting.

### Declaration

public class com.imsl.math.LU
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Fields

- protected `double[][]` **factor**
  - LU factorization of A with partial pivoting
- protected `int[]` **ipvt**
  - Pivot sequence for the factorization

### Constructor

- *LU*
  `public` **LU**`( double[][]` **a** `) throws com.imsl.math.SingularMatrixException`
  - **Description**
    Creates the LU factorization of a square matrix of type `double`.
  - **Parameters**
    * `a` – the `double` square matrix to be factored
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when the row lengths of input matrix are not equal (for example, the matrix edges are “jagged”.)
    * `com.imsl.math.SingularMatrixException` – is thrown when the input matrix is singular.

## Methods

- *condition*
  `public double` **condition**`( double[][]` **a** `)`
  - **Description**
    Return an estimate of the reciprocal of the L1 condition number of a matrix.
  - **Parameters**
    * `a` – the `double` square matrix for which the reciprocal of the L1 condition number is desired
  - **Returns** – a `double` value representing an estimate of the reciprocal of the L1 condition number of the matrix

- *determinant*
  `public double` **determinant**`( )`
  - **Description**
    Return the determinant of the matrix used to construct this instance.
  - **Returns** – a `double` scalar containing the determinant of the matrix used to construct this instance

- *inverse*
  `public double[][]` **inverse**`( )`
  - **Description**
    Returns the inverse of the matrix used to construct this instance.
  - **Returns** – a `double` matrix representing the inverse of the matrix used to construct this instance

- *solve*
  `public double[]` **solve**`( double[]` **b** `)`
  - **Description**
    Return the solution $x$ of the linear system $Ax = b$ using the LU factorization of A.
  - **Parameters**
    * `b` – a `double` array containing the right-hand side of the linear system
  - **Returns** – a `double` array containing the solution to the linear system of equations

- *solve*
  `public static double[]` **solve**`( double[][]` **a**`, double[]` **b** `) throws com.imsl.math.SingularMatrixException`

- **Description**
  Solve ax=b for x using the LU factorization of a.
- **Parameters**
  * `a` – a `double` square matrix
  * `b` – a `double` column vector
- **Returns** – a `double` column vector containing the solution to the linear system of equations
- **Throws**
  * `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of the input matrix are not uniform, and (2) the number of rows in the input matrix is not equal to the number of elements in x.
  * `com.imsl.math.SingularMatrixException` – is thrown when the matrix is singular.

---

- *solveTranspose*
  `public double[]` **solveTranspose**`( double[]` **b** `)`
  - **Description**
    Return the solution x of the linear system $A^T = b$.
  - **Parameters**
    * `b` – `double` array containing the right-hand side of the linear system
  - **Returns** – `double` array containing the solution to the linear system of equations

### Example: LU Factorization of a Matrix

The LU Factorization of a Matrix is performed. A linear system is then solved using the factorization. The inverse, determinant, and condition number of the input matrix are also computed.

```
import com.imsl.math.*;

public class LUEx1 {
    public static void main(String args[]) throws SingularMatrixException {
        double a[][] = {
            {1, 3, 3},
            {1, 3, 4},
            {1, 4, 3}
        };
        double b[] = {12, 13, 14};

        // Compute the LU factorization of A
```

```
        LU lu = new LU(a);

        // Solve Ax = b
        double x[] = lu.solve(b);
        new PrintMatrix("x").print(x);

        // Find the inverse of A.
        double ainv[][] = lu.inverse();
        new PrintMatrix("ainv").print(ainv);

        // Find the condition number of A.
        double condition = lu.condition(a);
        System.out.println("condition number = "+condition);
        System.out.println();

        // Find the determinant of A.
        double determinant = lu.determinant();
        System.out.println("determinant = "+determinant);
    }
}
```

### Output

```
  x
   0
0  3
1  2
2  1

     ainv
   0   1   2
0   7  -3  -3
1  -1   0   1
2  -1   1   0

condition number = 0.015120274914089344

determinant = -0.9999999999999998
```

# *class* ComplexLU

LU factorization of a matrix of type `Complex`.

`ComplexLU` performs an $LU$ factorization of a complex general coefficient matrix. `ComplexLU`'s method `condition` estimates the condition number of the matrix. The $LU$ factorization is done using scaled partial pivoting. Scaled partial pivoting differs from partial pivoting in that the pivoting strategy is the same as if each row were scaled to have the same infinity norm.

The $L_1$ condition number of the matrix $A$ is defined to be $\kappa(A) = \|A\|_1 \left\|A^{-1}\right\|_1$. Since it is expensive to compute $\left\|A^{-1}\right\|_1$, the condition number is only estimated. The estimation algorithm is the same as used by LINPACK and is described by Cline et al. (1979).

An estimated condition number greater than $1/\epsilon$ (where $\epsilon$ is machine precision) indicates that very small changes in $A$ can cause very large changes in the solution $x$. Iterative refinement can sometimes find the solution to such a system.

`ComplexLU` fails if $U$, the upper triangular part of the factorization, has a zero diagonal element. This can occur only if $A$ either is singular or is very close to a singular matrix.

The `solve` method can be used to solve systems of equations. The method `determinant` can be called to compute the determinant of the coefficient matrix.

ComplexLU is based on the LINPACK routine `CGECO`; see Dongarra et al. (1979). `CGECO` uses unscaled partial pivoting.

## Declaration

public class com.imsl.math.ComplexLU
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Fields

---

- protected `Complex[][]` **factor**
  - LU factorization of A with partial pivoting
- protected `int[]` **ipvt**
  - Pivot sequence for the factorization

## Constructor

---

- *ComplexLU*
  `public` **ComplexLU**`( Complex[][]` **a** `) throws com.imsl.math.SingularMatrixException`
  - **Description**
    Creates the LU factorization of a square matrix of type `Complex`.
  - **Parameters**
    * `a` – `Complex` square matrix to be factored
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when the row lengths of input matrix are not equal (for example, the matrix edges are "jagged".)
    * `com.imsl.math.SingularMatrixException` – is thrown when the input matrix is singular.

## Methods

---

- *condition*
  `public double` **condition**`( Complex[][]` **a** `)`
  - **Description**
    Return an estimate of the reciprocal of the L1 condition number.
  - **Parameters**
    * `a` – a `Complex` matrix
  - **Returns** – a `double` scalar value representing the estimate of the reciprocal of the L1 condition number of the matrix a

---

- *determinant*
  `public Complex` **determinant**`( )`
  - **Description**
    Return the determinant of the matrix used to construct this instance.
  - **Returns** – a `Complex` scalar containing the determinant of the matrix used to construct this instance

---

- *inverse*
  `public Complex[][]` **inverse**`( )`

– **Description**
Compute the inverse of a matrix of type `Complex`.

– **Returns** – a `Complex` matrix containing the inverse of the matrix used to construct this object.

---

- *solve*
`public Complex[] solve( Complex[] b )`

  – **Description**
  Return the solution x of the linear system Ax = b using the LU factorization of A.

  – **Parameters**
    - `b` – `Complex` array containing the right-hand side of the linear system

  – **Returns** – `Complex` array containing the solution to the linear system of equations

---

- *solve*
`public static Complex[] solve( Complex[][] a, Complex[] b ) throws com.imsl.math.SingularMatrixException`

  – **Description**
  Solve ax=b for x using the LU factorization of a.

  – **Parameters**
    - `a` – a `Complex` square matrix
    - `b` – a `Complex` column vector

  – **Returns** – a `Complex` column vector containing the solution to the linear system of equations.

  – **Throws**
    - `java.lang.IllegalArgumentException` – This exception is thrown when (1) the lengths of the rows of the input matrix are not uniform, and (2) the number of rows in the input matrix is not equal to the number of elements in x.
    - `com.imsl.math.SingularMatrixException` – is thrown when the matrix is singular.

---

- *solveTranspose*
`public Complex[] solveTranspose( Complex[] b )`

  – **Description**
  Return the solution x of the linear system $A^T x = b$.

  – **Parameters**
    - `b` – `Complex` array containing the right-hand side of the linear system

  – **Returns** – `Complex` array containing the solution to the linear system of equations

### Example: LU Decomposition of a Complex Matrix

The Complex class is used to convert a real matrix to a Complex matrix. An LU decomposition of the matrix is performed and the determinant and condition number of the matrix are obtained.

```
import com.imsl.math.*;

public class ComplexLUEx1 {
    public static void main(String args[]) throws SingularMatrixException {
        double ar[][] = {
            {1, 3, 3},
            {1, 3, 4},
            {1, 4, 3}
        };
        double br[] = {12, 13, 14};

        Complex a[][] = new Complex[3][3];
        Complex b[] = new Complex[3];

        for (int i = 0; i < 3; i++){
            b[i] = new Complex(br[i]);
            for (int j = 0; j < 3; j++) {
                a[i][j] = new Complex(ar[i][j]);
            }
        }

        // Compute the LU factorization of A
        ComplexLU clu = new ComplexLU(a);

        // Solve Ax = b
        Complex x[] = clu.solve(b);
        System.out.println("The solution is:");
        System.out.println(" ");
        new PrintMatrix("x").print(x);

        // Find the condition number of A.
        double condition = clu.condition(a);
        System.out.println("The condition number = "+condition);
        System.out.println();

        // Find the determinant of A.
        Complex determinant = clu.determinant();
```

```
        System.out.println("The determinant = "+determinant);
    }
}
```

### Output

```
The solution is:

  x
   0
0  3
1  2
2  1

The condition number = 0.014886731391585757

The determinant = -0.9999999999999998
```

## *class* Cholesky

Cholesky factorization of a matrix of type `double`.

Class `Cholesky` is based on the LINPACK routine `SCHDC`; see Dongarra et al. (1979).

Before the decomposition is computed, initial elements are moved to the leading part of $A$ and final elements to the trailing part of $A$. During the decomposition only rows and columns corresponding to the free elements are moved. The result of the decomposition is an upper triangular matrix $R$ and a permutation matrix $P$ that satisfy $P^T AP = R^T R$, where $P$ is represented by `ipvt`.

The method `update` is based on the LINPACK routine `SCHUD`; see Dongarra et al. (1979).

The Cholesky factorization of a matrix is $A = R^T R$, where $R$ is an upper triangular matrix. Given this factorization, `downdate` computes the factorization

$$A - xx^T = \tilde{R}^T \tilde{R}$$

`downdate` determines an orthogonal matrix $U$ as the product $G_N \ldots G_1$ of Givens rotations, such that

$$U \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ x^T \end{bmatrix}$$

By multiplying this equation by its transpose and noting that $U^T U = I$, the desired result

$$R^T R - xx^T = \tilde{R}^T \tilde{R}$$

is obtained.

Let $a$ be the solution of the linear system $R^T a = x$ and let

$$\alpha = \sqrt{1 - \|a\|_2^2}$$

The Givens rotations, $G_i$, are chosen such that

$$G_1 \cdots G_N \begin{bmatrix} a \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The $G_i$, are *(N + 1) * (N + 1)* matrices of the form

$$G_i = \begin{bmatrix} I_{i-1} & 0 & 0 & 0 \\ 0 & c_i & 0 & -s_i \\ 0 & 0 & I_{N-i} & 0 \\ 0 & s_i & 0 & c_i \end{bmatrix}$$

where $I_k$ is the identity matrix of order $k$; and $c_i = \cos\theta_i, s_i = \sin\theta_i$ for some $\theta_i$.

The Givens rotations are then used to form

$$\tilde{R},\ G_1 \cdots G_N \begin{bmatrix} R \\ 0 \end{bmatrix} = \begin{bmatrix} \tilde{R} \\ \tilde{x}^T \end{bmatrix}$$

The matrix

$$\tilde{R}$$

is upper triangular and

$$\tilde{x} = x$$

because

$$x = \left(R^T 0\right) \begin{bmatrix} a \\ \alpha \end{bmatrix} = \left(R^T 0\right) U^T U \begin{bmatrix} a \\ \alpha \end{bmatrix} = \left(\tilde{R}^T \tilde{x}\right) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \tilde{x}$$

.

## Declaration

```
public class com.imsl.math.Cholesky
extends java.lang.Object
implements java.io.Serializable, java.lang.Cloneable
```

### Inner Class

#### *class* **Cholesky.NotSPDException**

The matrix is not symmetric, positive definite.

#### Declaration

public static class com.imsl.math.Cholesky.NotSPDException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *Cholesky.NotSPDException*
  `public` **Cholesky.NotSPDException**`( )`

### Constructor

---

- *Cholesky*
  `public` **Cholesky**`( double[][] a ) throws com.imsl.math.SingularMatrixException, com.imsl.math.Cholesky.NotSPDException`
  - **Description**
    Create the Cholesky factorization of a symmetric positive definite matrix of type `double`.
  - **Parameters**
    * `a` – a `double` square matrix to be factored
  - **Throws**
    * `java.lang.IllegalArgumentException` – Thrown when the row lengths of matrix a are not equal (for example, the matrix edges are "jagged".)
    * `com.imsl.math.SingularMatrixException` – Thrown when the input matrix a is singular.
    * `com.imsl.math.Cholesky.NotSPDException` – Thrown when the input matrix is not symmetric, positive definite.

## Methods

- *downdate*
`public void` **downdate**`( double[] x ) throws com.imsl.math.Cholesky.NotSPDException`

  - **Description**
  Downdates the factorization by subtracting a rank-1 matrix. The object will contain the Cholesky factorization of $a - x \times x^T$, where a is the previously factored matrix.
  - **Parameters**
    - `x` – A `double` array which specifies the rank-1 matrix. `x` is not modified by this function.
  - **Throws**
    - `com.imsl.math.Cholesky.NotSPDException` – if $a - x \times x^T$ is not symmetric positive-definite.

- *getR*
`public double[][]` **getR**`( )`

  - **Description**
  Returns the R matrix that results from the Cholesky factorization. R is a lower triangular matrix and $A = RR^T$.
  - **Returns** – a `double` matrix which contains the R matrix that results from the Cholesky factorization

- *inverse*
`public double[][]` **inverse**`( )`

  - **Description**
  Returns the inverse of this matrix
  - **Returns** – a `double` matrix containing the inverse

- *solve*
`public double[]` **solve**`( double[] b )`

  - **Description**
  Solve Ax = b where A is a positive definite matrix with elements of type `double`.
  - **Parameters**
    - `b` – a `double` array containing the right-hand side of the linear system
  - **Returns** – a `double` array containing the solution to the system of linear equations

- *update*
  `public void` **update**`( double[] x )`
  - **Description**
    Updates the factorization by adding a rank-1 matrix. The object will contain the Cholesky factorization of $a + x * X^T = b$, where $a$ is the previously factored matrix.
  - **Parameters**
    * `x` – A `double` array which specifies the rank-1 matrix. `x` is not modified by this function.

### Example: Cholesky Factorization

The Cholesky Factorization of a matrix is performed as well as its inverse.

```
import com.imsl.math.*;

public class CholeskyEx1 {
    public static void main(String args[]) throws com.imsl.IMSLException {
        double a[][] = {
            { 1, -3,  2},
            {-3, 10, -5},
            { 2, -5,  6}
        };
        double b[] = {27, -78, 64};

        // Compute the Cholesky factorization of A
        Cholesky cholesky = new Cholesky(a);

        // Solve Ax = b
        double x[] = cholesky.solve(b);
        new PrintMatrix("x").print(x);

        // Find the inverse of A.
        double ainv[][] = cholesky.inverse();
        new PrintMatrix("ainv").print(ainv);
    }
}
```

### Output

```
   x
   0
0   1
1  -4
2   7

     ainv
   0   1   2
0  35   8  -5
1   8   2  -1
2  -5  -1   1
```

## *class* QR

QR Decomposition of a matrix.

Class `QR` computes the $QR$ decomposition of a matrix using Householder transformations. It is based on the LINPACK routine `SQRDC`; see Dongarra et al. (1979).

`QR` determines an orthogonal matrix $Q$, a permutation matrix $P$, and an upper trapezoidal matrix $R$ with diagonal elements of nonincreasing magnitude, such that $AP = QR$. The Householder transformation for column $k$ is of the form

$$I - \frac{u_k u_k^T}{P_k}$$

for $k = 1, 2, \ldots$, min(number of rows of $A$, number of columns of $A$), where u has zeros in the first *k - 1* positions. The matrix $Q$ is not produced directly by `QR`. Instead the information needed to reconstruct the Householder transformations is saved. If the matrix $Q$ is needed explicitly, the method `getQ` can be called after `QR`. This method accumulates $Q$ from its factored form.

Before the decomposition is computed, initial columns are moved to the beginning of the array A and the final columns to the end. Both initial and final columns are frozen in place during the computation. Only free columns are pivoted. Pivoting is done on the free columns of largest reduced norm.

## Declaration

public class com.imsl.math.QR
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *QR*
  `public QR( double[][] a )`
  - **Description**
    Constructs the QR decomposition of a matrix with elements of type `double`.
  - **Parameters**
    * `a` – a `double` matrix to be factored
  - **Throws**
    * `java.lang.IllegalArgumentException` – Thrown when the row lengths of input matrix a are not equal (i.e. the matrix edges are "jagged".)

## Methods

- *getPermute*
  `public int[] getPermute( )`
  - **Description**
    Returns an integer vector containing information about the permutation of the elements of the matrix during pivoting.
  - **Returns** – an `int` array containing the permutation information. The $k$-th element contains the index of the column of the matrix that has been interchanged into the $k$-th column.

- *getQ*
  `public double[][] getQ( )`
  - **Description**
    Returns the orthogonal or unitary matrix Q.
  - **Returns** – a `double` matrix containing the accumulated orthogonal matrix Q from the QR decomposition

- *getR*
  `public double[][]` **`getR`**`( )`
  - **Description**
    Returns the upper trapezoidal matrix R.
  - **Returns** – the upper trapezoidal `double` matrix R of the QR decomposition

---

- *getRank*
  `public int` **`getRank`**`( )`
  - **Description**
    Returns the rank of the matrix used to construct this instance.
  - **Returns** – an `int` specifying the rank of the matrix used to construct this instance

---

- *rank*
  `public int` **`rank`**`( double` **`tolerance`** `)`
  - **Description**
    Returns the rank of the matrix given an input tolerance.
  - **Parameters**
    * `tolerance` – a `double` scalar value used in determining the rank of the matrix
  - **Returns** – an `int` specifying the rank of the matrix

---

- *solve*
  `public double[]` **`solve`**`( double[]` **`b`** `) throws com.imsl.math.SingularMatrixException`
  - **Description**
    Returns the solution to the least-squares problem Ax = b.
  - **Parameters**
    * `b` – a `double` array to be manipulated
  - **Returns** – a `double` array containing the solution vector to Ax = b with components corresponding to the unused columns set to zero
  - **Throws**
    * `com.imsl.math.SingularMatrixException` – Thrown when the upper triangular matrix R resulting from the QR factorization is singular.

---

- *solve*
  `public double[]` **`solve`**`( double[]` **`b`**`, double` **`tol`** `) throws com.imsl.math.SingularMatrixException`

– **Description**
  Returns the solution to the least-squares problem Ax = b using an input tolerance.
– **Parameters**
  * `b` – a `double` array to be manipulated
  * `tol` – a `double` scalar value used in determining the rank of A
– **Returns** – a `double` array containing the solution vector to Ax = b with components corresponding to the unused columns set to zero
– **Throws**
  * `com.imsl.math.SingularMatrixException` – Thrown when the upper triangular matrix R resulting from the QR factorization is singular.

## Example: QR Factorization of a Matrix

The QR Factorization of a Matrix is performed. A linear system is then solved using the factorization. The rank of the input matrix is also computed.

```
import com.imsl.math.*;

public class QREx1 {
    public static void main(String args[]) throws SingularMatrixException {
        double a[][] = {
            {1, 2,  4},
            {1, 4, 16},
            {1, 6, 36},
            {1, 8, 64}
        };
        double b[]  = {4.999, 9.001, 12.999, 17.001};

        // Compute the QR factorization of A
        QR qr = new QR(a);

        // Solve Ax = b
        double x[] = qr.solve(b);
        new PrintMatrix("x").print(x);

        // Print Q and R.
        new PrintMatrix("Q").print(qr.getQ());
        new PrintMatrix("R").print(qr.getR());

        // Find the rank of A.
        int rank = qr.getRank();
        System.out.println("rank = "+rank);
```

```
    }
}
```

**Output**

```
     x
     0
0   0.999
1   2
2  -0

                 Q
     0       1       2       3
0  -0.053  -0.542   0.808  -0.224
1  -0.213  -0.657  -0.269   0.671
2  -0.478  -0.346  -0.449  -0.671
3  -0.85    0.393   0.269   0.224

             R
     0        1       2
0  -75.26  -10.63   -1.594
1    0      -2.647  -1.153
2    0       0       0.359
3    0       0       0

rank = 3
```

## *class* **SVD**

Singular Value Decomposition (SVD) of a rectangular matrix of type `double`.

`SVD` is based on the LINPACK routine `SSVDC`; see Dongarra et al. (1979).

Let $n$ be the number of rows in $A$ and let $p$ be the number of columns in $A$. For any $n$ x $p$ matrix $A$, there exists an $n$ x $n$ orthogonal matrix $U$ and a $p$ x $p$ orthogonal matrix $V$ such that

$$U^TAV = \begin{cases} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} & \text{if } n \ge p \\ [\Sigma\ 0] & \text{if } n \le p \end{cases}$$

where $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_m)$, and $m = \min(n, p)$. The scalars $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_m \geq 0$ are called the *singular values* of $A$. The columns of $U$ are called the *left singular vectors* of $A$. The columns of $V$ are called the *right singular vectors* of $A$.

The estimated rank of $A$ is the number of $\sigma_k$ that is larger than a tolerance $\eta$. If $\tau$ is the parameter `tol` in the program, then

$$\eta = \begin{cases} \tau & \text{if } \tau > 0 \\ |\tau| \, \|A\|_\infty & \text{if } \tau < 0 \end{cases}$$

The Moore-Penrose generalized inverse of the matrix is computed by partitioning the matricies $U$, $V$ and $\Sigma$ as $U = (U_1, U_2)$, $V = (V_1, V_2)$ and $\Sigma_1 = \text{diag}(\sigma_1, \ldots, \sigma_k)$ where the "1" matrices are $k$ by $k$. The Moore-Penrose generalized inverse is $V_1 \Sigma_1^{-1} U_1^T$.

## Declaration

public class com.imsl.math.SVD
**extends** java.lang.Object

## Inner Class

### *class* SVD.DidNotConvergeException

The iteration did not converge

#### Declaration

public static class com.imsl.math.SVD.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *SVD.DidNotConvergeException*
  `public` **SVD.DidNotConvergeException**`( java.lang.String` **message** `)`
- *SVD.DidNotConvergeException*
  `public` **SVD.DidNotConvergeException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

## Constructors

- *SVD*
  `public SVD( double[][] a ) throws com.imsl.math.SVD.DidNotConvergeException`
  - **Description**
    Construct the singular value decomposition of a rectangular matrix with default tolerance. The tolerance used is 2.2204460492503e-14. This tolerance is used to determine rank. A singular value is considered negligible if the singular value is less than or equal to this tolerance.
  - **Parameters**
    * `a` – a `double` matrix for which the singular value decomposition is to be computed
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when the row lengths of input matrix a are not equal (i.e. the matrix edges are “jagged”)

- *SVD*
  `public SVD( double[][] a, double tol ) throws com.imsl.math.SVD.DidNotConvergeException`
  - **Description**
    Construct the singular value decomposition of a rectangular matrix with a given tolerance. If `tol` is positive, then a singular value is considered negligible if the singular value is less than or equal to `tol`. If `tol` is negative, then a singular value is considered negligible if the singular value is less than or equal to the absolute value of the product of `tol` and the infinity norm of the input matrix. In the latter case, the absolute value of `tol` generally contains an estimate of the level of the relative error in the data.
  - **Parameters**
    * `a` – a `double` matrix for which the singular value decomposition is to be computed
    * `tol` – a `double` scalar containing the tolerance used to determine when a singular value is negligible
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when the row lengths of input matrix a are not equal (for example, the matrix edges are “jagged”)
    * `com.imsl.math.SVD.DidNotConvergeException` – is thrown when the rank cannot be determined because convergence was not obtained for all singular values

## Methods

- *getInfo*
  `public int` **getInfo**`( )`
  - **Description**
    Returns convergence information about S, U, and V.
  - **Returns** – Convergence was obtained for the info, info+1, ..., min(nra,nca) singular values and their corresponding vectors. Here, nra and nca represent the number of rows and columns of the input matrix respectively.

- *getRank*
  `public int` **getRank**`( )`
  - **Description**
    Returns the rank of the matrix used to construct this instance.
  - **Returns** – an `int` scalar containing the rank of the matrix used to construct this instance. The estimated rank of the input matrix is the number of singular values which are larger than a tolerance.

- *getS*
  `public double[]` **getS**`( )`
  - **Description**
    Returns the singular values.
  - **Returns** – a `double` array containing the singular values of the matrix

- *getU*
  `public double[][]` **getU**`( )`
  - **Description**
    Returns the left singular vectors.
  - **Returns** – a `double` matrix containing the left singular vectors

- *getV*
  `public double[][]` **getV**`( )`
  - **Description**
    Returns the right singular vectors.
  - **Returns** – a `double` matrix containing the right singular vectors

- *inverse*
  `public double[][]` **inverse**`( )`

– **Description**
  Compute the Moore-Penrose generalized inverse of a real matrix.
– **Returns** – a `double` matrix containing the generalized inverse of the matrix used to construct this instance

### Example: Singular Value Decomposition of a Matrix

The singular value decomposition of a matrix is performed. The rank of the matrix is also computed.

```
import com.imsl.math.*;

public class SVDEx1 {
    public static void main(String args[]) throws SVD.DidNotConvergeException {
        double a[][] = {
            {1, 2, 1, 4},
            {3, 2, 1, 3},
            {4, 3, 1, 4},
            {2, 1, 3, 1},
            {1, 5, 2, 2},
            {1, 2, 2, 3}
        };

        // Compute the SVD factorization of A
        SVD svd = new SVD(a);

        // Print U, S and V.
        new PrintMatrix("U").print(svd.getU());
        new PrintMatrix("S").print(svd.getS());
        new PrintMatrix("V").print(svd.getV());

        // Find the rank of A.
        int rank = svd.getRank();
        System.out.println("rank = "+rank);
    }
}
```

### Output

```
                         U
     0       1       2       3       4       5
```

```
0  -0.38    0.12    0.439  -0.565   0.024  -0.573
1  -0.404   0.345  -0.057   0.215   0.809   0.119
2  -0.545   0.429   0.051   0.432  -0.572   0.04
3  -0.265  -0.068  -0.884  -0.215  -0.063  -0.306
4  -0.446  -0.817   0.142   0.321   0.062  -0.08
5  -0.355  -0.102  -0.004  -0.546  -0.099   0.746

     S
     0
0  11.485
1   3.27
2   2.653
3   2.089

                 V
     0       1       2       3
0  -0.444   0.556  -0.435   0.552
1  -0.558  -0.654   0.277   0.428
2  -0.324  -0.351  -0.732  -0.485
3  -0.621   0.374   0.444  -0.526

rank = 4
```

# *class* **SingularMatrixException**

The matrix is singular.

## Declaration

public class com.imsl.math.SingularMatrixException
**extends** com.imsl.IMSLException (page 1240)

## Constructor

- *SingularMatrixException*
  `public` **SingularMatrixException**`( )`

# Chapter 2

# Eigensystem Analysis

Classes



## Usage Notes

An ordinary linear eigensystem problem is represented by the equation $Ax = \lambda x$ where $A$ denotes an $n$ x $n$ matrix. The value $\lambda$ is an *eigenvalue* and $x \neq 0$ is the corresponding *eigenvector*. The eigenvector is determined up to a scalar factor. In all functions, we have chosen this factor so that $x$ has Euclidean length one, and the component of $x$ of largest magnitude is positive. If $x$ is a complex vector, this component of largest magnitude is scaled to be real and positive. The entry where this component occurs can be arbitrary for eigenvectors having nonunique maximum magnitude values.

## Error Analysis and Accuracy

Except in special cases, functions will not return the exact eigenvalue-eigenvector pair for the ordinary eigenvalue problem $Ax = \lambda x$. Typically, the computed pair

$$\tilde{x},\ \tilde{\lambda}$$

are an exact eigenvector-eigenvalue pair for a "nearby" matrix A + E. Information about

$E$ is known only in terms of bounds of the form $\|E\|_2 \le f(n)\|A\|_2 \varepsilon$. The value of *f(n)* depends on the algorithm, but is typically a small fractional power of $n$. The parameter $\varepsilon$ is the machine precision. By a theorem due to Bauer and Fike (see Golub and Van Loan 1989, p. 342),

$$\min\left|\tilde{\lambda} - \lambda\right| \le \kappa(X)\|E\|_2 \quad \text{for all}\, \lambda \,\text{in}\, \sigma(A)$$

where $\sigma(A)$ is the set of all eigenvalues of $A$ (called the *spectrum* of $A$), $X$ is the matrix of eigenvectors, $\|\cdot\|_2$ is Euclidean length, and $\kappa(X)$ is the condition number of $X$ defined as $\kappa(X) = \|X\|_2 \left\|X^{-1}\right\|_2$. If $A$ is a real symmetric or complex Hermitian matrix, then its eigenvector matrix $X$ is respectively orthogonal or unitary. For these matrices, $\kappa(X) = 1$.

The accuracy of the computed eigenvalues

$$\tilde{\lambda}_j$$

and eigenvectors

$$\tilde{x}_j$$

can be checked by computing their performance index $\tau$. The performance index is defined to be

$$\tau = \max_{1 \le j \le n} \frac{\left\|A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j\right\|_2}{n\varepsilon \|A\|_2 \|\tilde{x}_j\|_2}$$

where $\varepsilon$ is again the machine precision.

The performance index $\tau$ is related to the error analysis because

$$\|E\tilde{x}_j\|_2 = \left\|A\tilde{x}_j - \tilde{\lambda}_j \tilde{x}_j\right\|_2$$

where $E$ is the "nearby" matrix discussed above.

While the exact value of $\tau$ is precision and data dependent, the performance of an eigensystem analysis function is defined as excellent if $\tau < 1$, good if $1 \le \tau \le 100$, and poor if $\tau > 100$. This is an arbitrary definition, but large values of $\tau$ can serve as a warning that there is a significant error in the calculation.

If the condition number $\kappa(X)$ of the eigenvector matrix $X$ is large, there can be large errors in the eigenvalues even if $\tau$ is small. In particular, it is often difficult to recognize near multiple eigenvalues or unstable mathematical problems from numerical results. This facet of the eigenvalue problem is often difficult for users to understand. Suppose the accuracy of an individual eigenvalue is desired. This can be answered approximately by computing the *condition number of an individual eigenvalue*(see Golub and Van Loan 1989, pp. 344-345). For matrices $A$, such that the computed array of normalized eigenvectors $X$ is invertible, the condition number of $\lambda_i$ is

$$\kappa_j = \left\| e_j^T X^{-1} \right\|,$$

the Euclidean length of the $j$-th row of $X^{-1}$. Users can choose to compute this matrix using the class LU in "Linear Systems." An approximate bound for the accuracy of a computed eigenvalue is then given by $\kappa_j \varepsilon \|A\|$. To compute an approximate bound for the relative accuracy of an eigenvalue, divide this bound by $|\lambda_j|$.

## *class* **Eigen**

Collection of Eigen System functions.

`Eigen` computes the eigenvalues and eigenvectors of a real matrix. The matrix is first balanced. Orthogonal similarity transformations are used to reduce the balanced matrix to a real upper Hessenberg matrix. The implicit double-shifted QR algorithm is used to compute the eigenvalues and eigenvectors of this Hessenberg matrix. The eigenvectors are normalized such that each has Euclidean length of value one. The largest component is real and positive.

The balancing routine is based on the EISPACK routine `BALANC`. The reduction routine is based on the EISPACK routines `ORTHES` and `ORTRAN`. The QR algorithm routine is based on the EISPACK routine `HQR2`. See Smith et al. (1976) for the EISPACK routines. Further details, some timing data, and credits are given in Hanson et al. (1990).

While the exact value of the performance index, $\tau$, is highly machine dependent, the performance of `Eigen` is considered excellent if $\tau < 1$, good if $1 \le \tau \le 100$, and poor if $\tau > 100$.

The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124-125).

### Declaration

public class com.imsl.math.Eigen
**extends** java.lang.Object

### Inner Class

### *class* **Eigen.DidNotConvergeException**

The iteration did not converge

#### Declaration

public static class com.imsl.math.Eigen.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *Eigen.DidNotConvergeException*
  `public` **Eigen.DidNotConvergeException**`( java.lang.String` **message** `)`

- *Eigen.DidNotConvergeException*
  `public` **Eigen.DidNotConvergeException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

### Constructors

---

- *Eigen*
  `public` **Eigen**`( double[][]` **a** `) throws`
  `com.imsl.math.Eigen.DidNotConvergeException`
  - **Description**
    Constructs the eigenvalues and the eigenvectors of a real square matrix.
  - **Parameters**
    * `a` – is the `double` square matrix whose eigensystem is to be constructed
  - **Throws**
    * `com.imsl.math.Eigen.DidNotConvergeException` – is thrown when the algorithm fails to converge on the eigenvalues of the matrix.

- *Eigen*
  `public` **Eigen**`( double[][]` **a**`, boolean` **computeVectors** `) throws`
  `com.imsl.math.Eigen.DidNotConvergeException`

- **Description**
  Constructs the eigenvalues and (optionally) the eigenvectors of a real square matrix.
- **Parameters**
  * `a` – is the `double` square matrix whose eigensystem is to be constructed
  * `computeVectors` – is true if the eigenvectors are to be computed
- **Throws**
  * `com.imsl.math.Eigen.DidNotConvergeException` – is thrown when the algorithm fails to converge on the eigenvalues of the matrix.

## Methods

---

- *getValues*
  `public Complex[]` **getValues**`( )`
  - **Description**
    Returns the eigenvalues of a matrix of type `double`.
  - **Returns** – a `Complex` array containing the eigenvalues of this matrix in descending order

---

- *getVectors*
  `public Complex[][]` **getVectors**`( )`
  - **Description**
    Returns the eigenvectors.
  - **Returns** – A `Complex` matrix containing the eigenvectors. The eigenvector corresponding to the j-th eigenvalue is stored in the j-th column. Each vector is normalized to have Euclidean length one.

---

- *performanceIndex*
  `public double` **performanceIndex**`( double[][]` **a** `)`
  - **Description**
    Returns the performance index of a real eigensystem.
  - **Parameters**
    * `a` – a `double` matrix
  - **Returns** – A `double` scalar value indicating how well the algorithms which have computed the eigenvalue and eigenvector pairs have performed. A performance index less than 1 is considered excellent, 1 to 100 is good, while greater than 100 is considered poor.

### Example: Eigensystem Analysis

The eigenvalues and eigenvectors of a matrix are computed.

```
import com.imsl.math.*;

public class EigenEx1 {
    public static void main(String args[]) throws
    Eigen.DidNotConvergeException {
        double a[][] = {
            { 8, -1, -5},
            {-4,  4, -2},
            {18, -5, -7}
        };
        Eigen eigen = new Eigen(a);
        new PrintMatrix("Eigenvalues").print(eigen.getValues());
        new PrintMatrix("Eigenvectors").print(eigen.getVectors());
    }
}
```

### Output

```
Eigenvalues
    0
0  2+4i
1  2-4i
2  1

             Eigenvectors
        0             1          2
0  0.316-0.316i  0.316+0.316i  0.408
1  0.632         0.632         0.816
2      0-0.632i      0+0.632i  0.408
```

## *class* SymEigen

Computes the eigenvalues and eigenvectors of a real symmetric matrix. Orthogonal similarity transformations are used to reduce the matrix to an equivalent symmetric tridiagonal matrix. These transformations are accumulated. An implicit rational QR algorithm is used to compute the eigenvalues of this tridiagonal matrix. The eigenvectors are computed using the eigenvalues as perfect shifts, Parlett (1980, pages 169, 172). The reduction routine is based on the EISPACK routine `TRED2`. See Smith et al. (1976) for the EISPACK routines. Further details, some timing data, and credits are given in Hanson et al. (1990).

Let $M$ = the number of eigenvalues, $\lambda$ = the array of eigenvalues, and $x_j$ is the associated eigenvector with jth eigenvalue.

Also, let $\varepsilon$ be the machine precision. The performance index, $\tau$, is defined to be

$$\tau = \max_{1 \le j \le M} \frac{\|Ax_j - \lambda_j x_j\|_1}{10N\varepsilon \|A\|_1 \|x_j\|_1}$$

While the exact value of $\tau$ is highly machine dependent, the performance of `SymEigen` is considered excellent if $\tau < 1$, good if $1 \le 100$, and poor if $\tau > 100$. The performance index was first developed by the EISPACK project at Argonne National Laboratory; see Smith et al. (1976, pages 124-125).

## Declaration

public class com.imsl.math.SymEigen
**extends** java.lang.Object

## Constructors

---

- *SymEigen*
  `public` **SymEigen**`( double[][] a )`
  - **Description**
    Constructs the eigenvalues and the eigenvectors for a real symmetric matrix.
  - **Parameters**
    * `a` – is the symmetric matrix whose eigensystem is to be constructed.

- *SymEigen*
  `public` **SymEigen**`( double[][] a, boolean` **computeVectors** `)`
  - **Description**
    Constructs the eigenvalues and (optionally) the eigenvectors for a real symmetric matrix.

– **Parameters**
  * `a` – a `double` symmetric matrix whose eigensystem is to be constructed
  * `computeVectors` – a `boolean`, true if the eigenvectors are to be computed
– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown when the lengths of the rows of the input matrix are not uniform.

## Methods

---

• *getValues*
`public double[]` **getValues**`( )`

– **Description**
Returns the eigenvalues
– **Returns** – a `double` array containing the eigenvalues in descending order. If the algorithm fails to converge on an eigenvalue, that eigenvalue is set to NaN.

---

• *getVectors*
`public double[][]` **getVectors**`( )`

– **Description**
Return the eigenvectors of a symmetric matrix of type `double`.
– **Returns** – a `double` array containing the eigenvectors. The j-th column of the eigenvector matrix corresponds to the j-th eigenvalue. The eigenvectors are normalized to have Euclidean length one. If the eigenvectors were not computed by the constructor, then null is returned.

---

• *performanceIndex*
`public double` **performanceIndex**`( double[][]` **a** `)`

– **Description**
Returns the performance index of a real symmetric eigensystem.
– **Parameters**
  * `a` – a `double` symmetric matrix
– **Returns** – a `double` scalar value indicating how well the algorithms which have computed the eigenvalue and eigenvector pairs have performed. A performance index less than 1 is considered excellent, 1 to 100 is good, while greater than 100 is considered poor.
– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown when the lengths of the rows of the input matrix are not uniform.

### Example: Eigenvalues and Eigenvectors of a Symmetric Matrix

The eigenvalues and eigenvectors of a symmetric matrix are computed.

```
import com.imsl.math.*;

public class SymEigenEx1 {
    public static void main(String args[]) {
        double a[][] = {
            {1, 1, 1},
            {1, 1, 1},
            {1, 1, 1}
        };

        SymEigen eigen = new SymEigen(a);
        new PrintMatrix("Eigenvalues").print(eigen.getValues());
        new PrintMatrix("Eigenvectors").print(eigen.getVectors());
    }
}
```

### Output

```
Eigenvalues
   0
0   3
1  -0
2  -0

       Eigenvectors
     0      1       2
0  0.577   0.816   0
1  0.577  -0.408  -0.707
2  0.577  -0.408   0.707
```

# Chapter 3

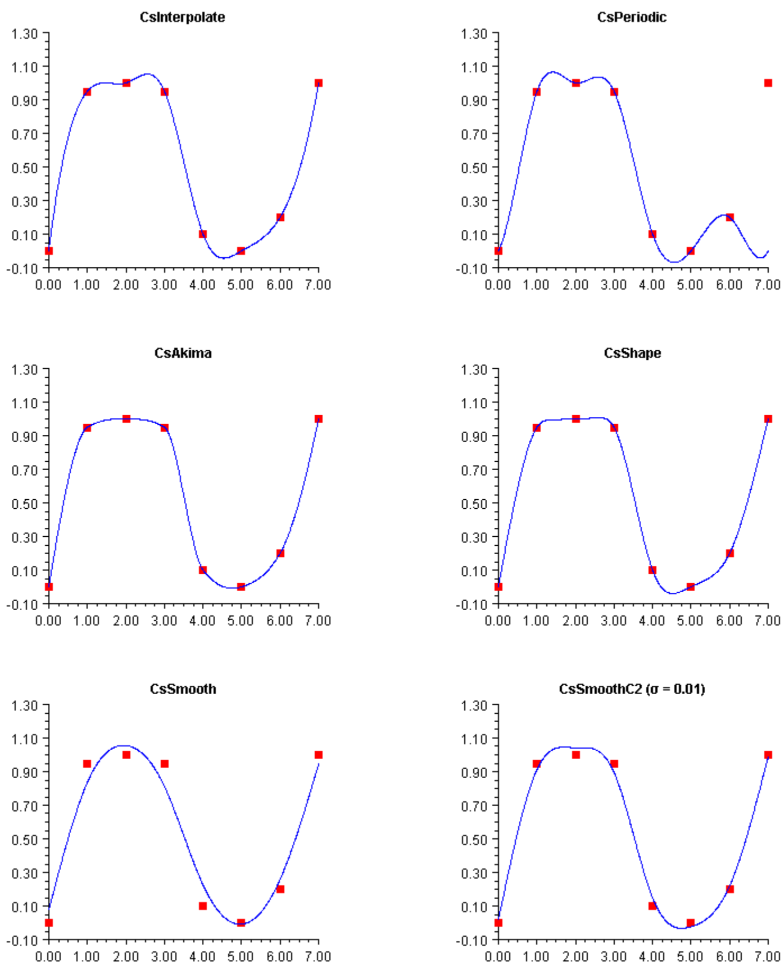
# Interpolation and Approximation

Classes

This chapter contains classes to interpolate and approximate data with cubic splines. Interpolation means that the fitted curve passes through all of the specified data points. An approximation spline does not have to pass through any of the data points. An appoximating curve can therefore be smoother than an interpolating curve.

Cubic splines are smooth $C^1$ or $C^2$ fourth-order piecewise-polynomial (pp) functions. For historical and other reasons, cubic splines are the most heavily used pp functions.

This chapter contains four cubic spline interpolation classes and two approximation classes. These classes are dervived from the base class `Spline`, which provides basic services, such as spline evaluation and integration.‘




The chart shows how the six cubic splines in this chapter fit a single data set.

Class CsInterpolate allows the user to specify various endpoint conditions (such as the value of the first and second derviatives at the right and left endpoints).

Class CsPeriodic is used to fit periodic (repeating) data. The sample data set used is not periodic and so the curve does not pass through the final data point.

Class CsAkima keeps the shape of the data while minimizing oscillations.

Class CsShape keeps the shape of the data by preserving its convexity.

Class CsSmooth constructs a smooth spline from noisy data.

Class CsSmoothC2 constructs a smooth spline from noisy data using cross-validation and a user-supplied smoothing parameter.

## *class* Spline

Spline represents and evaluates univariate piecewise polynomial splines.

A univariate piecewise polynomial (function) *p(x)* is specified by giving its breakpoint sequence $\xi \in \mathbf{R}^n$, the order $k$ (degree $k$-1) of its polynomial pieces, and the $k \times (n-1)$ matrix $c$ of its local polynomial coefficients. In terms of this information, the piecewise polynomial (ppoly) function is given by

$$p(x) = \sum_{j=1}^{k} c_{ji} \frac{(x-\xi_i)^{j-1}}{(j-1)!} \text{ for } \xi_i \le x \le \xi_{i+1}$$

The breakpoint sequence $\xi$ is assumed to be strictly increasing, and we extend the ppoly function to the entire real axis by extrapolation from the first and last intervals.

### Declaration

public abstract class com.imsl.math.Spline
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Fields

- protected `double[][]` **coef**
  - Coefficients of the piecewise polynomials. This is an $n$ by $k$ array, where $n$ is the number of piecewise polynomials and $k$ is the order (degree+1) of the piecewise polynomials.

`coef[i]` contains the coefficients for the piecewise polynomial valid in the interval [`x[k]`,`x[k+1]`).

- protected `double[]` **breakPoint**
  - The breakpoint array of length $n$, where $n$ is the number of piecewise polynomials.
- protected static final `double` **EPSILON_LARGE**
  - The largest relative spacing for double.

## Constructor

---

- *Spline*
  `public` **Spline**`( )`

## Methods

---

- *copyAndSortData*
  `protected void` **copyAndSortData**`( double[]` **xData**, `double[]` **yData** `)`
  - **Description**
    Copy and sort xData into breakPoint and yData into the first column of coef.

---

- *copyAndSortData*
  `protected void` **copyAndSortData**`( double[]` **xData**, `double[]` **yData**, `double[]` **weight** `)`
  - **Description**
    Copy and sort xData into breakPoint and yData into the first column of coef.

---

- *derivative*
  `public double` **derivative**`( double` **x** `)`
  - **Description**
    Returns the value of the first derivative of the spline at a point.
  - **Parameters**
    * `x` – a `double`, the point at which the derivative is to be evaluated
  - **Returns** – a `double` containing the value of the first derivative of the spline at the point x

- *derivative*
  `public double[] derivative( double[] x, int ideriv )`
  - **Description**
    Returns the value of the derivative of the spline at each point of an array.
  - **Parameters**
    * `x` – a `double` array of points at which the derivative is to be evaluated
    * `ideriv` – an `int` specifying the derivative to be computed. If zero, the function value is returned. If one, the first derivative is returned, etc.
  - **Returns** – a `double` array containing the value of the derivative of the spline at each point of the array x

---

- *derivative*
  `public double derivative( double x, int ideriv )`
  - **Description**
    Returns the value of the derivative of the spline at a point.
  - **Parameters**
    * `x` – a `double`, the point at which the derivative is to be evaluated
    * `ideriv` – an `int` specifying the derivative to be computed. If zero, the function value is returned. If one, the first derivative is returned, etc.
  - **Returns** – a `double` containing the value of the derivative of the spline at the point x

---

- *getBreakpoints*
  `public double[] getBreakpoints( )`
  - **Description**
    Returns a copy of the breakpoints.
  - **Returns** – a `double` array containing a copy of the breakpoints

---

- *integral*
  `public double integral( double a, double b )`
  - **Description**
    Returns the value of an integral of the spline.
  - **Parameters**
    * `a` – a `double` specifying the lower limit of integration
    * `b` – a `double` specifying the upper limit of integration
  - **Returns** – a `double`, the integral of the spline from a to b

---

- *value*
  `public double value( double x )`

– **Description**
Returns the value of the spline at a point.

– **Parameters**
  * `x` – a `double`, the point at which the spline is to be evaluated

– **Returns** – a `double` giving the value of the spline at the point x

---

• *value*
`public double[] value( double[] x )`

– **Description**
Returns the value of the spline at each point of an array.

– **Parameters**
  * `x` – a `double` array of points at which the spline is to be evaluated

– **Returns** – a `double` array containing the value of the spline at each point of the array x

## *class* CsAkima

Extension of the Spline class to handle the Akima cubic spline.

Class `CsAkima` computes a $C^1$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, \ldots, n-1$. The breakpoints of the spline are the abscissas. Endpoint conditions are automatically determined by the program; see Akima (1970) or de Boor (1978).

If the data points arise from the values of a smooth, say $C^4$, function $f$, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$\|f - s\|_{[\xi_0, \xi_{n-1}]} \leq C \left\| f^{(2)} \right\|_{[\xi_0, \xi_{n-1}} |\xi|^2$$

where

$$|\xi| \; := \max_{i=1,\ldots,n-1} |\xi_i - \xi_{i-1}|$$

`CsAkima` is based on a method by Akima (1970) to combat wiggles in the interpolant. The method is nonlinear; and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.)

### Declaration

public class com.imsl.math.CsAkima
**extends** com.imsl.math.Spline (page 51)

### Constructor

---

- *CsAkima*
  `public` **CsAkima**`( double[]` **xData**`, double[]` **yData** `)`
  - **Description**
    Constructs the Akima cubic spline interpolant to the given data points.
  - **Parameters**
    * `xData` – a `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – a `double` array containing the y-coordinates of the data.
  - **Throws**
    * `java.lang.IllegalArgumentException` – This exception is thrown if the arrays xData and yData do not have the same length.

### Example: The Akima cubic spline interpolant

A cubic spline interpolant to a function is computed. The value of the spline at point 0.25 is printed.

```
import com.imsl.math.*;

public class CsAkimaEx1 {
    public static void main(String args[]) {
        int n = 11;
        double x[] = new double[n];
        double y[] = new double[n];

        for (int k = 0;  k < n;  k++) {
            x[k] = (double)k/(double)(n-1);
            y[k] = Math.sin(15.0*x[k]);
        }

        CsAkima cs = new CsAkima(x, y);
        double csv =  cs.value(0.25);
        System.out.println("The computed cubic spline value at point .25 is "
```

```
        + csv);
    }
}
```

### Output

```
The computed cubic spline value at point .25 is -0.478185519991867
```

## *class* CsInterpolate

Extension of the Spline class to interpolate data points.

`CsInterpolate` computes a $C^2$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, \ldots, n-1$. The breakpoints of the spline are the abscissas. Endpoint conditions can be automatically determined by the program, or explicitly specified by using the appropriate constructor. Constructors are provided that allow setting specific values for first or second derivative values at the endpoints, or for specifying conditions that correspond to the "not-a-knot" condition (see de Boor 1978).

The "not-a-knot" conditions require that the third derivative of the spline be continuous at the second and next-to-last breakpoint. If $n$ is 2 or 3, then the linear or quadratic interpolating polynomial is computed, respectively.

If the data points arise from the values of a smooth, say, $C^4$ function f, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$|f - s|_{[\xi_0, \xi_n]} \leq C \left\| f^{(4)} \right\|_{[\xi_0, \xi_n]} |\xi|^4$$

where

$$|\xi| := \max_{i=0,\ldots,n-1} |\xi_{i+1} - \xi_i|$$

For more details, see de Boor (1978, pages 55-56).

### Declaration

public class com.imsl.math.CsInterpolate
**extends** com.imsl.math.Spline (page 51)

## Fields

- public static final int **NOT_A_KNOT**
- public static final int **FIRST_DERIVATIVE**
- public static final int **SECOND_DERIVATIVE**

## Constructors

- *CsInterpolate*
  `public` **CsInterpolate**`( double[]` **xData**`, double[]` **yData** `)`
  - **Description**
    Constructs a cubic spline that interpolates the given data points. The interpolant satisfies the "not-a-knot" condition.
  - **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

- *CsInterpolate*
  `public` **CsInterpolate**`( double[]` **xData**`, double[]` **yData**`, int` **typeLeft**`, double` **valueLeft**`, int` **typeRight**`, double` **valueRight** `)`
  - **Description**
    Constructs a cubic spline that interpolates the given data points with specified derivative endpoint conditions.
  - **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
    * `typeLeft` – An `int` denoting the type of condition at the left endpoint. This can be `NOT_A_KNOT`, `FIRST_DERIVATIVE` or `SECOND_DERIVATIVE`.
    * `valueLeft` – A `double` value at the left endpoint. If `typeLeft` is `NOT_A_KNOT` this is ignored, Otherwise, it is the value of the specified derivative.
    * `typeRight` – An `int` denoting the type of condition at the right endpoint. This can be `NOT_A_KNOT`, `FIRST_DERIVATIVE` or `SECOND_DERIVATIVE`.
    * `valueRight` – A `double` value at the right endpoint.

### Example: The cubic spline interpolant

A cubic spline interpolant to a function is computed. The value of the spline at point 0.25 is printed.

```
import com.imsl.math.*;

public class CsInterpolateEx1 {
    public static void main(String args[]) {
        int n = 11;
        double x[] = new double[n];
        double y[] = new double[n];

        for (int k = 0;  k < n;  k++) {
            x[k] = (double)k/(double)(n-1);
            y[k] = Math.sin(15.0*x[k]);
        }

        CsInterpolate cs = new CsInterpolate(x, y);
        double csv =  cs.value(0.25);
        System.out.println("The computed cubic spline value at point .25 is "
        + csv);
    }
}
```

### Output

```
The computed cubic spline value at point .25 is -0.5487725038121579
```

## *class* CsPeriodic

Extension of the Spline class to interpolate data points with periodic boundary conditions.

Class `CsPeriodic` computes a $C^2$ cubic spline interpolant to a set of data points $(x_i, f_i)$ for $i = 0, \ldots n-1$. The breakpoints of the spline are the abscissas. The program enforces periodic endpoint conditions. This means that the spline $s$ satisfies $s(a) = s(b)$, $s'(a) = s'(b)$, and $s''(a) = s''(b)$, where $a$ is the leftmost abscissa and $b$ is the rightmost abscissa. If the ordinate values corresponding to $a$ and $b$ are not equal, then a warning message is issued. The ordinate value at $b$ is set equal to the ordinate value at $a$ and the interpolant is computed.

If the data points arise from the values of a smooth (say $C^4$) periodic function $f$, i.e. $f_i = f(x_i)$, then the error will behave in a predictable fashion. Let $\xi$ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

$$|f - s|_{[\xi_0,\xi_{n-1}]} \leq C|f^{(4)}|_{[\xi_0,\xi_{n-1}]}|\xi|^4$$

where

$$|\xi| := \max_{i=1,\ldots,n-1} |\xi_i - \xi_{i-1}|$$

For more details, see de Boor (1978, pages 320-322).

### Declaration

public class com.imsl.math.CsPeriodic
**extends** com.imsl.math.Spline (page 51)

### Constructor

---

- *CsPeriodic*
  `public` **`CsPeriodic`**`( double[]` **`xData`**`, double[]` **`yData`** `)`
  - **Description**
    Constructs a cubic spline that interpolates the given data points with periodic boundary conditions.
  - **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. There must be at least 4 data points and values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

### Example: The cubic spline interpolant with periodic boundary conditions

A cubic spline interpolant to a function is computed. The value of the spline at point 0.23 is printed.

```
import com.imsl.math.*;

public class CsPeriodicEx1 {
    public static void main(String args[]) {
        int n = 11;
```

```
        double x[] = new double[n];
        double y[] = new double[n];

        double h = 2.*Math.PI/15./10.;
        for (int k = 0;  k < n;  k++) {
            x[k] = h * (double)(k);
            y[k] = Math.sin(15.0*x[k]);
        }

        CsPeriodic cs = new CsPeriodic(x, y);
        double csv =  cs.value(0.23);
        System.out.println("The computed cubic spline value at point .23 is "
        + csv);
    }
}
```

### Output

```
The computed cubic spline value at point .23 is -0.3034014726064514
```

## *class* CsShape

Extension of the Spline class to interpolate data points consistent with the concavity of the data.

Class `CsShape` computes a cubic spline interpolant to $n$ data points $x_i, f_i$ for $i = 0, \ldots, n-1$. For ease of explanation, we will assume that $x_i < x_{i+1}$, although it is not necessary for the user to sort these data values. If the data are strictly convex, then the computed spline is convex, $C^2$, and minimizes the expression

$$\int_{x_1}^{x_n} (g'')^2$$

over all convex $C^1$ functions that interpolate the data. In the general case when the data have both convex and concave regions, the convexity of the spline is consistent with the data and the above integral is minimized under the appropriate constraints. For more information on this interpolation scheme, we refer the reader to Micchelli et al. (1985) and Irvine et al. (1986).

One important feature of the splines produced by this class is that it is not possible, a priori, to predict the number of breakpoints of the resulting interpolant. In most cases, there will be breakpoints at places other than data locations. The method is nonlinear; and although the interpolant is a piecewise cubic, cubic polynomials are not reproduced. (However, linear polynomials are reproduced.) This routine should be used when it is important to preserve the convex and concave regions implied by the data.

## Declaration

public class com.imsl.math.CsShape
**extends** com.imsl.math.Spline (page 51)

## Inner Class

### *class* **CsShape.TooManyIterationsException**

Too many iterations.

#### Declaration

public static class com.imsl.math.CsShape.TooManyIterationsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *CsShape.TooManyIterationsException*
  `public` **CsShape.TooManyIterationsException**`( )`
- *CsShape.TooManyIterationsException*
  `public` **CsShape.TooManyIterationsException**`( java.lang.Object[]` **arguments** `)`
- *CsShape.TooManyIterationsException*
  `public` **CsShape.TooManyIterationsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## Constructor

- *CsShape*
  `public` **CsShape**`(` `double[]` **xData**`,` `double[]` **yData** `)` `throws`
  `com.imsl.math.CsShape.TooManyIterationsException,`
  `com.imsl.math.SingularMatrixException`

  – **Description**
    Construct a cubic spline interpolant which is consistent with the concavity of the data.
  – **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

### Example: The shape preserving cubic spline interpolant

A cubic spline interpolant to a function is computed consistent with the concavity of the data. The spline value at 0.05 is printed.

```
import com.imsl.math.*;

public class CsShapeEx1 {
    public static void main(String args[]) throws com.imsl.IMSLException {
        double x[] = {0.00, 0.10, 0.20, 0.30, 0.40, 0.50, 0.60, 0.80, 1.00};
        double y[] = {0.00, 0.90, 0.95, 0.90, 0.10, 0.05, 0.05, 0.20, 1.00};

        CsShape cs = new CsShape(x, y);
        double csv =  cs.value(0.05);
        System.out.println("The computed cubic spline value at point .05 is "
        + csv);
    }
}
```

### Output

```
The computed cubic spline value at point .05 is 0.5582312228648201
```

## *class* CsSmooth

Extension of the Spline class to construct a smooth cubic spline from noisy data points.

Class `CsSmooth` is designed to produce a $C^2$ cubic spline approximation to a data set in which the function values are noisy. This spline is called a smoothing spline. It is a natural cubic spline with knots at all the data abscissas $x =$ `xData`, but it does not interpolate the data $(x_i, f_i)$. The smoothing spline $S$ is the unique $C^2$ function that minimizes

$$\int_a^b S''(x)^2\, dx$$

subject to the constraint

$$\sum_{i=0}^{n-1} \left|(S(x_i) - f_i)w_i\right|^2 \le \sigma$$

where $\sigma$ is the smoothing parameter. The reader should consult Reinsch (1967) for more information concerning smoothing splines. `CsSmooth` solves the above problem when the user provides the smoothing parameter $\sigma$. `CsSmoothC2` attempts to find the "optimal" smoothing parameter using the statistical technique known as cross-validation. This means that (in a very rough sense) one chooses the value of $\sigma$ so that the smoothing spline $(S_\sigma)$ best approximates the value of the data at $x_I$, if it is computed using all the data except the $i$-th; this is true for all $i = 0, \ldots, n-1$. For more information on this topic, we refer the reader to Craven and Wahba (1979).

## Declaration

public class com.imsl.math.CsSmooth
**extends** com.imsl.math.Spline (page 51)

## Constructors

- *CsSmooth*
  `public` **CsSmooth**`( double[]` **xData**`, double[]` **yData** `)`
  - **Description**
    Constructs a smooth cubic spline from noisy data using cross-validation to estimate the smoothing parameter. All of the points have equal weights.
  - **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.

* `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

---

- *CsSmooth*
  `public` **CsSmooth**`( double[]` **xData**`, double[]` **yData**`, double[]` **weight** `)`

  – **Description**
  Constructs a smooth cubic spline from noisy data using cross-validation to estimate the smoothing parameter. Weights are supplied by the user.

  – **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
    * `weight` – A `double` array containing the relative weights. This array must have the same length as xData.

### Example: The cubic spline interpolant to noisy data

A cubic spline interpolant to noisy data is computed using cross-validation to estimate the smoothing parameter. The value of the spline at point 0.3010 is printed.

```
import com.imsl.math.*;
import com.imsl.stat.*;

public class CsSmoothEx1 {
    public static void main(String args[]) {
        int n = 300;
        double x[]= new double[n];
        double y[]= new double[n];
        for (int k = 0;  k < n;  k++) {
            x[k] = (3.0*k)/(n-1);
            y[k] = 1.0/(0.1 + Math.pow(3.0*(x[k]-1.0),4));
        }

        // Seed the random number generator
        Random rn = new Random();
        rn.setSeed(1234579L);
        rn.setMultiplier(16807);

        // Contaminate the data
        for (int i = 0; i < n; i++) {
```

```
            y[i] += 2.0 * rn.nextFloat() - 1.0;
        }

        // Smooth the data
        CsSmooth cs = new CsSmooth(x, y);
        double csv = cs.value(0.3010);
        System.out.println("The computed cubic spline value at point .3010 is "
        + csv);
    }
}
```

### Output

```
The computed cubic spline value at point .3010 is 0.1078582256142388
```

## *class* CsSmoothC2

Extension of the Spline class used to construct a spline for noisy data points using an alternate method.

Class `CsSmoothC2` is designed to produce a $C^2$ cubic spline approximation to a data set in which the function values are noisy. This spline is called a smoothing spline. It is a natural cubic spline with knots at all the data abscissas $x$, but it does not interpolate the data $(x_i, f_i)$. The smoothing spline $S_\sigma$ is the unique $C^2$ function that minimizes

$$\int_a^b s''_\sigma(x)^2 \, dx$$

subject to the constraint

$$\sum_{i=0}^{n-1} |s_\sigma(x_i) - f_i|^2 \le \sigma$$

.

Recommended values for $\sigma$ depend on the weights, $w$. If an estimate for the standard deviation of the error in the $y$-values is availiable, then $w_i$ should be set to this value and the smoothing parameter should be choosen in the confidence interval corresponding to the left side of the above inequality. That is,

$$n - \sqrt{2n} \le \sigma \le n + \sqrt{2n}$$

`CsSmoothC2` is based on an algorithm of Reinsch (1967). This algorithm is also discussed in de Boor (1978, pages 235-243).

## Declaration

public class com.imsl.math.CsSmoothC2
**extends** com.imsl.math.Spline (page 51)

## Constructors

---

- *CsSmoothC2*
  `public` **CsSmoothC2**`(` `double[]` **xData**, `double[]` **yData**, `double` **sigma** `)`
  - **Description**
    Constructs a smooth cubic spline from noisy data using an algorithm based on Reinsch (1967). All of the points have equal weights.
  - **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
    * `sigma` – A `double` value specifying the smoothing parameter. Sigma must not be negative.

- *CsSmoothC2*
  `public` **CsSmoothC2**`(` `double[]` **xData**, `double[]` **yData**, `double[]` **weight**, `double` **sigma** `)`
  - **Description**
    Constructs a smooth cubic spline from noisy data using an algorithm based on Reinsch (1967) with weights supplied by the user.
  - **Parameters**
    * `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    * `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
    * `weight` – A `double` array containing the weights. The arrays xData and weight must have the same length.
    * `sigma` – A `double` value specifying the smoothing parameter. Sigma must not be negative.

### Example: The cubic spline interpolant to noisy data with supplied weights

A cubic spline interpolant to noisy data is computed using supplied weights and smoothing parameter. The value of the spline at point 0.3010 is printed.

```
import com.imsl.math.*;
import com.imsl.stat.*;

public class CsSmoothC2Ex1 {
    public static void main(String args[]) {
        // Set up a grid
        int n = 300;
        double x[]= new double[n];
        double y[]= new double[n];
        for (int k = 0;  k < n;  k++) {
            x[k] = 3. * ((double)(k)/(double)(n-1));
            y[k] = 1./(.1 + Math.pow(3.*(x[k]-1.),4));
        }

        // Seed the random number generator
        Random rn = new Random();
        rn.setSeed(1234579);
        rn.setMultiplier(16807);

        // Contaminate the data
        for (int i = 0; i < n; i++) {
            y[i] = y[i] + 2. * rn.nextFloat() - 1.;
        }

        // Set the weights
        double sdev = 1./Math.sqrt(3.);
        double weights[]= new double[n];
        for (int i = 0; i < n; i++) {
            weights[i] = sdev;
        }

        // Set the smoothing parameter
        double smpar = (double)n;

        // Smooth the data
        CsSmoothC2 cs = new CsSmoothC2(x, y, weights, smpar);
        double csv =  cs.value(0.3010);
```

```
        System.out.println("The computed cubic spline value at point .3010 is "
        + csv);
    }
}
```

### Output

```
The computed cubic spline value at point .3010 is 0.06458434076781128
```

## *class* BsInterpolate

Extension of the BSpline class to interpolate data points.

Given the data points $x =$ `xData`, $f =$ `yData`, and $n$ the number of elements in `xData` and `yData`, the default action of `BsInterpolate` computes a cubic (order = 4) spline interpolant $s$ to the data using a default "not-a-knot" knot sequence. Constructors are also provided that allow the order and knot sequence to be specified. This algorithm is based on the routine `SPLINT` by de Boor (1978, p. 204).

First, the `xData` vector is sorted and the result is stored in $x$. The elements of `yData` are permuted appropriately and stored in $f$, yielding the equivalent data $(x_i, f_i)$ for $i = 0$ to *n-1*. The following preliminary checks are performed on the data, with $k =$ `order`. We verify that

$x_i < x_{i+1}$ for $i = 0, \ldots, n-2$

$\mathbf{t}_i < \mathbf{t}_{i+k}$ for $i = 0, \ldots, n-1$

$\mathbf{t}_i < \mathbf{t}_{i+1}$ for $i = 0, \ldots, n+k-2$

The first test checks to see that the abscissas are distinct. The second and third inequalities verify that a valid knot sequence has been specified.

In order for the interpolation matrix to be nonsingular, we also check $\mathbf{t}_{k-1} \leq x_i \leq \mathbf{t}_n$ for $i$ = 0 to *n-1*. This first inequality in the last check is necessary since the method used to generate the entries of the interpolation matrix requires that the $k$ possibly nonzero B-splines at $x_i$, $B_{j-k+1}, ..., B_j$ where $j$ satisfies $\mathbf{t}_j \leq x_i < \mathbf{t}_{j+1}$ be well-defined (that is, $j - k + 1 \geq 0$).

### Declaration

public class com.imsl.math.BsInterpolate
**extends** com.imsl.math.BSpline

## Constructors

---

- *BsInterpolate*
`public` **BsInterpolate**`( double[]` **xData**`, double[]` **yData** `)`
  - **Description**
  Constructs a B-spline that interpolates the given data points. The computed B-spline will be order 4 (cubic) and have a default "not-a-knot" spline knot sequence.
  - **Parameters**
    - `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    - `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.

---

- *BsInterpolate*
`public` **BsInterpolate**`( double[]` **xData**`, double[]` **yData**`, int` **order** `)`
  - **Description**
  Constructs a B-spline that interpolates the given data points and order, using a default "not-a-knot" spline knot sequence.
  - **Parameters**
    - `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    - `yData` – A `double` array containing the y-coordinates of the data.The arrays xData and yData must have the same length.
    - `order` – An `int` denoting the order of the B-spline.

---

- *BsInterpolate*
`public` **BsInterpolate**`( double[]` **xData**`, double[]` **yData**`, int` **order**`, double[]` **knot** `)`
  - **Description**
  Constructs a B-spline that interpolates the given data points, using the specified order and knots.
  - **Parameters**
    - `xData` – A `double` array containing the x-coordinates of the data. Values must be distinct.
    - `yData` – A `double` array containing the y-coordinates of the data.The arrays xData and yData must have the same length.
    - `order` – An `int` denoting the order of the spline.
    - `knot` – A `double` array containing the knot sequence for the B-spline.

### Example: The B-spline interpolant

A B-Spline interpolant to data is computed. The value of the spline at point .23 is printed.

```
import com.imsl.math.*;

public class BsInterpolateEx1 {
    public static void main(String args[]) {
        int n = 11;
        double x[] = new double[n];
        double y[] = new double[n];

        double h = 2.*Math.PI/15./10.;
        for (int k = 0;  k < n;  k++) {
            x[k] = h * (double)(k);
            y[k] = Math.sin(15.0*x[k]);
        }

        BsInterpolate bs = new BsInterpolate(x, y);
        double bsv =  bs.value(0.23);
        System.out.println("The computed B-spline value at point .23 is "
        + bsv);
    }
}
```

### Output

```
The computed B-spline value at point .23 is -0.3034183992767692
```

## *class* BsLeastSquares

Extension of the BSpline class to compute a least squares spline approximation to data points.

Let's make the identifications

$n$ = xData.length

$x$ = xData

$f$ = yData

$m$ = nCoef

$k$ = order

For convenience, we assume that the sequence $x$ is increasing, although the class does not require this.

By default, $k = 4$, and the knot sequence we select equally distributes the knots through the distinct $x_i's$. In particular, the $m + k$ knots will be generated in $[x_1, x_n]$ with $k$ knots stacked at each of the extreme values. The interior knots will be equally spaced in the interval.

Once knots $\mathbf{t}$ and weights $w$ are determined, then the spline least-squares fit to the data is computed by minimizing over the linear coefficients $a_j$

$$\sum_{i=0}^{n-1} w_i \left[ f_i - \sum_{j=1}^{m} a_j B_j(x_i) \right]^2$$

where the $B_j, j = 1, ..., m$ are a (B-spline) basis for the spline subspace.

This algorithm is based on the routine L2APPR by deBoor (1978, p. 255).

## Declaration

public class com.imsl.math.BsLeastSquares
**extends** com.imsl.math.BSpline

## Fields

---

- protected `int` **nCoef**
  - Number of B-spline coefficients.
- protected `double[]` **weight**
  - The weight array of length $n$, where $n$ is the number of data points fit.

## Constructors

---

- *BsLeastSquares*
  `public` **BsLeastSquares**`(` `double[]` **xData**`,` `double[]` **yData**`,` `int` **nCoef** `)`

– **Description**
Constructs a least squares B-spline approximation to the given data points.

– **Parameters**

* `xData` – A `double` array containing the x-coordinates of the data.
* `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
* `nCoef` – An `int` denoting the linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline (whose default value is 4).

---

• *BsLeastSquares*
`public` **BsLeastSquares**`( double[]` **xData**`, double[]` **yData**`, int` **nCoef**`,`
`int` **order** `)`

– **Description**
Constructs a least squares B-spline approximation to the given data points.

– **Parameters**

* `xData` – A `double` array containing the x-coordinates of the data.
* `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
* `nCoef` – An `int` denoting the linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline.
* `order` – An `int` denoting the order of the spline.

---

• *BsLeastSquares*
`public` **BsLeastSquares**`( double[]` **xData**`, double[]` **yData**`, int` **nCoef**`,`
`int` **order**`, double[]` **weight**`, double[]` **knot** `)`

– **Description**
Constructs a least squares B-spline approximation to the given data points.

– **Parameters**

* `xData` – A `double` array containing the x-coordinates of the data.
* `yData` – A `double` array containing the y-coordinates of the data. The arrays xData and yData must have the same length.
* `nCoef` – An `int` denoting the linear dimension of the spline subspace. It should be smaller than the number of data points and greater than or equal to the order of the spline.
* `order` – An `int` denoting the order of the spline.
* `weight` – A `double` array containing the weights for the data. The arrays xData, yData and weights must have the same length.
* `knot` – A `double` array containing the knot sequence for the spline.

### Example: The B-spline least squares fit

A B-Spline least squares fit to data is computed. The value of the spline at point 4.5 is printed.

```
import com.imsl.math.*;

public class BsLeastSquaresEx1 {
    public static void main(String args[]) {
        int n = 11;
        double x[] = {0, 1, 2, 3, 4, 5, 8, 9, 10};
        double y[] = {1.0, 0.8, 2.4, 3.1, 4.5, 5.8, 6.2, 4.9, 3.7};

        BsLeastSquares bs = new BsLeastSquares(x, y, 5);
        double bsv =  bs.value(4.5);
        System.out.println("The computed B-spline value at point 4.5 is "
        + bsv);
    }
}
```

### Output

```
The computed B-spline value at point 4.5 is 5.228554323596942
```

## *class* RadialBasis

RadialBasis computes a least-squares fit to scattered data in $\mathbf{R}^d$, where $d$ is the dimension. More precisely, we are given data points

$$x_0, \ldots, x_{n-1} \in \mathbf{R}^d$$

and function values

$$f_0, \ldots, f_{n-1} \in \mathbf{R}^1$$

The radial basis fit to the data is a function $F$ which approximates the above data in the sense that it minimizes the sum-of-squares error

$$\sum_{i=0}^{n-1} w_i \left(F(x_i) - f_i\right)^2$$

where $w$ are the weights. Of course, we must restrict the functional form of $F$. Here we assume it is a linear combination of radial functions:

$$F(x) \equiv \sum_{j=0}^{m-1} \alpha_j \phi(\|x - c_j\|)$$

The $c_j$ are the *centers*.

A radial function, $\phi(r)$, maps $[0, \infty)$ into $\mathbf{R}^1$. The default radial function is the Hardy multiquadric,

$$\phi(r) \equiv \sqrt{r^2 + \delta^2}$$

with $\delta = 1$. An alternate radial function is the Gaussian, $e^{-ax^2}$.

By default, the centers are points in a Faure sequence, scaled to cover the box containing the data.

## Declaration

public class com.imsl.math.RadialBasis
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *interface* RadialBasis.Function

Public interface for the user supplied function to the `RadialBasis` object.

#### Declaration

public static interface com.imsl.math.RadialBasis.Function

#### Methods

---

- *f*
  `double f( double x )`
  - **Description**
    A radial basis function.

– **Parameters**

  * `x` – a `double`, the point at which the function is to be evaluated

– **Returns** – a `double`, the value of the function at x

---

• *g*

`double g( double x )`

– **Description**

  The derivative of the radial basis function.

– **Parameters**

  * `x` – a `double`, the point at which the function is to be evaluated

– **Returns** – a `double`, the value of the function at x

## *class* RadialBasis.HardyMultiquadric

The Hardy multiquadric basis function, $\sqrt{r^2 + \delta^2}$.

### Declaration

public static class com.imsl.math.RadialBasis.HardyMultiquadric
**extends** java.lang.Object
**implements** RadialBasis.Function

### Constructor

---

• *RadialBasis.HardyMultiquadric*

`public` **RadialBasis.HardyMultiquadric**`( double` **delta** `)`

– **Description**

  Creates a Hardy multiquadric basis function.

– **Parameters**

  * `delta` – is the parameter in the function definition.

### Methods

---

• *f*

`double f( double x )`

- **Description copied from RadialBasis.Function** (**page 74**)
  A radial basis function.
- **Parameters**
  * `x` – a `double`, the point at which the function is to be evaluated
- **Returns** – a `double`, the value of the function at x

---

• *g*
`double g( double x )`

- **Description copied from RadialBasis.Function** (**page 74**)
  The derivative of the radial basis function.
- **Parameters**
  * `x` – a `double`, the point at which the function is to be evaluated
- **Returns** – a `double`, the value of the function at x

## *class* RadialBasis.Gaussian

The Gaussian basis function, $e^{-ax^2}$.

### Declaration

public static class com.imsl.math.RadialBasis.Gaussian
**extends** java.lang.Object
**implements** RadialBasis.Function

### Constructor

---

• *RadialBasis.Gaussian*
`public` **RadialBasis.Gaussian**`( double a )`

### Methods

---

• *f*
`double f( double x )`

- **Description copied from RadialBasis.Function** (**page 74**)
  A radial basis function.

- **Parameters**
  - `x` – a `double`, the point at which the function is to be evaluated
- **Returns** – a `double`, the value of the function at x

---

- *g*

  `double g( double x )`

  - **Description copied from RadialBasis.Function** (**page 74**)
    The derivative of the radial basis function.
  - **Parameters**
    - `x` – a `double`, the point at which the function is to be evaluated
  - **Returns** – a `double`, the value of the function at x

## Constructor

---

- *RadialBasis*

  `public` **RadialBasis**`( int` **nDim**`, int` **nCenters** `)`

  - **Description**
    Creates a new instance of RadialBasis.
  - **Parameters**
    - `nDim` – is the number of dimensions.
    - `nCenters` – is the number of centers.

## Methods

---

- *getANOVA*

  `public com.imsl.stat.ANOVA` **getANOVA**`( )`

  - **Description**
    Returns the ANOVA statistics from the linear regression.
  - **Returns** – an `ANOVA` table and related statistics

---

- *getRadialFunction*

  `public RadialBasis.Function` **getRadialFunction**`( )`

  - **Description**
    Returns the radial function.
  - **Returns** – the current radial function.

---

- *gradient*
  `public double[] `**`gradient`**`( double[] `**`x`**` )`
  - **Description**
    Returns the gradient of the radial basis approximation at a point.
  - **Parameters**
    * `x` – is a `double` array containing the locations of the data point at which the approximation's gradient is to be computed.
  - **Returns** – a `double` array, of length `nDim` containing the value of the gradient of the radial basis approximation at $x$.

---

- *setRadialFunction*
  `public void `**`setRadialFunction`**`( RadialBasis.Function `**`radialFunction`**` )`
  - **Description**
    Sets the radial function.
  - **Parameters**
    * `radialFunction` – is the radial function.

---

- *update*
  `public void `**`update`**`( double[][] `**`x`**`, double[] `**`f`**` )`
  - **Description**
    Adds a set of data points, all with weight = 1.
  - **Parameters**
    * `x` – is a `double` matrix of size $n$ by $nDim$ containing the locations of the data points for each dimension.
    * `f` – is a `double` array containing the function values at the data points.

---

- *update*
  `public void `**`update`**`( double[][] `**`x`**`, double[] `**`f`**`, double[] `**`w`**` )`
  - **Description**
    Adds a set of data points with user-specified weights.
  - **Parameters**
    * `x` – is a `double` matrix of size $n$ by $nDim$ containing the locations of the data points for each dimension.
    * `f` – is a `double` array containing the function values at the data points.
    * `w` – is a `double` array containing the weights associated with the data points.

---

- *update*
  `public void `**`update`**`( double[] `**`x`**`, double `**`f`**` )`

- **Description**
  Adds a data point with weight = 1.
- **Parameters**
  * `x` – is a `double` array containing the locations of the data point.
  * `f` – is a `double` containing the function value at the data point.

---

- *update*
  `public void` **update**`( double[] ` **x**`, double ` **f**`, double ` **w** `)`
  - **Description**
    Adds a data point with a specified weight.
  - **Parameters**
    * `x` – is a `double` array containing the locations of the data point.
    * `f` – is a `double` containing the function value at the data point.
    * `w` – is a `double` containing the weight of this data point.

---

- *value*
  `public double` **value**`( double[] ` **x** `)`
  - **Description**
    Returns the value of the radial basis approximation at a point.
  - **Parameters**
    * `x` – is a `double` array containing the locations of the data point at which the approximation is to be computed.
  - **Returns** – the value of the radial basis approximation at $x$.

---

- *value*
  `public double[]` **value**`( double[][] ` **x** `)`
  - **Description**
    Returns the value of the radial basis at a point.
  - **Parameters**
    * `x` – a `double[]`, the point at which the radial basis is to be evaluated
  - **Returns** – a `double` giving the value of the radial basis at the point x

### Example: Radial Basis Function Approximation

The function

$$e^{-\|\vec{x}\|^2/d}$$

where $d$ is the dimension, is evaluated at a set of randomly choosen points. Random noise is added to the values and a radial basis approximated to the noisy data is computed. The radial basis fit is then compared to the original function at another set of randomly choosen points. Both the average error and the maximum error are computed and printed.

In this example, the dimension $d$=10. The function is sampled at 200 random points, in the $[-1, 1]^d$ cube, to which what noise in the range [-0.2,0.2] is added. The error is computed at 1000 random points, also from the $[-1, 1]^d$ cube. The compute errors are less than the added noise.

```
import com.imsl.math.*;
import java.util.Random;

public class RadialBasisEx1 {

    public static void main(String args[]) {
        int nDim = 10;

        // Sample, with noise, the function at 100 randomly choosen points
        int nData = 200;
        double xData[][] = new double[nData][nDim];
        double fData[] = new double[nData];
        Random rand = new Random(234567L);
        for (int k = 0;  k < nData;  k++) {
            for (int i = 0;  i < nDim;  i++) {
                xData[k][i] = 2.0*rand.nextDouble() - 1.0;
            }
            // noisy sample
            fData[k] = fcn(xData[k]) + 0.20*(2.0*rand.nextDouble()-1.0);
        }

        // Compute the radial basis approximation using 25 centers
        int nCenters = 25;
        RadialBasis rb = new RadialBasis(nDim, nCenters);
        rb.update(xData, fData);

        // Compute the error at a randomly selected set of points
        int nTest = 1000;
        double maxError = 0.0;
        double aveError = 0.0;
        double x[] = new double[nDim];
        for (int k = 0;  k < nTest;  k++) {
            for (int i = 0;  i < nDim;  i++) {
                x[i] = 2.0*rand.nextDouble() - 1.0;
            }
            double error = Math.abs(fcn(x)-rb.value(x));
```

```
            aveError += error;
            maxError = Math.max(error, maxError);
            double f = fcn(x);
        }
        aveError /= nTest;

        System.out.println("average error is "+aveError);
        System.out.println("maximum error is "+maxError);
    }


    // The function to approximate
    static double fcn(double x[]) {
        double sum = 0.0;
        for (int k = 0;  k < x.length;  k++) {
            sum += x[k]*x[k];
        }
        sum /= x.length;
        return Math.exp(-sum);
    }
}
```

### Output

```
average error is 0.02619296746295321
maximum error is 0.13197595135821727
```

# Chapter 4

# Quadrature

## Classes



## Usage Notes

### Univariate Quadrature

Class Quadrature computes approximations to integrals of the form

$$\int_c^b f(x)dx$$

Quadrature computes an estimated answer $R$. An optional value `ErrorEstimate` $= E$ estimates the error. These numbers are related as follows:

$$\left|\int_a^b f(x)\,dx - R\right| \le E \le \max\left\{\epsilon, \rho\left|\int_a^b f(x)\,dx\right|\right\}$$

One situation that occasionally arises in univariate quadrature concerns the approximation of integrals when only tabular data are given. The functions described above do not directly address this question. However, the standard method for handling this problem is first to interpolate the data, and then to integrate the interpolant. This can be accomplished by using a JMSL spline interpolation class derived from `com.imsl.math.Spline` and the method `com.imsl.Spline.integral` (a,b)

### Multivariate Quadrature

The class HypercubeQuadrature computes an approximation to the integral of a function of $n$ variables over a hyper-rectangle.

$$\int_{a_1}^{b_1} \ldots \int_{a_n}^{b_n} f\,(x_1,\ \ldots,\ x_n) dx_n \ldots dx_1$$

## *class* Quadrature

`Quadrature` is a general-purpose integrator that uses a globally adaptive scheme in order to reduce the absolute error. It subdivides the interval $[A,\ B]$ and uses a $(2k\ +\ 1)$-point Gauss-Kronrod rule to estimate the integral over each subinterval. The error for each subinterval is estimated by comparison with the $k$-point Gauss quadrature rule. The subinterval with the largest estimated error is then bisected and the same procedure is applied to both halves. The bisection process is continued until either the error criterion is satisfied, roundoff error is detected, the subintervals become too small, or the maximum number of subintervals allowed is reached. The Class `Quadrature` is based on the subroutine `QAG` by Piessens et al. (1983).

### Declaration

public class com.imsl.math.Quadrature
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Class

#### *interface* Quadrature.Function

Public interface function for the Quadrature class.

**Declaration**

public static interface com.imsl.math.Quadrature.Function

**Method**

---

- *f*
  `double` **f**`( double` **x** `)`
  - **Description**
    Returns the value of the function at the given point.
  - **Parameters**
    * `x` – a `double` specifying the point at which the function is to be evaluated
  - **Returns** – a `double` specifying the value of the function at x

## Constructor

---

- *Quadrature*
  `public` **Quadrature**`( )`
  - **Description**
    Constructs a Quadrature object.

## Methods

---

- *eval*
  `public synchronized double` **eval**`( Quadrature.Function` **objectF**`, double` **a**`, double` **b** `)`
  - **Description**
    Returns the value of the integral from a to b.
  - **Parameters**
    * `objectF` – an implementation of `Function` containing the function to be integrated
    * `a` – a `double` specifying the lower limit of integration

      * `b` – a `double` specifying the upper limit of integration, either or both of a and b can be Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY

---

- *getErrorEstimate*
  `public double` **getErrorEstimate**`( )`
  - **Description**
    Returns an estimate of the relative error in the computed result.
  - **Returns** – a `double` specifying an estimate of the relative error in the computed result

---

- *getErrorStatus*
  `public int` **getErrorStatus**`( )`
  - **Description**
    Returns the non-fatal error status.
  - **Returns** – an `int` specifying the non-fatal error status:

| Status | Meaning |
|---|---|
| 1 | Maximum number of subdivisions allowed has been achieved. One can allow more subdivisions by using setMaxSubintervals. If this yields no improvement it is advised to analyze the integrand in order to determine the integration difficulties. If the position of a local difficulty can be determined (e.g. singularity, discontinuity within the interval) one will probably gain from splitting up the interval at this point and calling the integrator on the subranges. If possible, an appropriate special-purpose integrator should be used, which is designed for handling the type of difficulty involved. |
| 2 | The occurrence of roundoff error is detected, which prevents the requested tolerance from being achieved. The error may be under-estimated. |
| 3 | Extremely bad integrand behavior occurs at some points of the integration interval. |
| 5 | The algorithm does not converge. Roundoff error is detected in the extrapolation table. It is presumed that the requested tolerance cannot be achieved, and that the returned result is the best that can be obtained. |
| 6 | The integral is probably divergent, or slowly convergent. It must be noted that divergence can occur with any other status value. |

---

- *setAbsoluteError*
  `public synchronized void` **setAbsoluteError**`( double` **errorAbsolute** `)`
  - **Description**
    Sets the absolute error tolerance.
  - **Parameters**

* `errorAbsolute` – a `double` scalar value specifying the absolute error

---

- *setExtrapolation*
  `public synchronized void` **setExtrapolation**`( boolean` **doExtrapolation** `)`
  - **Description**
    If true, the epsilon-algorithm for extrapolation is enabled. The default is false (extrapolation is not used).
  - **Parameters**
    * `doExtrapolation` – a `boolean`, true if the epsilon-algorithm for extrapolation is to be enabled, false otherwise

---

- *setMaxSubintervals*
  `public synchronized void` **setMaxSubintervals**`( int` **maxSubintervals** `)`
  - **Description**
    Sets the maximum number of subintervals allowed. The default value is 500.
  - **Parameters**
    * `maxSubintervals` – an `int` specifying the maximum number of subintervals to be allowed. The default is 500.

---

- *setRelativeError*
  `public synchronized void` **setRelativeError**`( double` **errorRelative** `)`
  - **Description**
    Sets the relative error tolerance.
  - **Parameters**
    * `errorRelative` – a `double` scalar value specifying the relative error

---

- *setRule*
  `public synchronized void` **setRule**`( int` **rule** `)`
  - **Description**
    Set the Gauss-Kronrod rule.

| Rule | Data points used |
|---|---|
| 1 | 7 - 15 |
| 2 | 10 - 21 |
| 3 | 15 - 31 |
| 4 | 20 - 41 |
| 5 | 25 - 51 |
| 6 | 30 - 61 |

    The default is rule 3.
  - **Parameters**
    * `rule` – an `int` specifying the rule to be used. The default is 3.

### Example 1: Integral $\int_1^3 e^{2x}\,dx$

The integral $\int_1^3 e^{2x}\,dx$ is computed and compared to its expected value.

```
import com.imsl.math.*;

public class QuadratureEx1 {
    public static void main(String args[]) {
        Quadrature.Function fcn = new Quadrature.Function() {
            public double f(double x) {
                return Math.exp(2.*x);
            }
        };

        Quadrature q = new Quadrature();
        double result = q.eval(fcn, 1.0, 3.0);

        double expect = (Math.exp(6)-Math.exp(2))/2.;
        System.out.println("result = "+result);
        System.out.println("expect = "+expect);
    }
}
```

### Output

```
result = 198.01986869690225
expect = 198.01986869690222
```

### Example 2: Integral $\int_0^\infty e^{-x}\,dx$

The integral $\int_0^\infty e^{-x}\,dx$ is computed and compared to its expected value.

```
import com.imsl.math.*;

public class QuadratureEx2 {
    public static void main(String args[]) {

        Quadrature.Function fcn = new Quadrature.Function() {
            public double f(double x) {
                return Math.exp(-x);
            }
        };
```

```
        Quadrature q = new Quadrature();
        double result = q.eval(fcn, 0.0, Double.POSITIVE_INFINITY);

        double expect = 1.;
        System.out.println("result = "+result);
        System.out.println("expect = "+expect);
    }
}
```

### Output

```
result = 0.999999999999999
expect = 1.0
```

### Example 3: Integral of the entire real line

The integral $\int_{-\infty}^{\infty} \frac{x}{4e^x+9e^{-x}}\,dx$ is computed and compared to its expected value. This integral is evaluated in Gradshteyn and Ryzhik (equation 3.417.1).

```
import com.imsl.math.*;

public class QuadratureEx3 {
    public static void main(String args[]) {
        Quadrature.Function fcn = new Quadrature.Function() {
            public double f(double x) {
                return x / (4*Math.exp(x)+9*Math.exp(-x));
            }
        };

        Quadrature q = new Quadrature();
        double result = q.eval(fcn, Double.NEGATIVE_INFINITY,
        Double.POSITIVE_INFINITY);

        double expect = Math.PI*Math.log(1.5)/12.;
        System.out.println("result = "+result);
        System.out.println("expect = "+expect);
    }
}
```

### Output

```
result = 0.10615051707662819
expect = 0.10615051707663337
```

### Reference

Gradshteyn, I. S. and I. M. Ryzhik (1965), *Table of Integrals, Series, and Products*, Academic Press, New York.

### Example 4: Integral of an oscillatory function

The integral of $\cos(ax)$ for $a = 10^4$ is computed and compared to its expected value. Because the function is highly oscillatory, the quadrature rule is set to 6. The relative error tolerance is also set.

```
import com.imsl.math.*;

public class QuadratureEx4 {
    public static void main(String args[]) {
        final double a = 1.0e4;

        Quadrature.Function fcn = new Quadrature.Function() {
            public double f(double x) {
                return Math.cos(a*x);
            }
        };

        Quadrature q = new Quadrature();
        q.setRule(6);
        q.setRelativeError(1.e-10);
        double result = q.eval(fcn, 0.0, 1.0);

        double expect = Math.sin(a)/a;
        System.out.println("result = "+result);
        System.out.println("expect = "+expect);
        System.out.println("relative error = "+(expect-result)/expect);
        System.out.println("relative error estimate = "+q.getErrorEstimate());
    }
}
```

### Output

```
result = -3.05614388902526E-5
expect = -3.056143888882521E-5
relative error = -4.670545934003717E-11
relative error estimate = 1.0488375541870691E-8
```

## *class* **HyperRectangleQuadrature**

HyperRectangleQuadrature integrates a function over a hypercube. This class is used to evaluate integrals of the form:

$$\int_{a_{n-1}}^{b_{n-1}} \cdots \int_{a_0}^{b_0} f(x_0, \ldots, x_{n-1})\, dx_0 \ldots dx_{n-1}$$

Integration of functions over hypercubes by Monte Carlo, in which the integral is evaluated as the value of the function averaged over a sequence of randomly chosen points. Under mild assumptions on the function, this method will converge like $1/\sqrt{n}$, where $n$ is the number of points at which the function is evaluated.

It is possible to improve on the performance of Monte Carlo by carefully choosing the points at which the function is to be evaluated. Randomly distributed points tend to be non-uniformly distributed. The alternative to a sequence of random points is a low-discrepancy sequence. A low-discrepancy sequence is one that is highly uniform.

This function is based on the low-discrepancy Faure sequence as computed by com.imsl.stat.FaureSequence.

### Declaration

public class com.imsl.math.HyperRectangleQuadrature
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Class

#### *interface* **HyperRectangleQuadrature.Function**

Public interface function for the HyperRectangleQuadrature class.

### Declaration

public static interface com.imsl.math.HyperRectangleQuadrature.Function

### Method

---

- *f*
  `double f( double[] x )`
  - **Description**
    Returns the value of the function at the given point.
  - **Parameters**
    * `x` – a `double` array specifying the point at which the function is to be evaluated
  - **Returns** – a `double` specifying the value of the function at x

## Constructors

---

- *HyperRectangleQuadrature*
  `public HyperRectangleQuadrature( int dim )`
  - **Description**
    Constructs a HyperRectangleQuadrature object.

- *HyperRectangleQuadrature*
  `public HyperRectangleQuadrature( com.imsl.stat.RandomSequence sequence )`
  - **Description**
    Constructs a HyperRectangleQuadrature object.

## Methods

---

- *eval*
  `public double eval( HyperRectangleQuadrature.Function objectF )`
  - **Description**
    Returns the value of the integral over the unit cube.

– **Parameters**
  * `objectF` – `Function` containing the function to be integrated

---

• *eval*
`public double` **eval**`( HyperRectangleQuadrature.Function` **objectF**`,`
`double[]` **a**`, double[]` **b** `)`

– **Description**
  Returns the value of the integral over a cube.

– **Parameters**
  * `objectF` – `Function` containing the function to be integrated
  * `a` – is a `double` specifying the lower limit of integration. If null all of the lower limits default to 0.
  * `b` – is a `double` specifying the upper limit of integration. If null all of the upper limits default to 1.

---

• *getErrorEstimate*
`public double` **getErrorEstimate**`( )`

– **Description**
  Returns an estimate of the relative error in the computed result.

– **Returns** – a `double` specifying an estimate of the relative error in the computed result

---

• *setAbsoluteError*
`public synchronized void` **setAbsoluteError**`( double` **errorAbsolute** `)`

– **Description**
  Sets the absolute error tolerance.

– **Parameters**
  * `errorAbsolute` – a `double` scalar value specifying the absolute error

---

• *setRelativeError*
`public synchronized void` **setRelativeError**`( double` **errorRelative** `)`

– **Description**
  Sets the relative error tolerance.

– **Parameters**
  * `errorRelative` – a `double` scalar value specifying the relative error

### Example: HyperRectangle Quadrature

This example evaluates the following multidimensional integral, with $n$=10.

$$\int_{a_{n-1}}^{b_{n-1}} \cdots \int_{a_0}^{b_0} \left[ \sum_{i=0}^{n} (-1)^i \prod_{j=0}^{i} x_j \right] dx_0 \ldots dx_{n-1} = \frac{1}{3} \left[ 1 - \left( -\frac{1}{2} \right)^n \right]$$

```
import com.imsl.math.*;

public class HyperRectangleQuadratureEx1 {
    public static void main(String args[]) {

        HyperRectangleQuadrature.Function fcn =
        new HyperRectangleQuadrature.Function() {
            public double f(double x[]) {
                int sign = 1;
                double sum = 0.0;
                for (int i = 0;  i < x.length;  i++) {
                    double prod = 1.0;
                    for (int j = 0;  j <= i;  j++) {
                        prod *= x[j];
                    }
                    sum += sign * prod;
                    sign = -sign;
                }
                return sum;
            }
        };

        HyperRectangleQuadrature q = new HyperRectangleQuadrature(10);
        double result = q.eval(fcn);
        System.out.println("result = "+result);
    }
}
```

### Output

```
result = 0.3331253832089543
```

# Chapter 5

# Differential Equations

## Classes



## Usage Notes

### Ordinary Differential Equations

An *ordinary differential equation* is an equation involving one or more dependent variables called $y_i$, one independent variable, $t$, and derivatives of the $y_i$ with respect to $t$.

In the *initial-value problem* (IVP), the initial or starting values of the dependent variables $y_i$ at a known value $t = t_0$ are given. Values of $y_i(t)$ for $t > 0$ or $t < t_0$ are required.

The `OdeRungeKutta` class solves the IVP for ODEs of the form

$$\frac{dy_i}{dt} = y_i' = f_i\left(t,\, y_1,\, \ldots,\, y_N\right) \qquad i = 1,\, \ldots,\, N$$

with $y_i = (t = t_0)$ specified. Here, $f_i$ is a user-supplied function that must be evaluated at any set of values $(t, y_1, \ldots, y_N), i = 1, \ldots, N$.

This problem statement is abbreviated by writing it as a system of first-order ODEs,

$$y\left(t\right)\left[y_1\left(t\right), \ldots, y_N\left(t\right)\right]^T, \left[f_1\left(t, y\right), \ldots, f_N\left(t, y\right)\right]^T$$

, so that the problem becomes $y' = f\,(t, y)$with initial values $y(t_0)$.

The system

$$\frac{dy}{dt} = y' = f\,(t, y)$$

is said to be *stiff* if some of the eigenvalues of the Jacobian matrix

$$\{\partial y'_i / \partial y_j\}$$

are large and negative. This is frequently the case for differential equations modeling the behavior of physical systems, such as chemical reactions proceeding to equilibrium where subspecies effectively complete their reactions in different epochs. An alternate model concerns discharging capacitors such that different parts of the system have widely varying decay rates (or *time constants*).

Users typically identify stiff systems by the fact that numerical differential equation solvers such as `OdeRungeKutta` are inefficient, or else completely fail. Special methods are often required. The most common inefficiency is that a large number of evaluations of *f(t, y)* (and hence an excessive amount of computer time) are required to satisfy the accuracy and stability requirements of the software.

## *class* OdeRungeKutta

Solves an initial-value problem for ordinary differential equations using the Runge-Kutta-Verner fifth-order and sixth-order method.

Class `OdeRungeKutta` finds an approximation to the solution of a system of first-order differential equations of the form $y_0 = f(t, y)$ with given initial data. The routine attempts to keep the global error proportional to a user-specified tolerance. This routine is efficient for nonstiff systems where the derivative evaluations are not expensive.

`OdeRungeKutta` is based on a code designed by Hull, Enright and Jackson (1976, 1977). It uses Runge-Kutta formulas of order five and six developed by J. H. Verner.

### Declaration

public class com.imsl.math.OdeRungeKutta
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Classes

### *interface* **OdeRungeKutta.Function**

Public interface for user supplied function to `OdeRungeKutta` object.

#### Declaration

public static interface com.imsl.math.OdeRungeKutta.Function

#### Method

---

- *f*
  `void` **f**`( double` **x**`, double[]` **y**`, double[]` **yprime** `)`
  - **Description**
    Returns the value of the function at the given point.
  - **Parameters**
    * `x` – a `double`, the point at which the function is to be evaluated
    * `y` – a `double` array which contains the dependent variable values
    * `yprime` – a `double` array which contains the value of the function at (x,y)

### *class* **OdeRungeKutta.ToleranceTooSmallException**

Tolerance is too small.

#### Declaration

public static class com.imsl.math.OdeRungeKutta.ToleranceTooSmallException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *OdeRungeKutta.ToleranceTooSmallException*
  `public` **OdeRungeKutta.ToleranceTooSmallException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **OdeRungeKutta.DidNotConvergeException**

The iteration did not converge.

#### Declaration

public static class com.imsl.math.OdeRungeKutta.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *OdeRungeKutta.DidNotConvergeException*
  `public` **OdeRungeKutta.DidNotConvergeException**( `java.lang.String` **message** )

- *OdeRungeKutta.DidNotConvergeException*
  `public` **OdeRungeKutta.DidNotConvergeException**( `java.lang.String` **key**, `java.lang.Object[]` **arguments** )

#### Fields

---

- public static final `int` **BEFORE_STEP**
  - Used by method `examineStep` to indicate examining before the next step
- public static final `int` **AFTER_SUCCESSFUL_STEP**
  - Used by method `examineStep` to indicate examining after a successful step
- public static final `int` **AFTER_UNSUCCESSFUL_STEP**
  - Used by method `examineStep` to indicate examining after an unsuccessful step

#### Constructor

---

- *OdeRungeKutta*
  `public` **OdeRungeKutta**( `OdeRungeKutta.Function` **function** )

- **Description**
  Constructs an ODE solver to solve the initial value problem $dy/dx = \mathrm{f(x,y)}$
- **Parameters**
  * `function` – Implementation of interface Function that defines the right-hand side function f(x,y)

## Methods

---

- *examineStep*
  `protected void` **examineStep**`( int` **state**`, double` **x**`, double[]` **y** `)`
  - **Description**
    Called before and after each internal step.
  - **Parameters**
    * `state` – an `int`, one of `BEFORE_STEP`, `AFTER_SUCCESSFUL_STEP` or `AFTER_UNSUCCESSFUL_STEP`.
    * `x` – `double` representing the indepenent variable.
    * `y` – `double` array containing the dependent variables.

---

- *setFloor*
  `public synchronized void` **setFloor**`( double` **floor** `)`
  - **Description**
    Sets the value used in the norm computation.
  - **Parameters**
    * `floor` – `double` used in the norm computation, default value is 1.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `floor` is less than or equal to zero.

---

- *setInitialStepsize*
  `public synchronized void` **setInitialStepsize**`( double` **stepsize** `)`
  - **Description**
    Sets the initial internal step size.
  - **Parameters**
    * `stepsize` – `double` specifying the initial internal step size.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `stepsize` is less than or equal to zero.

---

- *setMaximumStepsize*
  `public synchronized void` **setMaximumStepsize**`( double` **stepsize** `)`

  - **Description**
    Sets the maximum internal step size.
  - **Parameters**
    * `stepsize` – Maximum internal step size. Default value is 2.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `stepsize` is less than or equal to 0.

---

- *setMaxSteps*
  `public synchronized void` **setMaxSteps**`( int` **maxSteps** `)`

  - **Description**
    Sets the maximum number of internal steps allowed.
  - **Parameters**
    * `maxSteps` – `int` specifying the maximum number of internal steps allowed, default value is 500
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `maxSteps` is less than or equal to zero.

---

- *setMinimumStepsize*
  `public synchronized void` **setMinimumStepsize**`( double` **stepsize** `)`

  - **Description**
    Sets the minimum internal step size.
  - **Parameters**
    * `stepsize` – Minimum internal step size. Default value is 0.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `stepsize` is less than or equal to 0.

---

- *setNorm*
  `public synchronized void` **setNorm**`( int` **normMethod** `)`

  - **Description**
    Sets the switch for determining the error norm.
  - **Parameters**
    * `normMethod` – `int` specifying the switch for determining the error norm, default value is 0. In the following, $e_i$ is the absolute value fo an estimate of the error in $y_i(t)$

| norm | Constraint |
|---|---|
| 0 | Minimum of the absolute error and the relative error, equals the maximum of $e_i/max(\|y_i(t)\|, 1)$ |
| 1 | Absolute error, equals $max(e_i)$ |
| 2 | Maximum of $e_i/max(\|y_i(t)\|, floor)$ |

- **Throws**
  - `java.lang.IllegalArgumentException` – is thrown if `norm` is is not 0, 1, or 2.

---

- *setScale*

  `public synchronized void` **setScale**`( double` **scale** `)`

  - **Description**
    Sets the scaling factor.
  - **Parameters**
    - `scale` – `double` specifying the scaling factor, default value is 1.e0
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown if `scale` is less than or equal to 0.

---

- *setTolerance*

  `public synchronized void` **setTolerance**`( double` **tolerance** `)`

  - **Description**
    Sets the error tolerance.
  - **Parameters**
    - `tolerance` – `double` specifying the error tolerance. Default value is 1.0e-6.
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown if `tolerance` less than or equal 0.

---

- *solve*

  `public synchronized void` **solve**`( double` **x**`, double` **xEnd**`, double[]` **y** `)`
  `throws com.imsl.math.OdeRungeKutta.ToleranceTooSmallException,`
  `com.imsl.math.OdeRungeKutta.DidNotConvergeException`

  - **Description**
    Integrates the ODE system from x to xEnd. On all but the first call to solve, the value of x must equal the value of xEnd for the previous call.
  - **Parameters**
    - `x` – `double` specifying the independent variable
    - `xEnd` – `double` specifying the value of x at which the solution is desired

* y – On input, `double` array containing the initial values. On output, `double` array containing the approximate solution.

– **Throws**

  * `com.imsl.math.OdeRungeKutta.DidNotConvergeException` – is thrown if the number of internal steps exceeds maxSteps (default 500). This can be an indication that the ODE system is stiff. This exception can also be thrown if the error tolerance condition could not be met.
  * `com.imsl.math.OdeRungeKutta.ToleranceTooSmallException` – is thrown if the computation does not converge on some step.

---

• *vnorm*

`protected double` **vnorm**`( double[]` **v**`, double[]` **y**`, double[]` **ymax** `)`

– **Description**

Returns the norm of a vector.

– **Parameters**

  * v – `double` array containing the vector whose norm is to be computed
  * y – `double` array containing the values of the dependent variable
  * ymax – `double` array containing the maximum y values computed thus far

– **Returns** – `double` scalar value representing the norm of the vector v

### Example: Runge-Kutta-Verner ordinary differential equation solver

An ordinary differential equation problem is solved using a solver which implements the Runge-Kutta-Verner method. The solution at time t=10 is printed.

```
import com.imsl.math.*;

public class OdeRungeKuttaEx1 {
    public static void main(String args[]) throws com.imsl.IMSLException {
        OdeRungeKutta.Function fcn = new OdeRungeKutta.Function() {
            public void f(double t, double y[], double yprime[]) {
                yprime[0] = 2. * y[0] * (1-y[1]);
                yprime[1] = -y[1] * (1-y[0]);
            }
        };

        double y[] = {1,3};
        OdeRungeKutta q = new OdeRungeKutta(fcn);
        int nsteps = 10;
        for (int k = 0;  k < nsteps;  k++) {
            q.solve(k, k+1, y);
        }
```

```
        System.out.println("Result = {"+y[0]+","+y[1]+"}");
    }
}
```

**Output**

```
Result = {3.1443416765160768,0.3488265985196999}
```

# Chapter 6

# Transforms

**Classes**



## Usage Notes

### Fast Fourier Transforms

A fast Fourier transform (FFT) is simply a discrete Fourier transform that is computed efficiently. Basically, the straightforward method for computing the Fourier transform takes approximately $n^2$ operations where $n$ is the number of points in the transform, while the FFT (which computes the same values) takes approximately $n \log n$ operations. The algorithms in this chapter are modeled on the Cooley-Tukey (1965) algorithm. Hence, these functions are most efficient for integers that are highly composite; that is, integers that are a product of small primes.

For the two classes, FFT and ComplexFFT, a single instance can be used to transform multiple sequences of the same length. In this situation, the constructor computes the initial setup once. This may result in substantial computational savings. For more information on the use of these classes consult the documentation under the appropriate class name.

### Continuous Versus Discrete Fourier Transform

There is, of course, a close connection between the discrete Fourier transform and the continuous Fourier transform. Recall that the continuous Fourier transform is defined (Brigham 1974) as

$$\hat{f}\,(\omega) = (\Im f)\,(\omega) = \int_{-\infty}^{\infty} f\,(t)e^{-2\pi i\omega t}dt$$

We begin by making the following approximation:

$$\hat{f}\,(\omega) \approx \int_{-T/2}^{T/2} f\,(t)e^{-2\pi i\omega t}dt$$

$$= \int_{0}^{T} f\,(t - T/2)e^{-2\pi i\omega(t-T/2)}dt$$

$$= e^{\pi i\omega T}\int_{0}^{T} f\,(t - T/2)e^{-2\pi i\omega t}dt$$

If we approximate the last integral using the rectangle rule with spacing $h = T/n$ , we have

$$\hat{f}\,(\omega) \approx e^{\pi i\omega T}h\sum_{k=0}^{n-1} e^{-2\pi i\omega kh}f\,(kh - T/2)$$

Finally, setting $\omega = j/T$ for $j = 0, \ldots, n-1$ yields

$$\hat{f}\,(j/T) \approx e^{\pi ij}h\sum_{k=0}^{n-1} e^{-2\pi ijk/n}f\,(kh - T/2) = (-1)^j\sum_{k=0}^{n-1} e^{-2\pi ijk/n}f_k^h$$

where the vector $f^h = (f(-T/2), \ldots, f((n-1)h - T/2))$ . Thus, after scaling the components by $(-1)^h$ , the discrete Fourier transform, as computed in `ComplexFFT` (with input $f^h$) is related to an approximation of the continuous Fourier transform by the above formula.

## *class* **FFT**

FFT functions.

Class FFT computes the discrete Fourier transform of a real vector of size $n$. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $n$ is a product of small prime factors. If $n$ satisfies this condition, then the computational effort is proportional to $n$ log $n$.

The `forward` method computes the forward transform. If $n$ is even, then the forward transform is

$$q_{2m-1} = \sum_{k=0}^{n-1} p_k \cos \frac{2\pi km}{n} \quad m = 1,\ \ldots,\ n/2$$

$$q_{2m-2} = -\sum_{k=0}^{n-1} p_k \sin \frac{2\pi km}{n} \quad m = 1,\ \ldots,\ n/2-1$$

$$q_0 = \sum_{k=0}^{n-1} p_k$$

If $n$ is odd, $q_m$ is defined as above for $m$ from *1* to *(n - 1)/2*.

Let $f$ be a real valued function of time. Suppose we sample $f$ at $n$ equally spaced time intervals of length $\delta$ seconds starting at time $t_0$. That is, we have

$$p_i := f\left(t_0 + i\Delta\right)\ i = 0,\ 1, \ldots,\ n-1$$

We will assume that $n$ is odd for the remainder of this discussion. The class FFT treats this sequence as if it were periodic of period $n$. In particular, it assumes that $f\left(t_0\right) = f\left(t_0 + n\Delta\right)$. Hence, the period of the function is assumed to be $T = n\Delta$. We can invert the above transform for $p$ as follows:

$$p_m = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1} \cos \frac{2\pi km}{n} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2} \sin \frac{2\pi km}{n}\right]$$

This formula is very revealing. It can be interpreted in the following manner. The coefficients $q$ produced by FFT determine an interpolating trigonometric polynomial to the data. That is, if we define

$$g\left(t\right) = \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1} \cos \frac{2\pi k\left(t - t_0\right)}{n\Delta} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2} \sin \frac{2\pi k\left(t - t_0\right)}{n\Delta}\right]$$

$$= \frac{1}{n}\left[q_0 + 2\sum_{k=0}^{(n-3)/2} q_{2k+1}\cos\frac{2\pi k\left(t-t_0\right)}{T} - 2\sum_{k=0}^{(n-3)/2} q_{2k+2}\sin\frac{2\pi k\left(t-t_0\right)}{T}\right]$$

then we have

$$f\left(\mathrm{t}_0 + (i-1)\,\Delta\right) = g\left(\mathrm{t}_0 + (i-1)\right)\Delta$$

Now suppose we want to discover the dominant frequencies, forming the vector $P$ of length *(n + 1)/2* as follows:

$$P_0 := |q_0|$$

$$P_k := \sqrt{q_{2k-2}^2 + q_{2k-1}^2} \quad k = 1,\ 2,\ \ldots,\ (n-1)/2$$

These numbers correspond to the energy in the spectrum of the signal. In particular, $P_k$ corresponds to the energy level at frequency

$$\frac{k}{T} = \frac{k}{n\Delta} \qquad k = 0,\ 1,\ \ldots,\ \frac{n-1}{2}$$

Furthermore, note that there are only $(n+1)/2 \approx T/(2\Delta)$ resolvable frequencies when $n$ observations are taken. This is related to the Nyquist phenomenon, which is induced by discrete sampling of a continuous signal. Similar relations hold for the case when $n$ is even.

If the `backward` method is used, then the backward transform is computed. If `n` is even, then the backward transform is

$$q_m = p_0 + (-1)^m\, p_{n-1} + 2\sum_{k=0}^{n/2-1} p_{2k+1}\cos\frac{2\pi km}{n} - 2\sum_{k=0}^{n/2-2} p_{2k+2}\sin\frac{2\pi km}{n}$$

If $n$ is odd,

$$q_m = p_0 + 2\sum_{k=0}^{(n-3)/2} p_{2k+1}\cos\frac{2\pi km}{n} - 2\sum_{k=0}^{(n-3)/2} p_{2k+2}\sin\frac{2\pi km}{n}$$

The backward Fourier transform is the unnormalized inverse of the forward Fourier transform.

`FFT` is based on the real FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Declaration

public class com.imsl.math.FFT
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *FFT*
  `public` **FFT**`( int` **n** `)`
  - **Description**
    Constructs an FFT object.
  - **Parameters**
    * `n` – is the length of the sequence to be transformed

## Methods

- *backward*
  `public double[]` **backward**`( double[]` **coef** `)`
  - **Description**
    Compute the real periodic sequence from its Fourier coefficients.
  - **Parameters**
    * `coef` – a `double` array containing the Fourier coefficients
  - **Returns** – a `double` array containing the periodic sequence

- *forward*
  `public double[]` **forward**`( double[]` **seq** `)`
  - **Description**
    Compute the Fourier coefficients of a real periodic sequence.
  - **Parameters**
    * `seq` – a `double` array containing the sequence to be transformed
  - **Returns** – a `double` array containing the transformed sequence

### Example: Fast Fourier Transform

The Fourier coefficients of a periodic sequence are computed. The coefficients are then used to reproduce the periodic sequence.

```
import com.imsl.math.*;

public class FFTEx1 {
    public static void main(String args[]) {
        double x[] = {1, 2, 3, 4, 5, 6, 7, 8};
        FFT fft = new FFT(x.length);

        double y[] = fft.forward(x);
        double z[] = fft.backward(y);
        for (int i = 0; i < x.length; i++) {
            z[i] = z[i] / x.length;
        }

        new PrintMatrix("x").print(x);
        new PrintMatrix("y").print(y);
        new PrintMatrix("z").print(z);
    }
}
```

### Output

```
  x
   0
0  1
1  2
2  3
3  4
4  5
5  6
6  7
7  8

     y
     0
0  36
1  -4
2   9.657
```

```
3  -4
4   4
5  -4
6   1.657
7  -4

  z
   0
0  1
1  2
2  3
3  4
4  5
5  6
6  7
7  8
```

## *class* ComplexFFT

Complex FFT.

Class `ComplexFFT` computes the discrete complex Fourier transform of a complex vector of size $N$. The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when $N$ is a product of small prime factors. If $N$ satisfies this condition, then the computational effort is proportional to $N \log N$. This considerable savings has historically led people to refer to this algorithm as the “fast Fourier transform” or FFT.

Specifically, given an $N$-vector $x$, method `forward` returns

$$c_m = \sum_{n=0}^{N-1} x_n e^{-2\pi inm/N}$$

Furthermore, a vector of Euclidean norm $S$ is mapped into a vector of norm

$$\sqrt{N}S$$

Finally, note that we can invert the Fourier transform as follows:

$$x_n = \frac{1}{N} \sum_{j=0}^{N-1} c_m e^{2\pi inj/N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has the coefficients for a trigonometric interpolating polynomial to the data. An unnormalized inverse is implemented in `backward`. `ComplexFFT` is based on the complex FFT in FFTPACK. The package, FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Specifically, given an $N$-vector $c$, `backward` returns

$$s_m = \sum_{n=0}^{N} c_n e^{2\pi inm/N}$$

Furthermore, a vector of Euclidean norm $S$ is mapped into a vector of norm

$$\sqrt{N}S$$

Finally, note that we can invert the inverse Fourier transform as follows:

$$c_n = \frac{1}{N} \sum_{m=0}^{N-1} s_m e^{-2\pi inm/N}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, one has the coefficients for a trigonometric interpolating polynomial to the data. backward is based on the complex inverse FFT in FFTPACK. The package, FFTPACK was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

## Declaration

public class com.imsl.math.ComplexFFT
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *ComplexFFT*
  `public` **ComplexFFT**`( int` **n** `)`
  - **Description**
    Constructs a complex FFT object.
  - **Parameters**
    * `n` – is the array size that this object can handle.

## Methods

---

- *backward*
  `public Complex[]` **backward**`( Complex[]` **coef** `)`
  - **Description**
    Compute the complex periodic sequence from its Fourier coefficients.
  - **Parameters**
    * `coef` – `Complex` array of Fourier coefficients
  - **Returns** – `Complex` array containing the periodic sequence

---

- *forward*
  `public Complex[]` **forward**`( Complex[]` **seq** `)`
  - **Description**
    Compute the Fourier coefficients of a complex periodic sequence.
  - **Parameters**
    * `seq` – is the `Complex` array containing the sequence to be transformed.
  - **Returns** – a `Complex` array containing the transformed sequence.

## Example: Complex FFT

The Fourier coefficients of a complex periodic sequence are computed. Then the coefficients are used to try to reproduce the periodic sequence.

```
import com.imsl.math.*;

public class ComplexFFTEx1 {
    public static void main(String args[]) {
        Complex x[] = {
            new Complex(1,8),
            new Complex(2,7),
            new Complex(3,6),
```

```
            new Complex(4,5),
            new Complex(5,4),
            new Complex(6,3),
            new Complex(7,2),
            new Complex(8,1)
        };
        ComplexFFT fft = new ComplexFFT(x.length);

        Complex y[] = fft.forward(x);
        Complex z[] = fft.backward(y);
        for (int i = 0; i < x.length; i++) {
            z[i] = Complex.divide(z[i], x.length);
        }

        new PrintMatrix("x").print(x);
        new PrintMatrix("y").print(y);
        new PrintMatrix("z").print(z);
    }
}
```

## Output

```
    x
    0
0  1+8i
1  2+7i
2  3+6i
3  4+5i
4  5+4i
5  6+3i
6  7+2i
7  8+1i

         y
          0
0       36+36i
1    5.657+13.657i
2         +8i
3   -2.343+5.657i
4       -4+4i
5   -5.657+2.343i
```

```
6        -8
7  -13.657-5.657i

    z
    0
0  1+8i
1  2+7i
2  3+6i
3  4+5i
4  5+4i
5  6+3i
6  7+2i
7  8+1i
```

# Chapter 7

# Nonlinear Equations

## Classes



## Usage Notes

### Zeros of a Polynomial

A polynomial function of degree $n$ can be expressed as follows:

$$p(z) = a_n z^n n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0$$

where $a_n \neq 0$. The class finds zeros of a polynomial with real or complex coefficients using Aberth's method.

### Zeros of a Function

The class uses Muller's method to find the real zeros of a real-valued function.

### Root of System of Equations

A system of equations can be stated as follows:

$$f_i(x) = 0, \text{for} \quad i = 1, 2, \ldots, n$$

where $x \in \mathbf{R}^n$, and $f_i : \mathbf{R^n} \rightarrow \mathbf{R}$. The `ZeroSystem` class uses a modified hybrid method due to M.J.D. Powell to find the zero of a system of nonlinear equations.

## *class* ZeroPolynomial

The ZeroPolynomial class computes the zeros of a polynomial with complex coefficients, Aberth's method. This class is a Java translation of a Fortran code written by Dario Andrea Bini, University of Pisa, Italy (bini@dm.unipi.it). Numerical computation of polynomial zeros by means of Aberth's method, Numerical Algorithms, 13 (1996), pp. 179-200. The original Fortran code includes the following notice.



### Declaration

public class com.imsl.math.ZeroPolynomial
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Class

*class* **ZeroPolynomial.DidNotConvergeException**

The iteration did not converge

#### Declaration

public static class com.imsl.math.ZeroPolynomial.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *ZeroPolynomial.DidNotConvergeException*
  `public` **ZeroPolynomial.DidNotConvergeException**`(` `java.lang.String` **message** `)`
- *ZeroPolynomial.DidNotConvergeException*
  `public` **ZeroPolynomial.DidNotConvergeException**`(` `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

### Field

---

- public static final `double` **EPSILON_SMALL**
  – The smallest relative spacing for doubles.

### Constructor

---

- *ZeroPolynomial*
  `public` **ZeroPolynomial**`( )`
  – **Description**
    Creates an instance of the solver.

## Methods

---

- *computeRoots*
`public synchronized Complex[] computeRoots( Complex[] coef ) throws com.imsl.math.ZeroPolynomial.DidNotConvergeException`
  - **Description**
  Computes the roots of the polynomial with Complex coefficients.
  $$p(x) = \mathrm{coef}[n] \times x^n + \mathrm{coef}[n-1] \times x^{n-1} + \ldots + \mathrm{coef}[0]$$
  - **Parameters**
    * `coef` – a `Complex` array containing the polynomial coefficients.
  - **Returns** – a `Complex` array containing the roots of the polynomial.

---

- *computeRoots*
`public synchronized Complex[] computeRoots( double[] coef ) throws com.imsl.math.ZeroPolynomial.DidNotConvergeException`
  - **Description**
  Computes the roots of the polynomial with real coefficients.
  $$p(x) = \mathrm{coef}[n] \times x^n + \mathrm{coef}[n-1] \times x^{n-1} + \ldots + \mathrm{coef}[0]$$
  - **Parameters**
    * `coef` – a `double` array containing the polynomial coefficients
  - **Returns** – a `Complex` array containing the roots of the polynomial

---

- *getRadius*
`public double getRadius( int index )`
  - **Description**
  Returns an a-posteriori absolute error bound on the root.
  - **Parameters**
    * `index` – an `int` specifying the (0-based) index of the root whose error bound is to be returned
  - **Returns** – a `double` representing the error bound on the index-th root. NaN is returned if the corresponding root cannot be represented as floating point due to overflow or underflow or if the roots have not yet been computed.

---

- *getRoot*
`public Complex getRoot( int index )`

– **Description**
  Returns a zero of the polynomial.
– **Parameters**
  * `index` – an `int` which specifies the (0-based) index of the root to be returned
– **Returns** – a `Complex` which represents the index-th root of the polynomial

---

• *getRoots*
`public Complex[]` **getRoots**`( )`

– **Description**
  Returns the zeros of the polynomial.
– **Returns** – a `Complex` array containing the roots of the polynomial

---

• *getStatus*
`public boolean` **getStatus**`( int` **index** `)`

– **Description**
  Returns the error status of a root.
– **Parameters**
  * `index` – an `int` representing the (0-based) index of the root whose error status is to be returned
– **Returns** – a `boolean` representing the error status on the index-th root. It is false if the approximation of the index-th root has been carried out successfully, for example, the computed approximation can be viewed as the exact root of a slightly perturbed polynomial. It is true if more iterations are needed for the index-th root.

---

• *setMaxIterations*
`public synchronized void` **setMaxIterations**`( int` **maxIterations** `)`

– **Description**
  Sets the maximum number of iterations allowed. The default value is 30.
– **Parameters**
  * `maxIterations` – an `int` which specifies the maximum number of iterations allowed
– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if `maxIterations` is less than or equal to zero.

### Example 1: Zeros of a Polynomial

The zeros of a polynomial with real coefficients are computed.

```
import com.imsl.math.*;

public class ZeroPolynomialEx1 {
    public static void main(String args[]) throws
    ZeroPolynomial.DidNotConvergeException {
        double coef[] = {-2, 4, -3, 1};

        ZeroPolynomial zp = new ZeroPolynomial();
        Complex root[] = zp.computeRoots(coef);

        for (int k = 0;  k < root.length;  k++) {
            System.out.println("root = " + root[k]);
            System.out.println("    radius = "+ zp.getRadius(k));
            System.out.println("    status = "+ zp.getStatus(k));
        }
    }
}
```

### Output

```
root = 0.9999999999999999-0.9999999999999997i
    radius = 1.9197212602501468E-14
    status = false
root = 1.0000000000000004+1.0000000000000002i
    radius = 1.9618522761623435E-14
    status = false
root = 1.0000000000000002-3.3087224502121107E-24i
    radius = 2.5512925105887074E-14
    status = false
```

### Example 2: Zeros of a Polynomial with Complex Coefficients

The zeros of a polynomial with Complex coefficients are computed.

```
import com.imsl.math.*;

public class ZeroPolynomialEx2 {
    public static void main(String args[]) throws
```

```
    ZeroPolynomial.DidNotConvergeException {
        // Find zeros of z∧3-(3+6i)*z∧2+(-8+12i)*z+10
        Complex coef[] = {
            new Complex(10),
            new Complex(-8, 12),
            new Complex(-3, -6),
            new Complex(1)
        };

        ZeroPolynomial zp = new ZeroPolynomial();
        Complex root[] = zp.computeRoots(coef);

        for (int k = 0;  k < root.length;  k++) {
            System.out.println("root = " + root[k]);
            System.out.println("    radius = "+ zp.getRadius(k));
            System.out.println("    status = "+ zp.getStatus(k));
        }
    }
}
```

## Output

```
root = 1.0+1.0i
    radius = 6.105673569140261E-14
    status = false
root = 1.0000000000000002+2.0000000000000004i
    radius = 1.9846776908049295E-13
    status = false
root = 0.9999999999999992+2.999999999999999i
    radius = 1.5275632034267045E-13
    status = false
```

# *class* **ZeroFunction**

Muller's method to find the zeros of a univariate function, f(x).

`ZeroFunction` computes $n$ real zeros of a real function *f*. Given a user-supplied function *f(x)* and an n-vector of initial guesses $x_1, x_2, \ldots, x_n$, the routine uses Muller's method to locate $n$ real zeros of *f*, that is, $n$ real values of $x$ for which *f(x) = 0*. The routine has two convergence criteria: the first requires that

$$|f\ (x_i^m)|$$

be less than `errorAbsolute`, specified by the `setAbsoluteError` method; the second requires that the relative change of any two successive approximations to an $x_i$ be less than `ErrorRelative`, specified by the `setAbsoluteError` method.

Here,

$$x_i^m$$

is the m-th approximation to $x_i$. Let `errorAbsolute` be $\varepsilon_1$, and `errorRelative` be $\varepsilon_2$. The criteria may be stated mathematically as follows:

Criterion 1:

$$|f\ (x_i^m)| < \ \varepsilon_1$$

Criterion 2:

$$\left|\frac{x_i^{m+1} - x_i^m}{x_i^m}\right| < \ \varepsilon_2$$

"Convergence" is the satisfaction of either criterion.

## Declaration

public class com.imsl.math.ZeroFunction
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Class

### *interface* **ZeroFunction.Function**

Public interface for the user supplied function to ZeroFunction.

#### Declaration

public static interface com.imsl.math.ZeroFunction.Function

### Method

- *f*
  `double` **`f`**`( double` **`x`** `)`
  - **Description**
    Returns the value of the function at the given point.
  - **Parameters**
    * `x` – a `double` specifying the point at which the function is to be evaluated
  - **Returns** – a `double` specifying the value of the function at x

### Constructor

- *ZeroFunction*
  `public` **ZeroFunction**`( )`
  - **Description**
    Creates an instance of the solver.

### Methods

- *allConverged*
  `public synchronized boolean` **allConverged**`( )`
  - **Description**
    Returns true if the iterations for all of the roots have converged.

---

- *computeZeros*
  `public synchronized double[]` **computeZeros**`( ZeroFunction.Function` **objectF**`, double[]` **guess** `)`
  - **Description**
    Returns the zeros of a univariate function.
  - **Parameters**
    * `objectF` – contains the function for which the zeros will be found.
    * `guess` – a `double` array containing an initial guess of the zeros. A zero will be found for each point in guess.

---

- *getIterations*
  `public synchronized int` **getIterations**`( int` **nRoot** `)`
  - **Description**
    Returns the number of iterations used to compute a root.
  - **Parameters**
    * `nRoot` – an `int` specifying the index of the root

---

- *setAbsoluteError*
  `public synchronized void` **setAbsoluteError**`( double` **errorAbsolute** `)`
  - **Description**
    Sets first stopping criterion. A zero x[i] is accepted if $|f(x[i])|$ is less than this tolerance. Its default value is about 1.0e-8.
  - **Parameters**
    * `errorAbsolute` – a `double` value specifying the first stopping criterion
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if errorAbsolute is less than 0

---

- *setMaxIterations*
  `public synchronized void` **setMaxIterations**`( int` **maxIterations** `)`
  - **Description**
    Sets the maximum number of iterations allowed per root. Its default value is 100.
  - **Parameters**
    * `maxIterations` – an `int` specifying the maximum number of iterations allowed per root
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if maxIterations is less than zero.

---

- *setRelativeError*
  `public synchronized void` **setRelativeError**`( double` **errorRelative** `)`
  - **Description**
    Sets second stopping criterion is the relative error. A zero x[i] is accepted if the relative change of two successive approximations to x[i] is less than this tolerance. Its default value is about 1.0e-8.
  - **Parameters**
    * `errorRelative` – a `double` value specifying the second stopping criterion
  - **Throws**

* `java.lang.IllegalArgumentException` – is thrown if errorRelative is less than 0 or greater than 1

---

• *setSpread*
`public synchronized void` **setSpread**`( double` **spread** `)`

– **Description**
Sets the spread. See setSpreadTolerance.

– **Parameters**
* `spread` – is the new spread. Its default value is 1.0.

---

• *setSpreadTolerance*
`public synchronized void` **setSpreadTolerance**`( double` **spreadTolerance** `)`

– **Description**
Sets the spread criteria for multiple zeros. If the zero `x[i]` has been computed and $|x[i] - x[j]| <$ `spreadTolerance`, where `x[j]` is a previously computed zero, then the computation is restarted with a guess equal to `x[i]`+spread. The default value for spreadTolerance is 1.0e-5.

– **Parameters**
* `spreadTolerance` – a `double` value specifying the spread tolerance

– **Throws**
* `java.lang.IllegalArgumentException` – is thrown if spreadTolerance is less than zero.

### Example: Zeros of a Univariate Function

In this example 3 zeros of the sin function are found.

```
import com.imsl.math.*;

public class ZeroFunctionEx1 {
    public static void main(String args[]) {

        ZeroFunction.Function fcn = new ZeroFunction.Function() {
            public double f(double x) {
                return Math.sin(x);
            }
        };

        ZeroFunction zf = new ZeroFunction();
        double guess[] = {5, 18, -6};
```

```
        double zeros[] = zf.computeZeros(fcn, guess);
        for (int k = 0;  k < zeros.length;  k++) {
            System.out.println(zeros[k]+" = "+(zeros[k]/Math.PI) + " pi");
        }
    }
}
```

### Output

```
6.283185307179564 = 1.999999999999993 pi
18.84955592156295 = 6.0000000000077 pi
-6.283185307179641 = -2.0000000000000173 pi
```

## *class* ZeroSystem

Solves a system of n nonlinear equations f(x) = 0 using a modified Powell hybrid algorithm.

`ZeroSystem` is based on the MINPACK subroutine `HYBRD1`, which uses a modification of M.J.D. Powell's hybrid algorithm. This algorithm is a variation of Newton's method, which uses a finite-difference approximation to the Jacobian and takes precautions to avoid large step sizes or increasing residuals. For further description, see More et al. (1980).

A finite-difference method is used to estimate the Jacobian. Whenever the exact Jacobian can be easily provided, `objectF` should implement `ZeroSystem.Jacobian`.

### Declaration

public class com.imsl.math.ZeroSystem
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Classes

#### *class* ZeroSystem.DidNotConvergeException

The iteration did not converge.

### Declaration

public static class com.imsl.math.ZeroSystem.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ZeroSystem.DidNotConvergeException*
  `public` **ZeroSystem.DidNotConvergeException**`( java.lang.String` **message** `)`
- *ZeroSystem.DidNotConvergeException*
  `public` **ZeroSystem.DidNotConvergeException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

## *interface* ZeroSystem.Function

Public interface for user supplied function to `ZeroSystem` object.

### Declaration

public static interface com.imsl.math.ZeroSystem.Function

### Method

---

- *f*
  `void` **f**`( double[]` **x**`, double[]` **f** `)`
  - **Description**
    Returns the value of the function at the given point.
  - **Parameters**
    * `x` – a `double` array which contains the point at which the function is to be evaluated. The contents of this array must not be altered by this function.
    * `f` – a `double` array which contains the value of the function at x.

## *interface* ZeroSystem.Jacobian

Public interface for user supplied function to `ZeroSystem` object.

### Declaration

public static interface com.imsl.math.ZeroSystem.Jacobian
**implements** ZeroSystem.Function

### Method

---

- *jacobian*
  `void` **jacobian**`( double[] x, double[][]` **jac** `)`
  - **Description**
    Returns the value of the Jacobian at the given point.
  - **Parameters**
    * `x` – a `double` array which contains the point at which the Jacobian is to be evaluated. The contents of this array must not be altered by this function.
    * `jac` – a `double` matrix which contains the value of the Jacobian at x. The value of jac[i][j] is the derivative of f[i] with respect to x[j].

## *class* ZeroSystem.ToleranceTooSmallException

Tolerance too small

### Declaration

public static class com.imsl.math.ZeroSystem.ToleranceTooSmallException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *ZeroSystem.ToleranceTooSmallException*
  `public` **ZeroSystem.ToleranceTooSmallException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* ZeroSystem.TooManyIterationsException

Too many iterations.

#### Declaration

public static class com.imsl.math.ZeroSystem.TooManyIterationsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *ZeroSystem.TooManyIterationsException*
  `public` **ZeroSystem.TooManyIterationsException**`( )`

- *ZeroSystem.TooManyIterationsException*
  `public` **ZeroSystem.TooManyIterationsException**`( java.lang.Object[]` **arguments** `)`

- *ZeroSystem.TooManyIterationsException*
  `public` **ZeroSystem.TooManyIterationsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### Constructor

- *ZeroSystem*
  `public` **ZeroSystem**`( int` **n** `)`
  - **Description**
    Creates an object to find the zeros of a system of n equations.
  - **Parameters**
    * `n` – is the number of equations that the solver handles

### Methods

- *setGuess*
  `public void` **setGuess**`( double[]` **xguess** `)`
  - **Description**
    Sets the initial guess for the array x. The default is to set x to all zeros.
  - **Parameters**
    * `xguess` – a `double` array containing the initial guess

- *setMaxIterations*
  `public synchronized void` **setMaxIterations**`( int` **maxIterations** `)`

  – **Description**
    Sets the maximum number of iterations allowed. The default value is 200.
  – **Parameters**
    * `maxIterations` – an `int` specifying the maximum number of iterations allowed
  – **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if maxIterations is less than or equal to zero.

---

- *setRelativeError*
  `public synchronized void` **setRelativeError**`( double` **errorRelative** `)`

  – **Description**
    Sets the relative error tolerance. The root is accepted if the relative error between two successive approximations to this root is within errorRelative. The default is the square root of the precision, about 1.0e-08.
  – **Parameters**
    * `errorRelative` – a `double` specifying the relative error tolerance
  – **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `errorRelative` is less than 0 or greater than 1.

---

- *solve*
  `public synchronized double[]` **solve**`( ZeroSystem.Function` **objectF** `)`
  `throws com.imsl.math.ZeroSystem.TooManyIterationsException,`
  `com.imsl.math.ZeroSystem.ToleranceTooSmallException,`
  `com.imsl.math.ZeroSystem.DidNotConvergeException`

  – **Description**
    Solve a system of nonlinear equations using the Levenberg-Marquardt algorithm
  – **Parameters**
    * `objectF` – defines the function whose zero is to be found. If objectF implements a Jacobian then its Jacobian is used. Otherwise a finite difference is computed.
  – **Returns** – a `double` array containing the solution
  – **Throws**
    * `com.imsl.math.ZeroSystem.TooManyIterationsException` – is thrown if the maximum number of iterations is exceeded
    * `com.imsl.math.ZeroSystem.ToleranceTooSmallException` – is thrown if the error tolerance is too small

* com.imsl.math.ZeroSystem.DidNotConvergeException – is thrown if the algorithm does not converge

### Example: Solve a System of Nonlinear Equations

A system of nonlinear equations is solved.

```
import com.imsl.math.*;

public class ZeroSystemEx1 {
    public static void main(String args[]) throws com.imsl.IMSLException {

        ZeroSystem.Function fcn = new ZeroSystem.Function() {
            public void f(double x[], double f[]) {
                f[0] = x[0] + Math.exp(x[0]-1.0) +
                (x[1]+x[2])*(x[1]+x[2]) - 27.0;
                f[1] = Math.exp(x[1]-2.0)/x[0] + x[2]*x[2] - 10.0;
                f[2] = x[2] + Math.sin(x[1]-2.0) + x[1]*x[1] - 7.0;
            }
        };

        ZeroSystem zf = new ZeroSystem(3);
        double guess[] = {4, 4, 4};
        zf.setGuess(guess);
        new PrintMatrix("zeros").print(zf.solve(fcn));
    }
}
```

### Output

```
zeros
   0
0  1
1  2
2  3
```

# Chapter 8

# Optimization

Classes

## Usage Notes

### Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in R^n} f(x)$$

where $f : \mathbf{R}^n \to \mathbf{R}$ is continuous and has derivatives of all orders required by the algorithms. The functions for unconstrained minimization are grouped into three categories: univariate functions, multivariate functions, and nonlinear least-squares functions.

For the univariate functions, it is assumed that the function is unimodal within the specified interval. For discussion on unimodality, see Brent (1973).

The class `MinUnconMultiVar` finds the minimum of a multivariate function using a quasi-Newton method. The default is to use a finite-difference approximation of the gradient of *f(x)*. Here, the gradient is defined to be the vector

$$\nabla f(x) = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right]$$

However, when the exact gradient can be easily provided, the gradient should be provided by implementing the interface `MinUnconMultiVar.Gradient`.

The nonlinear least-squares function uses a modified Levenberg-Marquardt algorithm. The most common application of the function is the nonlinear data-fitting problem where the user is trying to fit the data with a nonlinear model.

These functions are designed to find only a local minimum point. However, a function may have many local minima. Try different initial points and intervals to obtain a better local solution.

### Linearly Constrained Minimization

The linearly constrained minimization problem can be stated as follows:

$$\begin{array}{l} \min\limits_{x \in R^n} f(x) \\ \text{subject to } \ A_1 x = b_1 \end{array}$$

where $f : \mathbf{R}^n \to \mathbf{R}$, $A_1$ and $A_2$ are coefficient matrices, and $b_1$ and $b_2$ are vectors. If *f(x)* is linear, then the problem is a linear programming problem. If *f(x)* is quadratic, the problem is a quadratic programming problem.

The class LinearProgramming uses a revised simplex method to solve small- to medium-sized linear programming problems. No sparsity is assumed since the coefficients are stored in full matrix form.

The class QuadraticProgramming is designed to solve convex quadratic programming problems using a dual quadratic programming algorithm. If the given Hessian is not positive definite, then QuadraticProgramming modifies it to be positive definite. In this case, output should be interpreted with care because the problem has been changed slightly. Here, the Hessian of *f(x)* is defined to be the $n$ x $n$ matrix

$$\nabla^2 f(x) = \left[\frac{\partial^2}{\partial x_i \partial x_j} f(x)\right]$$

### Nonlinearly Constrained Minimization

The nonlinearly constrained minimization problem can be stated as follows:

$$\begin{array}{l} \min\limits_{x \in R^n} f(x) \\ \text{subject to } \; g_i(x) = 0 \;\; \text{for } \; i = 1, \; 2, \; \ldots, \; m_1 \\ g_i(x) \geq 0 \;\; \text{for } \; i = m_1 + 1, \; \ldots, \; m \end{array}$$

where $f : \mathbf{R}^n \to \mathbf{R}$ and $g_i : \mathbf{R}^n \to \mathbf{R}$, for $i = 1, 2, \ldots, m$.

The class MinConNLP uses a sequential equality constrained quadratic programming algorithm to solve this problem. A more complete discussion of this algorithm can be found in the documentation.

## *class* MinUncon

Unconstrained minimization.

MinUncon uses two separate algorithms to compute the minimum depending on what the user supplies as the function f.

If f defines the function whose minimum is to be found MinUncon uses a safeguarded quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the routine ZXLSF written by M.J.D. Powell at the University of Cambridge.

MinUncon finds the least value of a univariate function, *f*, where f implements MinUnconFunction f. Optional data include an initial estimate of the solution, and a positive number bound, specified by the setBound method. Let $x_0 = xguess$ where xguess is specified by the `setGuess` method and $b$ = *bound*, then $x$ is restricted to the interval $[x_0 - b, x_0 + b]$. Usually, the algorithm begins the search by moving from $x_0$ to $x = x_0 + s$, where $s$ = *step*. `step` is set by the `setStep` method. If `setStep` is not called then `step` is set to *0.1*. `step` may be positive or negative. The first two function evaluations indicate the direction to the minimum point, and the search strides out along this direction until a bracket on a minimum point is found or until $x$ reaches one of the bounds $x_0 \pm b$. During this stage, the step length increases by a factor of between two and nine per function evaluation; the factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, we will have three points, $x_1$, $x_2$, and $x_3$, with $x_1 < x_2 < x_3$ and $f(x_2) \le f(x_1)$ and $f(x_2) \le f(x_3)$. There are three main ingredients in the technique for choosing the new $x$ from these three points. They are (i) the estimate of the minimum point that is given by quadratic interpolation of the three function values, (ii) a tolerance parameter $\varepsilon$, that depends on the closeness of $f$ to a quadratic, and (iii) whether $x_2$ is near the center of the range between $x_1$ and $x_3$ or is relatively close to an end of this range. In outline, the new value of $x$ is as near as possible to the predicted minimum point, subject to being at least $\varepsilon$ from $x_2$, and subject to being in the longer interval between $x_1$ and $x_2$ or $x_2$ and $x_3$ when $x_2$ is particularly close to $x_1$ or $x_3$. There is some elaboration, however, when the distance between these points is close to the required accuracy; when the distance is close to the machine precision; or when $\varepsilon$ is relatively large.

The algorithm is intended to provide fast convergence when $f$ has a positive and continuous second derivative at the minimum and to avoid gross inefficiencies in pathological cases, such as

$$f(x) = x + 1.001\,|x|$$

The algorithm can make $\varepsilon$ large automatically in the pathological cases. In this case, it is usual for a new value of $x$ to be at the midpoint of the longer interval that is adjacent to the least calculated function value. The midpoint strategy is used frequently when changes to $f$ are dominated by computer rounding errors, which will almost certainly happen if the user requests an accuracy that is less than the square root of the machine precision. In such cases, the routine claims to have achieved the required accuracy if it knows that there is a local minimum point within distance $\delta$ of $x$, where $\delta = xacc$, specified by the `setAccuracy` method even though the rounding errors in $f$ may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high precision arithmetic is recommended.

If `f` implements `MinUnconDerivative` then `MinUncon` uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the routine terminates with a solution.

Otherwise, the point with least function value will be used as the starting point.

From the starting point, say $x_c$, the function value $f_c = f(x_c)$, the derivative value $g_c = g(x_c)$, and a new point $x_n$ defined by $x_n = x_c - g_c$ are computed. The function $f_n = f(x_n)$, and the derivative $g_n = g(x_n)$ are then evaluated. If either $f_n \geq f_c$ or $g_n$ has the opposite sign of $g_c$, then there exists a minimum point between $x_c$ and $x_n$; and an initial interval is obtained. Otherwise, since $x_c$ is kept as the point that has lowest function value, an interchange between $x_n$ and $x_c$ is performed. The secant method is then used to get a new point

$$x_s = x_c - g_c\left(\frac{g_n - g_c}{x_n - x_c}\right)$$

Let $x_n \leftarrow x_s$ and repeat this process until an interval containing a minimum is found or one of the convergence criteria is satisfied. The convergence criteria are as follows:
Criterion 1:

$$|x_c - x_n| \leq \ \varepsilon_c$$

Criterion 2:

$$|g_c| \leq \ \varepsilon_g$$

where $\varepsilon_c = \max\{1.0, |x_c|\}\,\varepsilon$, $\varepsilon$ is a relative error tolerance and $\varepsilon_c$ is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. Function and derivative are then evaluated at that point; and accordingly, a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval reduces by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this procedure is repeated until one of the stopping criteria is met.

## Declaration

public class com.imsl.math.MinUncon
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *interface* MinUncon.Function

Public interface for the user supplied function to the `MinUncon` object.

#### Declaration

public static interface com.imsl.math.MinUncon.Function

#### Method

---

- *f*
  `double f( double x )`
  - **Description**
    Public interface for the smooth function of a single variable to be minimized.
  - **Parameters**
    * `x` – a `double`, the point at which the function is to be evaluated
  - **Returns** – a `double`, the value of the function at x

### *interface* MinUncon.Derivative

Public interface for the user supplied function to the `MinUncon` object.

#### Declaration

public static interface com.imsl.math.MinUncon.Derivative
**implements** MinUncon.Function

#### Method

---

- *g*
  `double g( double x )`
  - **Description**
    Public interface for the smooth function of a single variable to be minimized.

- **Parameters**
  - `x` – a `double`, the point at which the derivative of the function is to be evaluated
- **Returns** – a `double`, the value of the derivative of the function at x

## Constructor

---

- *MinUncon*
  `public` **MinUncon**`( )`
  - **Description**
    Unconstrained minimum constructor for a smooth function of a single variable of type `double`.

## Methods

---

- *computeMin*
  `public double` **computeMin**`( MinUncon.Function` **F** `)`
  - **Description**
    Return the minimum of a smooth function of a single variable of type `double` using function values only or using function values and derivatives.
  - **Parameters**
    - `F` – defines the function whose minimum is to be found. If `F` implements `Derivative` then derivatives are used. Otherwise, an attempt to find the minimum is made using function values only.
  - **Returns** – a `double` scalar value containing the minimum of the input function

- *setAccuracy*
  `public void` **setAccuracy**`( double` **xacc** `)`
  - **Description**
    Set the required absolute accuracy in the final value returned by member function `computeMin`. If this member function is not called, the required accuracy is set to 1.0e-8.
  - **Parameters**
    - `xacc` – a `double`scalar value specifying the required absolute accuracy in the final value returned by member function `computeMin`.

---

- *setBound*
  `public void` **setBound**`( double` **bound** `)`
  - **Description**
    Set the amount by which X may be changed from its initial value, xguess. If this member function is not called, bound is set to 100.
  - **Parameters**
    * `bound` – a `double` scalar value specifying the amount by which X may be changed from its initial value. In other words, X is restricted to the interval [`xguess-bound`, `xguess+bound`].

---

- *setDerivtol*
  `public void` **setDerivtol**`( double` **gtol** `)`
  - **Description**
    Set the derivative tolerance used by member function `computeMin` to decide if the current point is a local minimum. This is the second stopping criterion. x is returned as a solution when G(x) is less than or equal to `gtol`. `gtol` should be nonnegative, otherwise zero will be used. If this member function is not called, the derivative tolerance is set to 1.0e-8.
  - **Parameters**
    * `gtol` – a `double`scalar value specifying the derivative tolerance used by member function `computeMin`.

---

- *setGuess*
  `public void` **setGuess**`( double` **xguess** `)`
  - **Description**
    Set the initial guess of the minimum point of the input function. If this member function is not called, an initial guess of 0.0 is used.
  - **Parameters**
    * `xguess` – a `double` scalar value specifying the initial guess of the minimum point of the input function

---

- *setStep*
  `public void` **setStep**`( double` **step** `)`
  - **Description**
    Set the stepsize to use when changing x. If this member function is not called, step is set to 0.1.
  - **Parameters**
    * `step` – a `double` scalar value specifying the order of magnitude estimate of the required change in $x$ when stepping towards the minimum

### Example 1: Minimum of a smooth function

The minimum of $e^x - 5x$ is found using function evaluations only.

```
import com.imsl.math.*;

public class MinUnconEx1 {
    public static void main(String args[]) {
        MinUncon zf = new MinUncon();
        zf.setGuess(0.0);
        zf.setAccuracy(0.001);
        MinUncon.Function fcn = new MinUncon.Function() {
            public double f(double x) {
                return Math.exp(x) - 5.*x;
            }
        };
        System.out.println("Minimum is " + zf.computeMin(fcn));
    }
}
```

### Output

```
Minimum is 1.6094175999200253
```

### Example 2: Minimum of a smooth function

The minimum of $e^x - 5x$ is found using function evaluations and first derivative evaluations.

```
import com.imsl.math.*;

public class MinUnconEx2 implements MinUncon.Derivative {
    public double f(double x) {
        return Math.exp(x) - 5.*x;
    }

    public double g(double x) {
        return Math.exp(x) - 5.;
    }

    public static void main(String args[]) {
        int n = 1;
```

```
        double xinit = 0.;
        double x[] = {0.};
        MinUncon zf = new MinUncon();
        zf.setGuess(xinit);
        zf.setAccuracy(.001);
        MinUnconEx2 fcn = new MinUnconEx2();
        x[0] = zf.computeMin(fcn);
        for (int k = 0;  k < n;  k++) {
            System.out.println("x["+k+"] = "+x[k]);
        }
    }
}
```

### Output

```
x[0] = 1.6100113162270329
```

## *class* MinUnconMultiVar

Unconstrained multivariate minimization.

Class `MinUnconMultivar` uses a quasi-Newton method to find the minimum of a function *f(x)* of $n$ variables. The problem is stated as follows:

$$\min_{x \in R^n} f(x)$$

Given a starting point $x_c$, the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where $B$ is a positive definite approximation of the Hessian, and $g_c$ is the gradient evaluated at $x_c$. A line search is then used to find a new point

$$x_n = x_c + \lambda d, \lambda > 0$$

such that

$$f(x_n) \le f(x_c) + \alpha g^T d, \quad \alpha \in (0,\, 0.5)$$

Finally, the optimality condition $||g(x)|| \leq \varepsilon$ where $\varepsilon$ is a gradient tolerance.

When optimality is not achieved, $B$ is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^TB}{s^TBs} + \frac{yy^T}{y^Ts}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, see Dennis and Schnabel (1983, Appendix A).

In this implementation, the first stopping criterion for `MinUnconMultivar` occurs when the norm of the gradient is less than the given gradient tolerance `gradientTolerance`. The second stopping criterion for `MinUnconMultivar` occurs when the scaled distance between the last two steps is less than the step tolerance `stepTolerance`.

Since by default, a finite-difference method is used to estimate the gradient for some single precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. Supply gradient for a more accurate gradient evaluation (`setGradient`).

## Declaration

public class com.imsl.math.MinUnconMultiVar
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *interface* **MinUnconMultiVar.Function**

Public interface for the user supplied function to the `MinUnconMultiVar` object.

#### Declaration

public static interface com.imsl.math.MinUnconMultiVar.Function

#### Method

- *f*
  `double f( double[] x )`

– **Description**
  Public interface for the multivariate function to be minimized.
– **Parameters**
  * `x` – a `double` array, the point at which the function is to be evaluated
– **Returns** – a `double`, the value of the function at x

## *interface* MinUnconMultiVar.Gradient

Public interface for the user supplied gradient to the `MinUnconMultiVar` object.

### Declaration

public static interface com.imsl.math.MinUnconMultiVar.Gradient
**implements** MinUnconMultiVar.Function

### Method

---

- *gradient*
  `void` **gradient**`( double[] x, double[]` **gradient** `)`
  – **Description**
    Public interface for the gradient of the multivariate function to be minimized.
  – **Parameters**
    * `x` – a `double` array, the point at which the gradient of the function is to be evaluated
    * `gradient` – a `double` array, the value of the gradient of the function at x

## *class* MinUnconMultiVar.ApproximateMinimumException

Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or the scaled step tolerance is too big.

### Declaration

public static class com.imsl.math.MinUnconMultiVar.ApproximateMinimumException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinUnconMultiVar.ApproximateMinimumException*
  `public` **MinUnconMultiVar.ApproximateMinimumException**`(`
  `java.lang.String` **message** `)`
- *MinUnconMultiVar.ApproximateMinimumException*
  `public` **MinUnconMultiVar.ApproximateMinimumException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* MinUnconMultiVar.FalseConvergenceException

False convergence error; the iterates appear to be converging to a noncritical point. Possibly incorrect gradient information is used, or the function is discontinuous, or the other stopping tolerances are too tight.

### Declaration

public static class com.imsl.math.MinUnconMultiVar.FalseConvergenceException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinUnconMultiVar.FalseConvergenceException*
  `public` **MinUnconMultiVar.FalseConvergenceException**`(`
  `java.lang.String` **message** `)`
- *MinUnconMultiVar.FalseConvergenceException*
  `public` **MinUnconMultiVar.FalseConvergenceException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* MinUnconMultiVar.MaxIterationsException

Maximum number of iterations exceeded.

### Declaration

public static class com.imsl.math.MinUnconMultiVar.MaxIterationsException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinUnconMultiVar.MaxIterationsException*
  `public` **MinUnconMultiVar.MaxIterationsException**`( java.lang.String` **message** `)`

- *MinUnconMultiVar.MaxIterationsException*
  `public` **MinUnconMultiVar.MaxIterationsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* MinUnconMultiVar.UnboundedBelowException

Five consecutive steps of the maximum allowable stepsize have been taken, either the function is unbounded below, or has a finite asymptote in some direction or the maximum allowable step size is too small.

### Declaration

public static class com.imsl.math.MinUnconMultiVar.UnboundedBelowException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinUnconMultiVar.UnboundedBelowException*
  `public` **MinUnconMultiVar.UnboundedBelowException**`( java.lang.String` **message** `)`

- *MinUnconMultiVar.UnboundedBelowException*
  `public` **MinUnconMultiVar.UnboundedBelowException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## Constructor

---

- *MinUnconMultiVar*
  `public` **MinUnconMultiVar**`( int` **n** `)`

- **Description**
  Unconstrained minimum constructor for a function of n variables of type `double`.
- **Parameters**
  * `n` – An `int` scalar value which defines the number of variables of the function whose minimum is to be found.

## Methods

---

- *computeMin*
  `public double[]` **computeMin**`( MinUnconMultiVar.Function` **F** `) throws com.imsl.math.MinUnconMultiVar.FalseConvergenceException, com.imsl.math.MinUnconMultiVar.MaxIterationsException, com.imsl.math.MinUnconMultiVar.UnboundedBelowException`
  - **Description**
    Return the minimum point of a function of n variables of type `double` using a finite-difference gradient or using a user-supplied gradient.
  - **Parameters**
    * `F` – defines the function whose minimum is to be found. `F` can be used to supply a gradient of the function. If `F` implements `Gradient` then the user-supplied gradient is used. Otherwise, an attempt to find the minimum is made using a finite-difference gradient.
  - **Returns** – a `double` array containing the point at which the minimum of the input function occurs.

- *getErrorStatus*
  `public int` **getErrorStatus**`( )`
  - **Description**
    Returns the non-fatal error status.
  - **Returns** – an `int` specifying the non-fatal error status:

| Status | Meaning |
|---|---|
| 1 | The last global step failed to locate a lower point than the current $x$ value. The current $x$ may be an approximate local minimizer and no more accuracy is possible or the step tolerance may be too large. |
| 2 | Relative function convergence; both the actual and predicted relative reductions in the function are less than or equal to the relative function convergence tolerance. |
| 3 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or the step tolerance is too big. |

---

- *getIterations*
  `public synchronized int` **getIterations**`( )`
  - **Description**
    Returns the number of iterations used to compute a minimum.
  - **Returns** – an `int` specifying the number of iterations used to compute the minimum.

---

- *setDigits*
  `public void` **setDigits**`( double` **fdigit** `)`
  - **Description**
    Set the number of good digits in the function. If this member function is not called, fdigit is set to 15.0.
  - **Parameters**
    * `fdigit` – a `double` scalar value specifying the number of good digits in the user supplied function
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `fdigit` is less than or equal to 0

---

- *setFalseConvergenceTolerance*
  `public void` **setFalseConvergenceTolerance**`( double` **falseConvergenceTolerance** `)`

- **Description**
  Set the false convergence tolerance. If this member function is not called, 2.22044604925031308e-14 is used as the false convergence tolerance.
- **Parameters**
  * `falseConvergenceTolerance` – a double scalar value specifying the false convergence tolerance
- **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if falseConvergenceTolerance is less than or equal to 0

---

- *setFscale*
  `public void` **setFscale**`( double` **fscale** `)`
  - **Description**
    Set the function scaling value for scaling the gradient. If this member function is not called, the value of this scalar is set to 1.0.
  - **Parameters**
    * `fscale` – a `double` scalar specifying the function scaling value for scaling the gradient
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `fscale` is less than or equal to 0.

---

- *setGradientTolerance*
  `public void` **setGradientTolerance**`( double` **gradientTolerance** `)`
  - **Description**
    Sets the gradient tolerance. This first stopping criterion for this optimizer is that the norm of the gradient be less than the gradient tolerance. If this member function is not called, the cube root of machine precision squared is used to compute the gradient.
  - **Parameters**
    * `gradientTolerance` – a `double` specifying the gradient tolerance used to compute the gradient
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if gradientTolerance is less than or equal to 0

---

- *setGuess*
  `public void` **setGuess**`( double[]` **xguess** `)`
  - **Description**
    Set the initial guess of the minimum point of the input function. If this member function is not called, the elements of this array are set to 0.0..

- **Parameters**
  - `xguess` – a `double` array specifying the initial guess of the minimum point of the input function

---

- *setIhess*

  `public void` **setIhess**`( int` **ihess** `)`

  - **Description**

    Set the Hessian initialization parameter. If this member function is not called, ihess is set to 0.0 and the Hessian is initialized to the identity matrix. If this member function is called and ihess is set to anything other than 0.0, the Hessian is initialized to the diagonal matrix containing max(abs(f(xguess)),fscale)*xscale*xscale
  - **Parameters**
    - `ihess` – an `int` scalar value specifying the Hessian initialization parameter. If `ihess` = 0.0 the Hessian is initialized to the identity matrix. Otherwise, the Hessian is initialized to the diagonal matrix containing max(abs(f(xguess)),fscale)*xscale*xscale where xguess is the initial guess of the computed solution and xscale is the scaling vector for the variables.

---

- *setMaximumStepsize*

  `public void` **setMaximumStepsize**`( double` **maximumStepsize** `)`

  - **Description**

    Set the maximum allowable stepsize to use. If this member function is not called, maximum stepsize is set to a default value based on a scaled xguess.
  - **Parameters**
    - `maximumStepsize` – a nonnegative `double` value specifying the maximum allowable stepsize
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown if maximumStepsize is less than or equal to 0

---

- *setMaxIterations*

  `public void` **setMaxIterations**`( int` **maxIterations** `)`

  - **Description**

    Set the maximum number of iterations allowed. If this member function is not called, the maximum number of iterations is set to 100.
  - **Parameters**
    - `maxIterations` – an `int` specifying the maximum number of iterations allowed
  - **Throws**

    * `java.lang.IllegalArgumentException` – is thrown if maxIterations is less than or equal to 0

---

- *setRelativeTolerance*

  `public void` **setRelativeTolerance**`( double` **relativeTolerance** `)`

  - **Description**
    Set the relative function tolerance. If this member function is not called, 3.66685e-11 is used as the relative function tolerance.
  - **Parameters**
    * `relativeTolerance` – a `double` scalar value specifying the relative function tolerance
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if relativeTolerance is less than or equal to 0

---

- *setStepTolerance*

  `public void` **setStepTolerance**`( double` **stepTolerance** `)`

  - **Description**
    Set the scaled step tolerance to use when changing x. If this member function is not called, the scaled step tolerance is set to 3.66685e-11.
    The second stopping criterion for this optimizer is that the scaled distance between the last two steps be less than the step tolerance.
  - **Parameters**
    * `stepTolerance` – a `double` scalar value specifying the scaled step tolerance. The i-th component of the scaled step between two points x and y is computed as abs(x(i)-y(i))/max(abs(x(i)),1/xscale(i)) where xscale is the scaling vector for the variables.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if stepTolerance is less than or equal to 0

---

- *setXscale*

  `public void` **setXscale**`( double[]` **xscale** `)`

  - **Description**
    Set the diagonal scaling matrix for the variables. If this member function is not called, the elements of this array are set to 1.0..
  - **Parameters**
    * `xscale` – a `double` array specifying the diagonal scaling matrix for the variables
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if any of the elements of xscale is less than or equal to 0

### Example 1: Minimum of a multivariate function

The minimum of $100(x_2 - x_1^2)^2 + (1 - x_1)^2$ is found using function evaluations only.

```
import com.imsl.math.*;

public class MinUnconMultiVarEx1 {
    public static void main(String args[]) throws Exception {
        MinUnconMultiVar solver = new MinUnconMultiVar(2);
        solver.setGuess(new double[]{-1.2, 1.0});
        double x[] = solver.computeMin(new MinUnconMultiVar.Function() {
            public double f(double[] x) {
                return 100.*((x[1] - x[0] * x[0]) * (x[1] - x[0] * x[0])) +
                (1. - x[0]) * (1. - x[0]);
            }
        });
        System.out.println("Minimum point is (" +x[0] +", "+x[1]+")");
    }
}
```

### Output

```
Minimum point is (0.9999999672651304, 0.9999999330452095)
```

### Example 2: Minimum of a multivariate function

The minimum of $100(x_2 - x_1^2)^2 + (1 - x_1)^2$ is found using function evaluations and a user supplied gradient.

```
import com.imsl.math.*;

public class MinUnconMultiVarEx2  {

    static class MyFunction implements MinUnconMultiVar.Gradient {
        public double f(double[] x) {
            return 100.*((x[1] - x[0] * x[0]) * (x[1] - x[0] * x[0])) +
            (1. - x[0]) * (1. - x[0]);
        }
        public void gradient(double[] x, double[] gp) {
            gp[0] =  -400. * (x[1] - x[0] * x[0]) * x[0] - 2. * (1. - x[0]);
            gp[1] = 200. * (x[1] - x[0]*x[0]);
        }
```

```
    }

    public static void main(String args[]) throws Exception {
        MinUnconMultiVar solver = new MinUnconMultiVar(2);
        solver.setGuess(new double[]{-1.2, 1.0});
        double x[] = solver.computeMin(new MyFunction());
        System.out.println("Minimum point is (" +x[0] +", "+x[1]+")");
    }
}
```

### Output

```
Minimum point is (0.9999999668823014, 0.9999999322542452)
```

## *class* NonlinLeastSquares

Nonlinear least squares.

`NonlinLeastSquares` is based on the MINPACK routine `LMDIF` by Mor et al. (1980). It uses a modified Levenberg-Marquardt method to solve nonlinear least squares problems. The problem is stated as follows:

$$\min_{x \in R^n} \frac{1}{2} F(x)^T F(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

where $m \geq n$, $F: R^n \rightarrow R^m$, and $f_i(x)$ is the i-th component function of *F(x)*. From a current point, the algorithm uses the trust region approach:

$$\min_{x_n \in R^n} \|F(x_c) + J(x_c)(x_n - x_c)\|_2$$

subject to

$$\|x_n - x_c\|_2 \leq \delta_c$$

to get a new point $x_n$, which is computed as

$$x_n = x_c - \left(J(x_c)^T J(x_c) + \mu_c I\right)^{-1} J(x_c)^T F(x_c)$$

where $\mu_c = 0$ if $\delta_c \geq \left\| \left( J(x_c)^T J(x_c) \right)^{-1} J(x_c)^T F(x_c) \right\|_2$ and $\mu_c > 0$ otherwise. $F(x_c)$ and $J(x_c)$ are the function values and the Jacobian evaluated at the current point $x_c$. This procedure is repeated until the stopping criteria are satisfied. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

A finite-difference method is used to estimate the Jacobian when the user supplied function, `f`, defines the least-squares problem. Whenever the exact Jacobian can be easily provided, `f` should implement `NonlinLeastSquares.Jacobian`.

## Declaration

public class com.imsl.math.NonlinLeastSquares
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* NonlinLeastSquares.FalseConvergenceException

The iterates appear to be converging to a non-critical point.

#### Declaration

public static class com.imsl.math.NonlinLeastSquares.FalseConvergenceException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *NonlinLeastSquares.FalseConvergenceException*
  `public` **NonlinLeastSquares.FalseConvergenceException**`(`
  `java.lang.String` **message** `)`
- *NonlinLeastSquares.FalseConvergenceException*
  `public` **NonlinLeastSquares.FalseConvergenceException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* NonlinLeastSquares.RelativeFunctionConvergenceException

The scaled and predicted reductions in the function are less than or equal to the relative function convergence tolerance.

#### Declaration

public static class com.imsl.math.NonlinLeastSquares.RelativeFunctionConvergenceException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *NonlinLeastSquares.RelativeFunctionConvergenceException*
  `public` **NonlinLeastSquares.RelativeFunctionConvergenceException**`( java.lang.String` **message** `)`
- *NonlinLeastSquares.RelativeFunctionConvergenceException*
  `public` **NonlinLeastSquares.RelativeFunctionConvergenceException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* NonlinLeastSquares.StepToleranceException

Various possible errors involving the step tolerance.

#### Declaration

public static class com.imsl.math.NonlinLeastSquares.StepToleranceException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *NonlinLeastSquares.StepToleranceException*
  `public` **NonlinLeastSquares.StepToleranceException**`( java.lang.String` **message** `)`
- *NonlinLeastSquares.StepToleranceException*
  `public` **NonlinLeastSquares.StepToleranceException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **NonlinLeastSquares.StepMaxException**

Either the function is unbounded below, has a finite asymptote in some direction, or the maximum stepsize is too small.

#### Declaration

public static class com.imsl.math.NonlinLeastSquares.StepMaxException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *NonlinLeastSquares.StepMaxException*
  `public` **NonlinLeastSquares.StepMaxException**`( java.lang.String` **message** `)`
- *NonlinLeastSquares.StepMaxException*
  `public` **NonlinLeastSquares.StepMaxException**`( java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

### *class* **NonlinLeastSquares.TooManyIterationsException**

Too many iterations.

#### Declaration

public static class com.imsl.math.NonlinLeastSquares.TooManyIterationsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *NonlinLeastSquares.TooManyIterationsException*
  `public` **NonlinLeastSquares.TooManyIterationsException**`( )`
- *NonlinLeastSquares.TooManyIterationsException*
  `public` **NonlinLeastSquares.TooManyIterationsException**`(` `java.lang.Object[]` **arguments** `)`

- *NonlinLeastSquares.TooManyIterationsException*
  `public` **NonlinLeastSquares.TooManyIterationsException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

*interface* **NonlinLeastSquares.Function**

Public interface for the user supplied function to the `NonlinLeastSquares` object.

**Declaration**

public static interface com.imsl.math.NonlinLeastSquares.Function

**Method**

---

- *f*
  `void` **f**`( double[] x, double[]` **f** `)`
  - **Description**
    Public interface for the nonlinear least-squares function.
  - **Parameters**
    * `x` – a `double` array containing the point at which the function is to be evaluated. The contents of this array must not be altered by this function.
    * `f` – a `double` array containing the returned value of the function at x.

*interface* **NonlinLeastSquares.Jacobian**

Public interface for the user supplied function to the `NonlinLeastSquares` object.

**Declaration**

public static interface com.imsl.math.NonlinLeastSquares.Jacobian
**implements** NonlinLeastSquares.Function

**Method**

---

- *jacobian*
  `void` **jacobian**`(` `double[]` **x**`,` `double[][]` **jacobian** `)`

  – **Description**
    Public interface for the nonlinear least squares function.
  – **Parameters**
    * `x` – is a `double` array containing the point at which the Jacobian of the function is to be evaluated
    * `jacobian` – is a `double` matrix containing the returned value of the Jacobian of the function at x

## Constructor

- *NonlinLeastSquares*
  `public` **NonlinLeastSquares**`(` `int` **m**`,` `int` **n** `)`

  – **Description**
    Creates an object to solve a nonlinear least squares problem.
  – **Parameters**
    * `m` – is the number of functions
    * `n` – is the number of variables. n must be less than or equal to m.

## Methods

- *getErrorStatus*
  `public int` **getErrorStatus**`( )`

  – **Description**
    Get information about the performance of NonlinLeastSquares.

– **Returns** – an `int` specifying information about convergence.

| *value* | *meaning* |
|---|---|
| 0 | All convergence tests were met. |
| 1 | Scaled step tolerance was satisfied. The current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or StepTolerance is too big. |
| 2 | Scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance RelativeTolerance. |
| 3 | Iterates appear to be converging to a noncritical point. Incorrect gradient information, a discontinuous function, or stopping tolerances being too tight may be the cause. |
| 4 | Five consecutive steps with the maximum stepsize have been taken. Either the function is unbounded below, or has a finite asymptote in some direction, or the maximum stepsize is too small. |

---

- *setAbsoluteTolerance*
  `public void` **setAbsoluteTolerance**`( double` **absoluteTolerance** `)`
  – **Description**
  Set the absolute function tolerance. If this member function is not called, 1.0e-32 is used as the absolute function tolerance.
  – **Parameters**
    * `absoluteTolerance` – a `double` scalar value specifying the absolute function tolerance
  – **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if absoluteTolerance is less than or equal to 0

---

- *setDigits*
  `public void` **setDigits**`( int` **ngood** `)`
  – **Description**

Set the number of good digits in the function. If this member function is not called, the number of good digits is set to 7.

– **Parameters**
  * `ngood` – an `int` specifying the number of good digits in the user supplied function which defines the least-squares problem

– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if ngood is less than or equal to 0

---

• *setFalseConvergenceTolerance*
`public void` **setFalseConvergenceTolerance**`( double` **falseConvergenceTolerance** `)`

– **Description**
Set the false convergence tolerance. If this member function is not called, 100.0e-16 is used as the false convergence tolerance.

– **Parameters**
  * `falseConvergenceTolerance` – a double scalar value specifying the false convergence tolerance

– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if falseConvergenceTolerance is less than or equal to 0

---

• *setFscale*
`public void` **setFscale**`( double[]` **fscale** `)`

– **Description**
Set the diagonal scaling matrix for the functions. If this member function is not called, the identity is used.

– **Parameters**
  * `fscale` – a `double` array specifying the diagonal scaling matrix for the functions

– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if any of the elements of fscale is less than or equal to 0

---

• *setGradientTolerance*
`public void` **setGradientTolerance**`( double` **gradientTolerance** `)`

– **Description**
Set the gradient tolerance used to compute the gradient. If this member function is not called, the cube root of machine precision squared is used to compute the gradient.

- **Parameters**
  - `gradientTolerance` – a `double` specifying the gradient tolerance used to compute the gradient
- **Throws**
  - `java.lang.IllegalArgumentException` – is thrown if gradientTolerance is less than or equal to 0

---

- *setGuess*

  `public void` **setGuess**`( double[]` **xguess** `)`

  - **Description**

    Set the initial guess of the minimum point of the input function. If this member function is not called, an initial guess of 0.0 is used.
  - **Parameters**
    - `xguess` – a `double` array specifying the initial guess of the minimum point of the input function

---

- *setInitialTrustRegion*

  `public void` **setInitialTrustRegion**`( double` **initialTrustRegion** `)`

  - **Description**

    Set the initial trust region radius. If this member function is not called, a default is set based on the initial scaled Cauchy step.
  - **Parameters**
    - `initialTrustRegion` – a `double` scalar value specifying the initial trust region radius
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown if initialTrustRegion is less than or equal to 0

---

- *setMaximumStepsize*

  `public void` **setMaximumStepsize**`( double` **maximumStepsize** `)`

  - **Description**

    Set the maximum allowable stepsize to use. If this member function is not called, maximum stepsize is set to a default value based on a scaled xguess.
  - **Parameters**
    - `maximumStepsize` – a nonnegative `double` value specifying the maximum allowable stepsize
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown if maximumStepsize is less than or equal to 0

---

- *setMaxIterations*
  `public void` **setMaxIterations**`( int` **maxIterations** `)`
  - **Description**
    Set the maximum number of iterations allowed. If this member function is not called, the maximum number of iterations is set to 100.
  - **Parameters**
    * `maxIterations` – an `int` specifying the maximum number of iterations allowed
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if maxIterations is less than or equal to 0

---

- *setRelativeTolerance*
  `public void` **setRelativeTolerance**`( double` **relativeTolerance** `)`
  - **Description**
    Set the relative function tolerance. If this member function is not called, 1.0e-20 is used as the relative function tolerance.
  - **Parameters**
    * `relativeTolerance` – a `double` scalar value specifying the relative function tolerance
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if relativeTolerance is less than or equal to 0

---

- *setStepTolerance*
  `public void` **setStepTolerance**`( double` **stepTolerance** `)`
  - **Description**
    Set the step tolerance used to step between two points. If this member function is not called, the cube root of machine precision is used as the step tolerance.
  - **Parameters**
    * `stepTolerance` – a `double` scalar value specifying the step tolerance used to step between two points
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if stepTolerance is less than or equal to 0

---

- *setXscale*
  `public void` **setXscale**`( double[]` **xscale** `)`

- **Description**
  Set the diagonal scaling matrix for the variables. If this member function is not called, the identity is used.
- **Parameters**
  * `xscale` – a `double` array specifying the diagonal scaling matrix for the variables
- **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if any of the elements of xscale is less than or equal to 0

---

- *solve*
  `public double[] solve( NonlinLeastSquares.Function F ) throws com.imsl.math.NonlinLeastSquares.TooManyIterationsException`
  - **Description**
    Solve a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm and a Jacobian.
  - **Parameters**
    * `F` – User supplied function that defines the least-squares problem. If F implements Jacobian then its Jacobian is used. Otherwise, a finite difference Jacobian is used.
  - **Returns** – a `double` array of length n containing the approximate solution
  - **Throws**
    * `com.imsl.math.NonlinLeastSquares.TooManyIterationsException` – is thrown if the number of iterations exceeds MaxIterations. MaxIterations is set to 100 by default.

### Example 1: Nonlinear least-squares problem

A nonlinear least-squares problem is solved using a finite-difference Jacobian.

```
import com.imsl.math.*;

public class NonlinLeastSquaresEx1 {
    public static void main(String args[]) throws
    NonlinLeastSquares.TooManyIterationsException {
        NonlinLeastSquares.Function zsf = new NonlinLeastSquares.Function() {
            public void f(double x[], double f[]) {
                f[0] = 10. * (x[1] - x[0]*x[0]);
                f[1] = 1. - x[0];
            }
        };
```

```
        int m = 2;
        int n = 2;
        double xguess[] = {-1.2, 1.};
        double xscale[] = {1., 1.};
        double fscale[] = {1., 1.};
        double x[] = new double[2];
        NonlinLeastSquares zs = new NonlinLeastSquares(m,n);
        zs.setGuess(xguess);
        zs.setXscale(xscale);
        zs.setFscale(fscale);
        x = zs.solve(zsf);

        for (int k = 0;  k < n;  k++) {
            System.out.println("x["+k+"] = "+x[k]);
        }
    }
}
```

### Output

```
x[0] = 1.0
x[1] = 1.0
```

### Example 2: Nonlinear least-squares problem

A nonlinear least-squares problem is solved using a user-supplied Jacobian.

```
import com.imsl.math.*;

public class NonlinLeastSquaresEx2 {
    public static void main(String args[]) throws
    NonlinLeastSquares.TooManyIterationsException {

        NonlinLeastSquares.Jacobian zsj = new NonlinLeastSquares.Jacobian() {
            public void f(double x[], double f[]) {
                f[0] = 10. * (x[1] - x[0]*x[0]);
                f[1] = 1. - x[0];
            }
            public void jacobian(double x[], double fjac[][]) {
                fjac[0][0] = -20.*x[0];
                fjac[1][0] = 10.;
```

```
                fjac[0][1] = -1.;
                fjac[1][1] = 0.;
            }
        };

        int m = 2;
        int n = 2;
        double xguess[] = {-1.2, 1.};
        double xscale[] = {1., 1.};
        double fscale[] = {1., 1.};
        double x[] = new double[2];
        NonlinLeastSquares zs = new NonlinLeastSquares(m,n);
        zs.setGuess(xguess);
        zs.setXscale(xscale);
        zs.setFscale(fscale);
        x = zs.solve(zsj);

        for (int k = 0;  k < n;  k++) {
            System.out.println("x["+k+"] = "+x[k]);
        }
    }
}
```

### Output

```
x[0] = 1.0
x[1] = 1.0
```

## *class* **LinearProgramming**

Linear programming problem using the revised simplex algorithm.

Class `LinearProgramming` uses a revised simplex method to solve linear programming problems, i.e., problems of the form

$$\min_{x \in R^n} c^T x$$

subject to

$$b_l \leq A_x \leq b_u$$

$$x_l \leq x \leq x_u$$

where $c$ is the objective coefficient vector, $A$ is the coefficient matrix, and the vectors $b_l$, $b_u$, $x_l$, and $x_u$ are the lower and upper bounds on the constraints and the variables, respectively.

For a complete description of the revised simplex method, see Murtagh (1981) or Murty (1983).

## Declaration

public class com.imsl.math.LinearProgramming
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* **LinearProgramming.WrongConstraintTypeException**

#### Declaration

public static class com.imsl.math.LinearProgramming.WrongConstraintTypeException
**extends** com.imsl.IMSLException (page 1240)

#### Deprecated

The values for the type of constraint must be either 0, 1 or 2.

#### Constructors

- *LinearProgramming.WrongConstraintTypeException*
  `public` **LinearProgramming.WrongConstraintTypeException**`(`
  `java.lang.String` **message** `)`
- *LinearProgramming.WrongConstraintTypeException*
  `public` **LinearProgramming.WrongConstraintTypeException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* LinearProgramming.BoundsInconsistentException

The bounds given are inconsistent.

#### Declaration

public static class com.imsl.math.LinearProgramming.BoundsInconsistentException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *LinearProgramming.BoundsInconsistentException*
  `public` **LinearProgramming.BoundsInconsistentException**`(`
  `java.lang.String` **message** `)`
- *LinearProgramming.BoundsInconsistentException*
  `public` **LinearProgramming.BoundsInconsistentException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* LinearProgramming.NumericDifficultyException

Numerical difficulty occurred. (Moved to a vertex that is poorly condidtioned).

#### Declaration

public static class com.imsl.math.LinearProgramming.NumericDifficultyException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *LinearProgramming.NumericDifficultyException*
  `public` **LinearProgramming.NumericDifficultyException**`(`
  `java.lang.String` **message** `)`
- *LinearProgramming.NumericDifficultyException*
  `public` **LinearProgramming.NumericDifficultyException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **LinearProgramming.ProblemInfeasibleException**

The problem is not feasible. The constraints are inconsistent.

#### Declaration

public static class com.imsl.math.LinearProgramming.ProblemInfeasibleException
**extends** com.imsl.math.LinearProgramming.NumericDifficultyException (page 171)

#### Constructors

---

- *LinearProgramming.ProblemInfeasibleException*
  `public` **LinearProgramming.ProblemInfeasibleException**`( )`
- *LinearProgramming.ProblemInfeasibleException*
  `public` **LinearProgramming.ProblemInfeasibleException**`(`
  `java.lang.String` **message** `)`

### *class* **LinearProgramming.ProblemUnboundedException**

The problem is unbounded.

#### Declaration

public static class com.imsl.math.LinearProgramming.ProblemUnboundedException
**extends** com.imsl.math.LinearProgramming.NumericDifficultyException (page 171)

#### Constructors

---

- *LinearProgramming.ProblemUnboundedException*
  `public` **LinearProgramming.ProblemUnboundedException**`( )`
- *LinearProgramming.ProblemUnboundedException*
  `public` **LinearProgramming.ProblemUnboundedException**`(`
  `java.lang.String` **message** `)`

### Constructor

- *LinearProgramming*
  `public` **LinearProgramming**`( double[][]` **a**`, double[]` **b**`, double[]` **c** `)`
  - **Description**
    Constructor variables of type `double`.
  - **Parameters**
    * `a` – A `double` matrix with coefficients of the constraints
    * `b` – A `double` array containing the right-hand side of the constraints.
    * `c` – A `double` array containing the coefficients of the objective function.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the dimensions of `a`, `b.length`, and `c.length` are not consistent.

### Methods

- *clone*
  `public java.lang.Object` **clone**`( )`
  - **Description**
    Creates and returns a copy of this object.

- *getDualSolution*
  `public double[]` **getDualSolution**`( )`
  - **Description**
    Returns the dual solution.
  - **Returns** – a `double` array containing the dual solution of the linear programming problem.

- *getOptimalValue*
  `public double` **getOptimalValue**`( )`
  - **Description**
    Returns the optimal value of the objective function.
  - **Returns** – a `double` scalar containing the optimal value of the objective function.

- *getPrimalSolution*
  `public double[]` **getPrimalSolution**`( )`

– **Description**
Returns the solution x of the linear programming problem.

– **Returns** – a `double` array containing the solution x of the linear programming problem.

---

- *setConstraintType*
`public void` **setConstraintType**`( int[]` **constraintType** `)`

  – **Description**
  Sets the types of general constraints in the matrix a.

  – **Parameters**
  * `constraintType` – a `int` array containing the types of general constraints.

| constraintType | Constraint |
|---|---|
| 0 | $r_i = b_i$ |
| 1 | $r_i \leq bu_i$ |
| 2 | $r_i \geq b_i$ |
| 3 | $b_i \leq r_i \leq bu_i$ |

---

- *setLowerBound*
`public void` **setLowerBound**`( double[]` **lowerBound** `)`

  – **Description**
  Sets the lower bound on the variables. If there is no lower bound on a variable, then 1.0e30 should be set as the lower bound.

  – **Parameters**
  * `lowerBound` – a `double` array containing the lower bound on the variables.

---

- *setMaximumIteration*
`public void` **setMaximumIteration**`( int` **iterations** `)`

  – **Description**
  Sets the maximum number of iterations. Default is set to 10000.

  – **Parameters**
  * `iterations` – a `int` scalar specifying the maximum number of iterations.

---

- *setUpperBound*
`public void` **setUpperBound**`( double[]` **upperBound** `)`

  – **Description**
  Sets the upper bound on the variables. If there is no upper bound on a variable, then -1.0e30 should be set as the upper bound.

  – **Parameters**

* `upperBound` – a `double` array containing the upper bound on the variables.

---

- *setUpperLimit*
  `public void` **setUpperLimit**`( double[]` **upperLimit** `)`
  - **Description**
    Sets the upper limit of the constraints.
  - **Parameters**
    * `upperLimit` – a `double` array containing the upper limit of the constraints that have both the lower and the upper bounds.

---

- *solve*
  `public final void` **solve**`( ) throws`
  `com.imsl.math.LinearProgramming.BoundsInconsistentException,`
  `com.imsl.math.LinearProgramming.NumericDifficultyException,`
  `com.imsl.math.LinearProgramming.ProblemInfeasibleException,`
  `com.imsl.math.LinearProgramming.ProblemUnboundedException,`
  `com.imsl.math.SingularMatrixException`
  - **Description**
    Solves the program using the revised simplex algorithm.
  - **Throws**
    * `com.imsl.math.LinearProgramming.BoundsInconsistentException` – is thrown if the bounds are inconsistent.
    * `com.imsl.math.LinearProgramming.ProblemInfeasibleException` – is thrown if there is no feasible solution to the problem.
    * `com.imsl.math.LinearProgramming.ProblemUnboundedException` – is thrown if there is no finite solution to the problem.
    * `com.imsl.math.LinearProgramming.NumericDifficultyException` – is thrown if there is a numerical problem during the solution.

### Example 1: Linear Programming

The linear programming problem in the standard form

$$\min\ f(x) = -x_1 - 3x_2$$

subject to:

$x_1 + x_2 + x_3 = 1.5$
$x_1 + x_2 - x_4 = 0.5$
$x_1 + x_5 = 1.0$

$x_2 + x_6 = 1.0$
$x_i \geq 0, \quad \text{for } i = 1, \ldots, 6$

is solved.

```
import com.imsl.math.*;

public class LinearProgrammingEx1 {
    public static void main(String args[]) throws Exception {
        double[][] a = {
            {1.0, 1.0, 1.0, 0.0, 0.0, 0.0},
            {1.0, 1.0, 0.0, -1.0, 0.0, 0.0},
            {1.0, 0.0, 0.0, 0.0, 1.0, 0.0},
            {0.0, 1.0, 0.0, 0.0, 0.0, 1.0}
        };
        double[] b = {1.5, 0.5, 1.0, 1.0};
        double[] c = {-1.0, -3.0, 0.0, 0.0, 0.0, 0.0};

        LinearProgramming zf = new LinearProgramming(a, b, c);

        zf.solve();
        new PrintMatrix("Solution").print(zf.getPrimalSolution());
    }
}
```

## Output

```
Solution
    0
0  0.5
1  1
2  0
3  1
4  0.5
5  0
```

### Example 2: Linear Programming

The linear programming problem

$$\min\ f(x) = -x_1 - 3x_2$$

subject to:

$0.5 \le x_1 + x_2 \le 1.5$
$0 \le x_1 \le 1.0$
$0 \le x_2 \le 1.0$

is solved.

```
import com.imsl.math.*;

public class LinearProgrammingEx2 {
    public static void main(String args[]) throws Exception {
        int[] constraintType = {3};
        double[] upperBound = {1.0, 1.0};
        double[][] a = {{1.0, 1.0}};
        double[] b = {0.5};
        double[] upperLimit = {1.5};
        double[] c = {-1.0, -3.0};

        LinearProgramming zf = new LinearProgramming(a, b, c);

        zf.setUpperLimit(upperLimit);
        zf.setConstraintType(constraintType);
        zf.setUpperBound(upperBound);
        zf.solve();
        new PrintMatrix("Solution").print(zf.getPrimalSolution());
        new PrintMatrix("Dual Solution").print(zf.getDualSolution());
        System.out.println("Optimal Value = " + zf.getOptimalValue());
    }
}
```

### Output

```
Solution
    0
0  0.5
1  1
```

```
Dual Solution
   0
0  -1

Optimal Value = -3.5
```

## *class* QuadraticProgramming

Solves the convex quadratic programming problem subject to equality or inequality constraints.

Class `QuadraticProgramming` is based on M.J.D. Powell's implementation of the Goldfarb and Idnani dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints (Goldfarb and Idnani 1983); i.e., problems of the form

$$\min_{x \in R^n} g^T x + \frac{1}{2} x^T H x$$

subject to

$$A_1 x = b_1$$

$$A_2 x \geq b_2$$

given the vectors $b_1$, $b_2$, and $g$, and the matrices $H$, $A_1$, and $A_2$. $H$ is required to be positive definite. In this case, a unique $x$ solves the problem or the constraints are inconsistent. If $H$ is not positive definite, a positive definite perturbation of $H$ is used in place of $H$. For more details, see Powell (1983, 1985).

If a perturbation of $H$, $H + \alpha I$, is used in the $QP$ problem, then $H + \alpha I$ also should be used in the definition of the Lagrange multipliers.

## Declaration

public class com.imsl.math.QuadraticProgramming
**extends** java.lang.Object

## Inner Class

### *class* QuadraticProgramming.InconsistentSystemException

Inconsistent system.

#### Declaration

public static class com.imsl.math.QuadraticProgramming.InconsistentSystemException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *QuadraticProgramming.InconsistentSystemException*
  `public` **QuadraticProgramming.InconsistentSystemException**`( )`

## Field

---

- public static final `double` **EPSILON_SMALL**
  - The smallest relative spacing for doubles.

## Constructor

---

- *QuadraticProgramming*
  `public` **QuadraticProgramming**`( double[][]` **h**`, double[]` **g**`, double[][]` **aEquality**`, double[]` **bEquality**`, double[][]` **aInequality**`, double[]` **bInequality** `) throws com.imsl.math.QuadraticProgramming.InconsistentSystemException`
  - **Description**
    Solve a quadratic programming problem.

- **Parameters**
  - `h` – is square array containing the Hessian. It must be positive definite.
  - `g` – contains the coefficients of the linear term of the objective function.
  - `aEquality` – is a rectangular matrix containing the equality constraints. It can be null if there are no equality constraints.
  - `bEquality` – contains the right-side of the equality constraints. It can be null if there are no equality constraints.
  - `aInequality` – is a rectangular matrix containing the inequality constraints. It can be null if there are no inequality constraints.
  - `bInequality` – contains the right-side of the inequality constraints. It can be null if there are no inequality constraints.

## Methods

---

- *getDual*
  `public double[]` **getDual**`( )`
  - **Description**
    Returns the dual (Lagrange multipliers).

---

- *getSolution*
  `public double[]` **getSolution**`( )`
  - **Description**
    Returns the solution.

---

- *isNoMoreProgress*
  `public boolean` **isNoMoreProgress**`( )`
  - **Description**
    Returns true if due to computer rounding error, a change in the variables fail to improve the objective function. Usually the solution is close to optimum.

## Example 1: Solve a Quadratic Programming Problem

The quadratic programming problem is to minimize

$$x_0^2 + x_1^2 + x_2^2 + x_3^2 + x_4^2 - 2x_1x_2 - 2x_3x_4 - 2x_0$$

subject to

$$x_0 + x_1 + x_2 + x_3 + x_4 = 5$$

$$x_2 - 2x_3 - 2x_4 = -3$$

```
import com.imsl.math.*;

public class QuadraticProgrammingEx1 {
    public static void main(String args[]) throws
    QuadraticProgramming.InconsistentSystemException {
        double h[][] = {
            {2, 0, 0, 0, 0},
            {0, 2,-2, 0, 0},
            {0,-2, 2, 0, 0},
            {0, 0, 0, 2,-2},
            {0, 0, 0,-2, 2},
        };
        double aeq[][] = {
            { 1, 1, 1, 1, 1},
            { 0, 0, 1,-2,-2}
        };
        double beq[] = {5, -3};
        double g[] = {-2, 0, 0, 0, 0};

        QuadraticProgramming qp =
        new QuadraticProgramming(h, g, aeq, beq, null, null);

        // Print the solution and its dual
        new PrintMatrix("x").print(qp.getSolution());
        new PrintMatrix("dual").print(qp.getDual());
    }
}
```

## Output

```
  x
   0
0  1
1  1
2  1
```

```
3  1
4  1

 dual
   0
0   0
1  -0
2   0
3   0
4   0
```

### Example 2: Solve a Quadratic Programming Problem

The quadratic programming problem is to minimize

$$x_0^2 + x_1^2 + x_2^2$$

subject to

$$x_0 + 2x_1 - x_2 = 4$$

$$x_0 - x_1 + x_2 = -2$$

```
import com.imsl.math.*;

public class QuadraticProgrammingEx2 {
    public static void main(String args[]) throws
    QuadraticProgramming.InconsistentSystemException {
        double h[][] = {
            {2, 0, 0},
            {0, 2, 0},
            {0, 0, 2}
        };
        double aeq[][] = {{1, 2,-1}, {1,-1, 1}};
        double beq[] = {4, -2};
        double g[] = {0, 0, 0};

        QuadraticProgramming qp =
```

```
        new QuadraticProgramming(h, g, aeq, beq, null, null);

        // Print the solution and its dual
        new PrintMatrix("x").print(qp.getSolution());
        new PrintMatrix("dual").print(qp.getDual());
    }
}
```

**Output**

```
     x
     0
0   0.286
1   1.429
2  -0.857

   dual
     0
0   1.143
1  -0.571
2   0
```

## *class* MinConGenLin

Minimizes a general objective function subject to linear equality/inequality constraints.

The class `MinConGenLin` is based on M.J.D. Powell's TOLMIN, which solves linearly constrained optimization problems, i.e., problems of the form

$$\min f(x)$$

subject to

$$A_1 x = b_1$$

$$A_2 x \le b_2$$

$$x_l \le x \le x_u$$

given the vectors $b_1$, $b_2$, $x_l$, and $x_u$ and the matrices $A_1$ and $A_2$.

The algorithm starts by checking the equality constraints for inconsistency and redundancy. If the equality constraints are consistent, the method will revise $x^0$, the initial guess, to satisfy

$$A_1 x = b_1$$

Next, $x^0$ is adjusted to satisfy the simple bounds and inequality constraints. This is done by solving a sequence of quadratic programming subproblems to minimize the sum of the constraint or bound violations.

Now, for each iteration with a feasible $x^k$, let $J_k$ be the set of indices of inequality constraints that have small residuals. Here, the simple bounds are treated as inequality constraints. Let $I_k$ be the set of indices of active constraints. The following quadratic programming problem

$$\min f\left(x^k\right) + d^T \nabla f\left(x^k\right) + \frac{1}{2} d^T B^k d$$

subject to

$$a_j d = 0,\ j \in I_k$$

$$a_j d \le 0,\ j \in J_k$$

is solved to get $(d^k, \lambda^k)$ where $a_j$ is a row vector representing either a constraint in $A_1$ or $A_2$ or a bound constraint on $x$. In the latter case, the $a_j = e_j$ for the bound constraint $x_i \le (x_u)_i$ and $a_j = -e_i$ for the constraint $-x_i \le (x_l)_i$. Here, $e_i$ is a vector with 1 as the $i$-th component, and zeros elsewhere. Variables $\lambda^k$ are the Lagrange multipliers, and $B^k$ is a positive definite approximation to the second derivative $\nabla^2 f(x^k)$.

After the search direction $d^k$ is obtained, a line search is performed to locate a better point. The new point $x^{k+1} = x^k + \alpha^k d^k$ has to satisfy the conditions

$$f(x^k + \alpha^k d^k) \le f(x^k) + 0.1\alpha^k (d^k)^T \nabla f(x^k)$$

and

$$(d^k)^T \nabla f(x^k + \alpha^k d^k) \ge 0.7 (d^k)^T \nabla f(x^k)$$

The main idea in forming the set $J_k$ is that, if any of the equality constraints restricts the step-length $\alpha^k$, then its index is not in $J_k$. Therefore, small steps are likely to be avoided.

Finally, the second derivative approximation $B^K$, is updated by the BFGS formula, if the condition

$$\left(d^K\right)^T \nabla f\left(x^k + \alpha^k d^k\right) - \nabla f\left(x^k\right) > 0$$

holds. Let $x^k \leftarrow x^{k+1}$, and start another iteration.

The iteration repeats until the stopping criterion

$$\left\|\nabla f(x^k) - A^k \lambda^K\right\|_2 \leq \tau$$

is satisfied. Here $\tau$ is the supplied tolerance. For more details, see Powell (1988, 1989).

## Declaration

```
public class com.imsl.math.MinConGenLin
extends java.lang.Object
implements java.io.Serializable, java.lang.Cloneable
```

## Inner Classes

### *interface* **MinConGenLin.Function**

Public interface for the user-supplied function to evaluate the function to be minimized.

#### Declaration

public static interface com.imsl.math.MinConGenLin.Function

#### Method

- *f*
  `double f( double[] x )`
  - **Description**
    Public interface for the function to be minimized.

– **Parameters**
  * `x` – a `double` array, the point at which the function is evaluated. `x.length` equals the number of variables.
– **Returns** – a `double` scalar, the function value at `x`

### *interface* **MinConGenLin.Gradient**

Public interface for the user-supplied function to compute the gradient.

#### Declaration

public static interface com.imsl.math.MinConGenLin.Gradient
**implements** MinConGenLin.Function

#### Method

---

- *gradient*
  `void` **gradient**`( double[] x, double[] g )`
  – **Description**
    Public interface for the user-supplied function to compute the gradient at point `x`.
  – **Parameters**
    * `x` – a `double` array, the point at which the gradient is evaluated. `x.length` equals the number of variables.
    * `g` – a `double` array, the values of the gradient of the objective function.

### *class* **MinConGenLin.ConstraintsInconsistentException**

The equality constraints are inconsistent.

#### Declaration

public static class com.imsl.math.MinConGenLin.ConstraintsInconsistentException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConGenLin.ConstraintsInconsistentException*
  `public` **MinConGenLin.ConstraintsInconsistentException**`(`
  `java.lang.String` **message** `)`
- *MinConGenLin.ConstraintsInconsistentException*
  `public` **MinConGenLin.ConstraintsInconsistentException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* MinConGenLin.VarBoundsInconsistentException

The equality constraints and the bounds on the variables are found to be inconsistent.

### Declaration

public static class com.imsl.math.MinConGenLin.VarBoundsInconsistentException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConGenLin.VarBoundsInconsistentException*
  `public` **MinConGenLin.VarBoundsInconsistentException**`(`
  `java.lang.String` **message** `)`
- *MinConGenLin.VarBoundsInconsistentException*
  `public` **MinConGenLin.VarBoundsInconsistentException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* MinConGenLin.ConstraintsNotSatisfiedException

No vector $x$ satisfies all of the constraints.

### Declaration

public static class com.imsl.math.MinConGenLin.ConstraintsNotSatisfiedException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConGenLin.ConstraintsNotSatisfiedException*
  `public` **MinConGenLin.ConstraintsNotSatisfiedException**`(`
  `java.lang.String` **message** `)`
- *MinConGenLin.ConstraintsNotSatisfiedException*
  `public` **MinConGenLin.ConstraintsNotSatisfiedException**`(`
  `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

## *class* MinConGenLin.EqualityConstraintsException

the variables are determined by the equality constraints.

### Declaration

public static class com.imsl.math.MinConGenLin.EqualityConstraintsException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConGenLin.EqualityConstraintsException*
  `public` **MinConGenLin.EqualityConstraintsException**`(` `java.lang.String` **message** `)`
- *MinConGenLin.EqualityConstraintsException*
  `public` **MinConGenLin.EqualityConstraintsException**`(` `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

## Constructor

---

- *MinConGenLin*
  `public` **MinConGenLin**`(` `MinConGenLin.Function` **fcn**`,` `int` **nvar**`,` `int` **ncon**`,` `int` **neq**`,` `double[]` **a**`,` `double[]` **b**`,` `double[]` **lowerBound**`,` `double[]` **upperBound** `)`
  - **Description**
    Constructor for `MinConGenLin`.

- **Parameters**
  - fcn – A Function object, user-supplied function to evaluate the function to be minimized.
  - nvar – A int scalar containing the number of variables.
  - ncon – A int scalar containing the number of linear constraints (excluding simple bounds).
  - neq – A int scalar containing the number of linear equality constraints.
  - a – A double array containing the equality constraint gradients in the first neq rows followed by the inequality constraint gradients. a.length = ncon * nvar
  - b – A double array containing the right-hand sides of the linear constraints.
  - lowerBound – A double array containing the lower bounds on the variables. Choose a very large negative value if a component should be unbounded below or set lowerBound[i] = upperBound[i] to freeze the $i$-th variable. lowerBound.length = nvar
  - upperBound – A double array containing the upper bounds on the variables. Choose a very large positive value if a component should be unbounded above. upperBound.length = nvar
- **Throws**
  - java.lang.IllegalArgumentException – is thrown if the dimensions of nvar, ncon, neq, a.length , b.length, lowerBound.length and upperBound.length are not consistent.

## Methods

---

- *getFinalActiveConstraints*
  public int[] **getFinalActiveConstraints**( )
  - **Description**
    Returns the indices of the final active constraints.
  - **Returns** – a int array containing the indices of the final active constraints.

---

- *getFinalActiveConstraintsNum*
  public int **getFinalActiveConstraintsNum**( )
  - **Description**
    Returns the final number of active constraints.
  - **Returns** – a int scalar containing the final number of active constraints.

---

- *getLagrangeMultiplerEst*
  public double[] **getLagrangeMultiplerEst**( )

**Deprecated**

Method name misspelled. Replaced by method getLagrangeMultiplierEst. Returns the Lagrange multiplier estimates of the final active constraints.

- **Returns** – a `double` array containing the Lagrange multiplier estimates of the final active constraints.

---

- *getLagrangeMultiplierEst*
  `public double[]` **getLagrangeMultiplierEst**`( )`
  - **Description**
    Returns the Lagrange multiplier estimates of the final active constraints.
  - **Returns** – a `double` array containing the Lagrange multiplier estimates of the final active constraints.

---

- *getObjectiveValue*
  `public double` **getObjectiveValue**`( )`
  - **Description**
    Returns the value of the objective function.
  - **Returns** – a `double` scalar containing the value of the objective function.

---

- *getSolution*
  `public double[]` **getSolution**`( )`
  - **Description**
    Returns the computed solution.
  - **Returns** – a `double` array containing the computed solution.

---

- *setGuess*
  `public void` **setGuess**`( double[]` **guess** `)`
  - **Description**
    Sets an initial guess of the solution.
  - **Parameters**
    * `guess` – a `double` array containing an initial guess.

---

- *setTolerance*
  `public void` **setTolerance**`( double` **tolerance** `)`
  - **Description**
    Sets the nonnegative tolerance on the first order conditions at the calculated solution.
  - **Parameters**

* tolerance – a double scalar containing the tolerance.

---

- *solve*

  `public final void solve( ) throws com.imsl.math.MinConGenLin.ConstraintsInconsistentException, com.imsl.math.MinConGenLin.VarBoundsInconsistentException, com.imsl.math.MinConGenLin.ConstraintsNotSatisfiedException, com.imsl.math.MinConGenLin.EqualityConstraintsException`

  - **Description**

    Minimizes a general objective function subject to linear equality/inequality constraints.

### Example 1: Linear Constrained Optimization

The problem

$$\min\ f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2 - 2x_2x_3 - 2x_4x_5 - 2x_1$$

subject to

$$x_1 + x_2 + x_3 + x_4 + x_5 = 5$$

$$x_3 - 2x_4 - 2x_5 = -3$$

$$0 \leq x \leq 10$$

is solved.

```
import com.imsl.math.*;

public class MinConGenLinEx1 {
    public static void main(String args[]) throws Exception {
        int neq = 2;
        int ncon = 2;
        int nvar = 5;
        double a[] = {1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, -2.0, -2.0};
        double b[] = {5.0, -3.0};
        double xlb[] = {0.0, 0.0, 0.0, 0.0, 0.0};
        double xub[] = {10.0, 10.0, 10.0, 10.0, 10.0};
```

```
        MinConGenLin.Function fcn = new MinConGenLin.Function() {
            public double f(double[] x) {
                return x[0]*x[0] + x[1]*x[1] + x[2]*x[2] + x[3]*x[3] +
                x[4]*x[4] - 2.0*x[1]*x[2] - 2.0*x[3] * x[4] - 2.0*x[0];
            }
        };

        MinConGenLin zf =
        new MinConGenLin(fcn, nvar, ncon, neq, a, b, xlb, xub);

        zf.solve();
        new PrintMatrix("Solution").print(zf.getSolution());
    }
}
```

### Output

```
Solution
   0
0  1
1  1
2  1
3  1
4  1
```

### Example 2: Linear Constrained Optimization

The problem

$$\min\ f(x) = -x_0 x_1 x_2$$

subject to

$$-x_0 - 2x_1 - 2x_2 \le 0$$

$$x_0 + 2x_1 + 2x_2 \le 72$$

$$0 \le x_0 \le 20$$

$$0 \le x_1 \le 11$$

$$0 \le x_2 \le 42$$

is solved with an initial guess of $x_0 = 10$, $x_1 = 10$ and $x_2 = 10$.

```
import com.imsl.math.*;

public class MinConGenLinEx2 {

    public static void main(String args[]) throws Exception {
        int neq = 0;
        int ncon = 2;
        int nvar = 3;
        double a[] = {-1.0, -2.0, -2.0, 1.0, 2.0, 2.0};
        double xlb[] = {0.0, 0.0, 0.0};
        double xub[] = {20.0, 11.0, 42.0};
        double xguess[] = {10.0, 10.0, 10.0};
        double b[] = {0.0, 72.0};

        MinConGenLin.Gradient grad = new MinConGenLin.Gradient() {
            public double f(double[] x) {
                return -x[0] * x[1] * x[2];
            }
            public void gradient(double[] x, double[] g) {
                g[0] = -x[1]*x[2];
                g[1] = -x[0]*x[2];
                g[2] = -x[0]*x[1];
            }
        };
```

```
        MinConGenLin zf =
        new MinConGenLin(grad, nvar, ncon, neq, a, b, xlb, xub);

        zf.setGuess(xguess);
        zf.solve();
        new PrintMatrix("Solution").print(zf.getSolution());
        System.out.println("Objective value = " + zf.getObjectiveValue());
    }
}
```

### Output

```
Solution
   0
0  20
1  11
2  15

Objective value = -3300.0
```

## *class* BoundedLeastSquares

Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.

Class `BoundedLeastSquares` uses a modified Levenberg-Marquardt method and an active set strategy to solve nonlinear least-squares problems subject to simple bounds on the variables. The problem is stated as follows:

$$min\frac{1}{2}F\left(x\right)^{T}F\left(x\right)=\frac{1}{2}\sum_{i=1}^{m}f_{i}\left(x\right)^{2}$$

subject to

$$l\leq x\leq u$$

where m $\geq$ n, $F:\mathrm{R}^{n}\rightarrow\mathrm{R}^{m}$, and $f_i(x)$ is the $i$-th component function of *F(x)*. From a given starting point, an active set `IA`, which contains the indices of the variables at their bounds, is built. A variable is called a "free variable" if it is not in the active set. The routine then computes the search direction for the free variables according to the formula

$$d = -\left(J^T J + \mu I\right)^{-1} J^T F$$

where $\mu$ is the Levenberg-Marquardt parameter, $F = F(x)$, and $J$ is the Jacobian with respect to the free variables. The search direction for the variables in `IA` is set to zero. The trust region approach discussed by Dennis and Schnabel (1983) is used to find the new point. Finally, the optimality conditions are checked. The conditions are:

$$\|g(x_i)\| \le \varepsilon, l_i < x_i < u_i$$

$$g(x_i) < 0, x_i = u_i$$

$$g(x_i) > 0, x_i = l_i$$

where $\varepsilon$ is a gradient tolerance. This process is repeated until the optimality criterion is achieved.

The active set is changed only when a free variable hits its bounds during an iteration or the optimality condition is met for the free variables but not for all variables in `IA`, the active set. In the latter case, a variable that violates the optimality condition will be dropped out of `IA`. For more details on the Levenberg-Marquardt method, see Levenberg (1944) or Marquardt (1963). For more detail on the active set strategy, see Gill and Murray (1976).

## Declaration

public class com.imsl.math.BoundedLeastSquares
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *interface* BoundedLeastSquares.Function

Public interface for the user-supplied function to evaluate the function that defines the least-squares problem.

#### Declaration

public static interface com.imsl.math.BoundedLeastSquares.Function

### Method

---

- *compute*
  `void` **compute**`( double[] x, double[]` **f** `)`
  - **Description**
    Public interface for the user-supplied function to evaluate the function that defines the least-squares problem.
  - **Parameters**
    * `x` – a `double` array containing the point at which the function is to evaluated. `x.length = nVariables`
    * `f` – a `double` array which contains the function values at point `x`. `f.length = mFunctions`

## *interface* BoundedLeastSquares.Jacobian

Public interface for the user-supplied function to compute the Jacobian.

### Declaration

public static interface com.imsl.math.BoundedLeastSquares.Jacobian

### Method

---

- *compute*
  `void` **compute**`( double[] x, double[]` **fjac** `)`
  - **Description**
    Public interface for the user-supplied function to compute the Jacobian.
  - **Parameters**
    * `x` – a `double` array, the point at which the Jacobian is to evaluated. `x.length = nVariables`
    * `fjac` – a `double` array, the computed Jacobian at the point x. point `x`. `fjac.length = mFunctions x nVariables`

## *class* BoundedLeastSquares.FalseConvergenceException

False convergence - The iterates appear to be converging to a noncritical point.

### Declaration

public static class com.imsl.math.BoundedLeastSquares.FalseConvergenceException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *BoundedLeastSquares.FalseConvergenceException*
  `public` **BoundedLeastSquares.FalseConvergenceException**`(`
  `java.lang.String` **message** `)`

  – **Description**
    Constructs an `FalseConvergenceException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  – **Parameters**
    * `message` – the detail message

---

- *BoundedLeastSquares.FalseConvergenceException*
  `public` **BoundedLeastSquares.FalseConvergenceException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

  – **Description**
    Constructs an `FalseConvergenceException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  – **Parameters**
    * `key` – the key of the error message in the resource bundle
    * `arguments` – an array containing arguments used within the error message string

### Constructor

---

- *BoundedLeastSquares*
  `public` **BoundedLeastSquares**`( BoundedLeastSquares.Function` **function**`,`
  `int` **mFunctions**`, int` **nVariables**`, int` **boundType**`, double[]`
  **lowerBound**`, double[]` **upperBound** `)`

  – **Description**
    Constructor for `BoundedLeastSquares`.
  – **Parameters**

- * `function` – a `Function` object, user-supplied function to evaluate the function
  * `mFunctions` – a `int` scalar containing the number of functions
  * `nVariables` – a `int` scalar containing the number of variables
  * `boundType` – a `int` scalar containing the types of bounds on the variable

| **boundType** | **Action** |
|---|---|
| 0 | User will supply all the bounds. |
| 1 | All variables are nonnegative. |
| 2 | All variables are nonpositive. |
| 3 | User supplies only the bounds on first variable, all other variables will have the same bounds. |

  * `lowerBound` – a `double` array containing the lower bounds on the variables
  * `upperBound` – a `double` array containing the upper bounds on the variables
- **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if the dimensions of `mFunctions`, `nVariables`, `boundType`, `lowerBound.length` and `upperBound.length` are not consistent

## Methods

---

- *getJacobian*
  `public double[][]` **getJacobian**`( )`
  - **Description**
    Returns the Jacobian at the approximate solution.
  - **Returns** – a `mFunctions x nVariables double` matrix containing the Jacobian at the approximate solution

---

- *getResiduals*
  `public double[]` **getResiduals**`( )`
  - **Description**
    Returns the residuals at the approximate solution.
  - **Returns** – a `double` array containing the residuals at the approximate solution

---

- *getSolution*
  `public double[]` **getSolution**`( )`
  - **Description**
    Returns the solution.

– **Returns** – a `double` array containing the computed solution

---

- *setAbsoluteFcnTol*
  `public void` **setAbsoluteFcnTol**`( double` **absoluteFcnTol** `)`

  – **Description**
  Sets the absolute function tolerance. If this member function is not called, a value of Math.max(1.0e-10, Math.pow(2.2204460492503131e-16, 2.0/3.0)), is used.

  – **Parameters**
    * `absoluteFcnTol` – a `double` scalar containing the absolute function tolerance

---

- *setDiagonalScalingMatrix*
  `public void` **setDiagonalScalingMatrix**`( double[]` **diagonalScalingMatrix** `)`

  – **Description**
  Sets the diagonal scaling matrix for the functions. The i-th component of the array is a positive scalar specifying the reciprocal magnitude of the i-th component function of the problem. If this member function is not called, an initial scaling of 1.0 is used.

  – **Parameters**
    * `diagonalScalingMatrix` – a `double` array containing the diagonal scaling for the functions

---

- *setGoodDigit*
  `public void` **setGoodDigit**`( int` **goodDigit** `)`

  – **Description**
  Sets the number of good digits in the function. If this member function is not called, a value of (int)(-Sfun.log10(2.2204460492503131e-16) + 0.1e0) is used.

  – **Parameters**
    * `goodDigit` – a `int` scalar containing the number of good digits

---

- *setGradientTol*
  `public void` **setGradientTol**`( double` **gradientTol** `)`

  – **Description**
  Sets the scaled gradient tolerance. If this member function is not called, a value of Math.pow(2.2204460492503131e-16, 1.0e0/3.0e0) is used.

  – **Parameters**
    * `gradientTol` – a `double` scalar containing the scaled gradient tolerance

---

- *setGuess*
  `public void` **setGuess**`( double[]` **guess** `)`
  - **Description**
    Sets the initial guess of the solution. If this member function is not called, an initial scaling of 1.0 is used.
  - **Parameters**
    * `guess` – a `double` array containing an initial guess

---

- *setInternalScale*
  `public void` **setInternalScale**`( )`
  - **Description**
    Sets the internal variable scaling option. With this option, scaling for the variables is set internally.

---

- *setJacobian*
  `public void` **setJacobian**`( BoundedLeastSquares.Jacobian` **jacobian** `)`
  - **Description**
    Sets the Jacobian.
  - **Parameters**
    * `jacobian` – a `Jacobian` object to compute the Jacobian.

---

- *setMaximumFunctionEvals*
  `public void` **setMaximumFunctionEvals**`( int` **evaluations** `)`
  - **Description**
    Sets the maximum number of function evaluations. If this member function is not called, a value of 400 is used.
  - **Parameters**
    * `evaluations` – a `int` scalar containing the maximum number of function evaluations

---

- *setMaximumIteration*
  `public void` **setMaximumIteration**`( int` **iterations** `)`
  - **Description**
    Sets the maximum number of iterations. If this member function is not called, a value of 100 is used.
  - **Parameters**
    * `iterations` – a `int` scalar containing the maximum number of iterations

---

- *setMaximumJacobianEvals*
  `public void` **setMaximumJacobianEvals**`( int` **evaluations** `)`
  - **Description**
    Sets the maximum number of Jacobian evaluations. If this member function is not called, a value of 400 is used.
  - **Parameters**
    * `evaluations` – a `int` scalar containing the maximum number of Jacobian evaluations

---

- *setMaximumStepSize*
  `public void` **setMaximumStepSize**`( double` **stepSize** `)`
  - **Description**
    Sets the maximum allowable step size.
  - **Parameters**
    * `stepSize` – a `double` scalar containing the maximum allowable step size

---

- *setRelativeFcnTol*
  `public void` **setRelativeFcnTol**`( double` **relativeFcnTol** `)`
  - **Description**
    Sets the relative function tolerance. If this member function is not called, a value of Math.pow(2.2204460492503131e-16, 2.0e0/3.0e0) is used.
  - **Parameters**
    * `relativeFcnTol` – a `double` scalar containing the relative function tolerance

---

- *setScaledStepTol*
  `public void` **setScaledStepTol**`( double` **scaledStepTol** `)`
  - **Description**
    Sets the scaled step tolerance. If this member function is not called, a value of Math.max(1.0e-10, Math.pow(2.2204460492503131e-16, 2.0e0/3.0e0) is used.
  - **Parameters**
    * `scaledStepTol` – a `double` scalar containing the scaled step tolerance

---

- *setScalingVector*
  `public void` **setScalingVector**`( double[]` **scalingVector** `)`
  - **Description**
    Sets the scaling vector for the variables. If this member function is not called, an initial scaling of 1.0 is used.
  - **Parameters**

* `scalingVector` – a `double` array containing the scaling vector for the variables

---

- *setTrustRegion*
  `public void` **setTrustRegion**`( double` **trustRegion** `)`
  - **Description**
    Sets the size of initial trust region radius. If this member function is not called, the value is based on the initial scaled Cauchy step.
  - **Parameters**
    * `trustRegion` – a `double` scalar containing the initial trust region radius

---

- *solve*
  `public final void` **solve**`( ) throws`
  `com.imsl.math.BoundedLeastSquares.FalseConvergenceException`
  - **Description**
    Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.

### Example 1: Bounded Least Squares

The nonlinear least-squares problem

$$\min \frac{1}{2} \sum_{i=0}^{1} f_i\left(x\right)^2$$

$$-2 \le x_0 \le 0.5$$

$$-1 \le x_1 \le 2$$

where

$$f_0(x) = 10(x_1 - x_0^2) \text{ and } f_1(x) = (1 - x_0)$$

is solved.

```
import com.imsl.math.*;

public class BoundedLeastSquaresEx1 {
    public static void main(String args[]) throws Exception {
```

```
        int m = 2;
        int n = 2;
        int ibtype = 0;
        double[] xlb = {-2.0, -1.0};
        double[] xub = {0.5, 2.0};

        BoundedLeastSquares.Function rosbck =
        new BoundedLeastSquares.Function() {
            public void compute(double[] x, double[] f) {
                f[0] = 10.0*(x[1] - x[0]*x[0]);
                f[1] = 1.0 - x[0];
            }
        };

        BoundedLeastSquares zf =
        new BoundedLeastSquares(rosbck, m, n, ibtype, xlb, xub);

        zf.solve();
        new PrintMatrix("Solution").print(zf.getSolution());
    }
}
```

## Output

```
Solution
    0
0  0.5
1  0.25
```

## Example 2: Bounded Least Squares

The nonlinear least-squares problem

$$\min \frac{1}{2}\sum_{i=0}^{1} f_i\left(x\right)^2$$

$$-2 \le x_0 \le 0.5$$

$$-1 \le x_1 \le 2$$

where

$$f_0(x) = 10(x_1 - x_0^2) \text{ and } f_1(x) = (1 - x_0)$$

is solved. An initial guess (-1.2, 1.0) is supplied, as well as the analytic Jacobian. The residual at the approximate solution is returned.

```
import com.imsl.math.*;

public class BoundedLeastSquaresEx2 {
    public static void main(String args[]) throws Exception {
        int m = 2;
        int n = 2;
        int ibtype = 0;
        double[] xlb = {-2.0, -1.0};
        double[] xub = {0.5, 2.0};
        double[] xguess = {-1.2, 1.0};

        BoundedLeastSquares.Function rosbck =
        new BoundedLeastSquares.Function() {
            public void compute(double[] x, double[] f) {
                f[0] = 10.0*(x[1] - x[0]*x[0]);
                f[1] = 1.0 - x[0];
            }
        };

        BoundedLeastSquares.Jacobian jacob =
        new BoundedLeastSquares.Jacobian() {
            public void compute(double[] x, double[] fjac) {
                fjac[0] = -20.0*x[0];
                fjac[1] = 10.0;
                fjac[2] = -1.0;
                fjac[3] = 0.0;
            }
        };

        BoundedLeastSquares zf =
        new BoundedLeastSquares(rosbck, m, n, ibtype, xlb, xub);

        zf.setJacobian(jacob);
```

```
        zf.setGuess(xguess);
        zf.solve();
        new PrintMatrix("Solution").print(zf.getSolution());
        new PrintMatrix("Residuals").print(zf.getResiduals());
    }
}
```

### Output

```
Solution
    0
0  0.5
1  0.25

Residuals
    0
0  0
1  0.5
```

## *class* MinConNLP

General nonlinear programming solver.

`MinConNLP` is based on the FORTRAN subroutine, `DONLP2`, by Peter Spellucci and licensed from TU Darmstadt. `MinConNLP` uses a sequential equality constrained quadratic programming method with an active set technique, and an alternative usage of a fully regularized mixed constrained subproblem in case of nonregular constraints (i.e. linear dependent gradients in the “working sets”). It uses a slightly modified version of the Pantoja-Mayne update for the Hessian of the Lagrangian, variable dual scaling and an improved Armjijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. Details may be found in the following two papers:

P. Spellucci: *An SQP method for general nonlinear programs using only equality constrained subproblems.* Math. Prog. 82, (1998), 413-448.

P. Spellucci: *A new technique for inconsistent problems in the SQP method.* Math. Meth.

of Oper. Res. 47, (1998), 355-500. (published by Physica Verlag, Heidelberg, Germany).

The problem is stated as follows:

$$\min_{x \in R^n} f(x)$$

subject to

$$g_j(x) = 0, \text{for } j = 1, \ldots, m_e$$

$$g_j(x) \geq 0, \text{for j} = \text{m}_\text{e} + 1, \ldots, \text{m}$$

$$x_l \leq x \leq x_u$$

where all problem functions are assumed to be continuously differentiable. Although default values are provided for optional input arguments, it may be necessary to adjust these values for some problems. Through the use of member functions, `MinConNLP` allows for several parameters of the algorithm to be adjusted to account for specific characteristics of problems. The provides detailed descriptions of these parameters as well as strategies for maximizing the performance of the algorithm. In addition, the following are a number of guidelines to consider when using `MinConNLP`:

- A good initial starting point is very problem specific and should be provided by the calling program whenever possible. See method `setGuess`.
- Gradient approximation methods can have an effect on the success of `MinConNLP`. Selecting a higher order approximation method may be necessary for some problems. See method `setDifferentiationType`.
- If a two sided constraint $l_i \leq g_i(x) \leq u_i$ is transformed into two constraints, $g_{2i}(x) \geq 0$ and $g_{2i+1}(x) \geq 0$, then choose $del0 < 1/2\,(u_i - l_i)\,/max\,\{1, \|\nabla g_i(x)\|\}$, or at least try to provide an estimate for that value. This will increase the efficiency of the algorithm. See method `setBindingThreshold`.
- The parameter `ierr` provided in the interface to the user supplied function `FCN` can be very useful in cases when evaluation is requested at a point that is not possible or reasonable. For example, if evaluation at the requested point would result in a floating point exception, then setting `ierr` to `true` and returning without performing the evaluation will avoid the exception. `MinConNLP` will then reduce the stepsize and try the step again. Note, if `ierr` is set to `true` for the initial guess, then an error is issued.

Note that one can use the JDK 1.4 JAVA Logging API to generate intermediate output for the solver. Accumulated levels of detail correspond to JAVA's CONFIG, FINE, FINER, and FINEST logging levels with CONFIG yielding the smallest amount of information and FINEST yielding the most. The levels of output yield the following:

| *Level* | *Output* |
|---|---|
| CONFIG | One line of intermediate results is printed with each iteration. A summary report is printed upon completion. |
| FINE | Lines of intermediate results giving the most important data for each step are printed after each step. A summary report is printed upon completion. |
| FINER | Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, etc. are printed. A summary report is printed upon completion. |
| FINEST | Lines of detailed intermediate results showing all primal and dual variables, the relevant values from the working set, progress in the backtracking, the gradients in the working set, the quasi-Newton updated, etc. are printed. A summary report is printed upon completion. |

## Declaration

public class com.imsl.math.MinConNLP
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *interface* MinConNLP.Function

Public interface for the user supplied function to the `MinConNLP` object.

### Declaration

public static interface com.imsl.math.MinConNLP.Function

### Method

---

- *f*
  `double f( double[] x, int iact, boolean[] ierr )`
  - **Description**
    Compute the value of the function at the given point.
  - **Parameters**
    * `x` – an input `double` array, the point at which the objective function or constraint is to be evaluated
    * `iact` – an input `int` value indicating whether evaluation of the objective function is requested or evaluation of a constraint is requested. If iact is zero, then an objective function evaluation is requested. If iact is nonzero then the value of iact indicates the index of the constraint to evaluate. (1 indicates the first constraint, 2 indicates the second, etc.)
    * `ierr` – an input/output `boolean` array of length 1. On input ierr[0] is set to false. If an error or other undesirable condition occurs during evaluation, then ierr[0] should be set to true. Setting ierr[0] to true will result in the step size being reduced and the step being tried again. (If ierr[0] is set to true for xguess, then an error is issued.)
  - **Returns** – a `double`. If iact is zero, then the value of the objective function at x is returned. If iact is nonzero, then the computed constraint value at the point x is returned.

## *interface* MinConNLP.Gradient

Public interface for the user supplied function to compute the gradient for `MinConNLP` object.

### Declaration

public static interface com.imsl.math.MinConNLP.Gradient
**implements** MinConNLP.Function

### Method

---

- *gradient*
  `void` **gradient**`( double[] x, int` **iact**`, double[]` **result** `)`
  - **Description**
    Computes the value of the gradient of the function at the given point.
  - **Parameters**
    * `x` – an input `double` array, the point at which the gradient of the objective function or gradient of a constraint is to be evaluated
    * `iact` – an input `int` value indicating whether evaluation of the objective function gradient is requested or evaluation of a constraint gradient is requested. If iact is zero, then an objective function gradient evaluation is requested. If iact is nonzero then the value of iact indicates the index of the constraint gradient to evaluate. (1 indicates the first constraint, 2 indicates the second, etc.)
    * `result` – a `double` array. If iact is zero, then the value of the objective function gradient at x is returned in result. If iact is nonzero, then the computed gradient of the requested constraint value at the point x is returned in result.

## *class* MinConNLP.ConstraintEvaluationException

Constraint evaluation returns an error with current point.

### Declaration

public static class com.imsl.math.MinConNLP.ConstraintEvaluationException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConNLP.ConstraintEvaluationException*
  `public` **MinConNLP.ConstraintEvaluationException**`( java.lang.String` **message** `)`

- *MinConNLP.ConstraintEvaluationException*
  `public` **MinConNLP.ConstraintEvaluationException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* **MinConNLP.ObjectiveEvaluationException**

Objective evaluation returns an error with current point.

### Declaration

public static class com.imsl.math.MinConNLP.ObjectiveEvaluationException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConNLP.ObjectiveEvaluationException*
  `public` **MinConNLP.ObjectiveEvaluationException**`( java.lang.String` **message** `)`

- *MinConNLP.ObjectiveEvaluationException*
  `public` **MinConNLP.ObjectiveEvaluationException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* **MinConNLP.NoAcceptableStepsizeException**

No acceptable stepsize in [SIGMA,SIGLA].

### Declaration

public static class com.imsl.math.MinConNLP.NoAcceptableStepsizeException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *MinConNLP.NoAcceptableStepsizeException*
  `public` **MinConNLP.NoAcceptableStepsizeException**`( java.lang.String` **message** `)`

- *MinConNLP.NoAcceptableStepsizeException*
  `public` **MinConNLP.NoAcceptableStepsizeException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **MinConNLP.WorkingSetSingularException**

Working set is singular in dual extended QP.

#### Declaration

public static class com.imsl.math.MinConNLP.WorkingSetSingularException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.WorkingSetSingularException*
  `public` **MinConNLP.WorkingSetSingularException**`( java.lang.String` **message** `)`

- *MinConNLP.WorkingSetSingularException*
  `public` **MinConNLP.WorkingSetSingularException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **MinConNLP.QPInfeasibleException**

QP problem seemingly infeasible.

#### Declaration

public static class com.imsl.math.MinConNLP.QPInfeasibleException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.QPInfeasibleException*
  `public` **MinConNLP.QPInfeasibleException**`( java.lang.String` **message** `)`

- *MinConNLP.QPInfeasibleException*
  `public` **MinConNLP.QPInfeasibleException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* MinConNLP.PenaltyFunctionPointInfeasibleException

Penalty function point infeasible.

#### Declaration

public static class com.imsl.math.MinConNLP.PenaltyFunctionPointInfeasibleException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.PenaltyFunctionPointInfeasibleException*
  `public` **MinConNLP.PenaltyFunctionPointInfeasibleException**`(`
  `java.lang.String` **message** `)`

- *MinConNLP.PenaltyFunctionPointInfeasibleException*
  `public` **MinConNLP.PenaltyFunctionPointInfeasibleException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* MinConNLP.LimitingAccuracyException

Limiting accuracy reached for a singular problem.

#### Declaration

public static class com.imsl.math.MinConNLP.LimitingAccuracyException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.LimitingAccuracyException*
  `public` **MinConNLP.LimitingAccuracyException**`( java.lang.String`
  **message** `)`

- *MinConNLP.LimitingAccuracyException*
  `public` **MinConNLP.LimitingAccuracyException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

*class* **MinConNLP.TooManyIterationsException**

Maximum number of iterations exceeded.

**Declaration**

public static class com.imsl.math.MinConNLP.TooManyIterationsException
**extends** com.imsl.IMSLException (page 1240)

**Constructors**

---

- *MinConNLP.TooManyIterationsException*
  `public` **MinConNLP.TooManyIterationsException**`( java.lang.String` **message** `)`
- *MinConNLP.TooManyIterationsException*
  `public` **MinConNLP.TooManyIterationsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

*class* **MinConNLP.BadInitialGuessException**

Penalty function point infeasible for original problem. Try new initial guess.

**Declaration**

public static class com.imsl.math.MinConNLP.BadInitialGuessException
**extends** com.imsl.IMSLException (page 1240)

**Constructors**

---

- *MinConNLP.BadInitialGuessException*
  `public` **MinConNLP.BadInitialGuessException**`( java.lang.String` **message** `)`
- *MinConNLP.BadInitialGuessException*
  `public` **MinConNLP.BadInitialGuessException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **MinConNLP.IllConditionedException**

Problem is singular or ill-conditioned.

#### Declaration

public static class com.imsl.math.MinConNLP.IllConditionedException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.IllConditionedException*
  `public` **MinConNLP.IllConditionedException**`( java.lang.String` **message** `)`

- *MinConNLP.IllConditionedException*
  `public` **MinConNLP.IllConditionedException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **MinConNLP.SingularException**

Problem is singular.

#### Declaration

public static class com.imsl.math.MinConNLP.SingularException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.SingularException*
  `public` **MinConNLP.SingularException**`( java.lang.String` **message** `)`

- *MinConNLP.SingularException*
  `public` **MinConNLP.SingularException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **MinConNLP.LinearlyDependentGradientsException**

Working set gradients are linearly dependent.

#### Declaration

public static class com.imsl.math.MinConNLP.LinearlyDependentGradientsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.LinearlyDependentGradientsException*
  `public` **MinConNLP.LinearlyDependentGradientsException**`(`
  `java.lang.String` **message** `)`
- *MinConNLP.LinearlyDependentGradientsException*
  `public` **MinConNLP.LinearlyDependentGradientsException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* **MinConNLP.TerminationCriteriaNotSatisfiedException**

Termination criteria are not satisfied.

#### Declaration

public static class com.imsl.math.MinConNLP.TerminationCriteriaNotSatisfiedException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *MinConNLP.TerminationCriteriaNotSatisfiedException*
  `public` **MinConNLP.TerminationCriteriaNotSatisfiedException**`(`
  `java.lang.String` **message** `)`
- *MinConNLP.TerminationCriteriaNotSatisfiedException*
  `public` **MinConNLP.TerminationCriteriaNotSatisfiedException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* **MinConNLP.Formatter**

Simple formatter for MinConNLP logging

### Declaration

public static class com.imsl.math.MinConNLP.Formatter
**extends** java.util.logging.Formatter

### Constructor

---

- *MinConNLP.Formatter*
  `public` **MinConNLP.Formatter**`( )`

### Method

---

- *format*
  `public abstract java.lang.String` **format**`( java.util.logging.LogRecord )`

## Constructor

---

- *MinConNLP*
  `public` **MinConNLP**`( int` **mTotalConstraints**`, int` **mEqualityConstraints**`, int` **nVariables** `) throws java.lang.IllegalArgumentException`
  - **Description**
    Nonlinear programming solver constructor.
  - **Parameters**
    * `mTotalConstraints` – An `int` scalar value which defines the total number of constraints
    * `mEqualityConstraints` – An `int` scalar value which defines the number of equality constraints
    * `nVariables` – An `int` scalar value which defines the number of variables.

## Methods

- *getConstraintResiduals*
  `public double[]` **getConstraintResiduals**`( )`
  - **Description**
    Returns the constraint residuals.
  - **Returns** – a `double` array containing the constraint residuals.

- *getLagrangeMultiplierEst*
  `public double[]` **getLagrangeMultiplierEst**`( )`
  - **Description**
    Returns the Lagrange multiplier estimates of the constraints.
  - **Returns** – a `double` array containing the Lagrange multiplier estimates of the constraints.

- *getLogger*
  `public java.util.logging.Logger` **getLogger**`( )`
  - **Description**
    Returns the logger object. Logger support requires JDK1.4. Use with earlier versions returns null.
  - **Returns** – the logger object, if present, or null.

- *setBindingThreshold*
  `public void` **setBindingThreshold**`( double` **del0** `)`
  - **Description**
    Set the binding threshold for constraints. In the initial phase of minimization a constraint is considered binding if $\frac{g_i(x)}{max(1,\|\nabla g_i(x)\|)} \leq del0 \qquad i = M_e + 1, \ldots, M$ Good values are between .01 and 1.0. If del0 is chosen too small then identification of the correct set of binding constraints may be delayed. Contrary, if del0 is too large, then the method will often escape to the full regularized SQP method, using individual slack variables for any active constraint, which is quite costly. For well scaled problems del0 = 1.0 is reasonable. If this member function is not called, del0 is set to .5 * tau0.
  - **Parameters**
    - `del0` – a `double` scalar value specifying the binding threshold for constraints.
  - **Throws**

* `java.lang.IllegalArgumentException` – is thrown if del0 is less than or equal to 0.0

---

- *setBoundViolationBound*
  `public void` **setBoundViolationBound**`( double` **taubnd** `)`
  - **Description**
    Set the amount by which bounds may be violated during numerical differentiation. If this member function is not called, taubnd is set to 1.0.
  - **Parameters**
    * `taubnd` – a `double` scalar value specifying the amount by which bounds may be violated during numerical differentiation.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if taubnd is less than or equal to 0.0

---

- *setDifferentiationType*
  `public void` **setDifferentiationType**`( int` **idtype** `)`
  - **Description**
    Set the type of numerical differentiation to be used.
  - **Parameters**
    * `idtype` – an `int` scalar value specifying the type of numerical differentiation to be used. If this member function is not called, idtype is set to 1.

| *idtype* | *Action* |
|---|---|
| 1 | Use a forward difference quotient with discretization stepsize $0.1\left(epsfcn^{1/2}\right)$ componentwise relative. This is the default value used. |
| 2 | Use the symmetric difference quotient with discretization stepsize $0.1\left(epsfcn^{1/3}\right)$ componentwise relative. |
| 3 | Use the sixth order approximation computing a Richardson extrapolation of three symmetric difference quotient values. This uses a discretization stepsize $0.01\left(epsfcn^{1/7}\right)$ |

  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if idtype is less than or equal to 0 or greater than or equal to 4.

---

- *setFunctionPrecision*
  `public void` **setFunctionPrecision**`( double` **epsfcn** `)`
  - **Description**
    Set the relative precision of the function evaluation routine. If this member function is not called, epsfcn is set to 2.2e-16.
  - **Parameters**
    * `epsfcn` – a `double` scalar value specifying the relative precision of the function evaluation routine.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if epsfcn is less than or equal to 0.0

---

- *setGradientPrecision*
  `public void` **setGradientPrecision**`( double` **epsdif** `)`
  - **Description**
    Set the relative precision in gradients. If this member function is not called, epsdif is set to 2.2e-16.
  - **Parameters**
    * `epsdif` – a `double` scalar value specifying the relative precision in gradients.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if epsdif is less than or equal to 0.0

---

- *setGuess*
  `public void` **setGuess**`( double[]` **xguess** `)`
  - **Description**
    Set the initial guess of the minimum point of the input function. If this member function is not called, the elements of this array are set to x, (with the smallest value of $\|x\|_2$) that satisfies the bounds.
  - **Parameters**
    * `xguess` – a `double` array specifying the initial guess of the minimum point of the input function

---

- *setMaxIterations*
  `public void` **setMaxIterations**`( int` **maxIterations** `)`
  - **Description**
    Set the maximum number of iterations allowed. If this member function is not called, the maximum number of iterations is set to 200.
  - **Parameters**

- * `maxIterations` – an `int` specifying the maximum number of iterations allowed
- **Throws**
  - * `java.lang.IllegalArgumentException` – is thrown if `maxIterations` is less than or equal to 0

---

- *setMultiplierError*
  `public void` **setMultiplierError**`( double` **smallw** `)`
  - **Description**
    Set the error allowed in the multipliers. A negative multiplier of an inequality constraint is accepted (as zero) if its absolute value is less than smallw. If this member function is not called, it is set to $e^{2\log\epsilon/3}$.
  - **Parameters**
    - * `smallw` – a `double` scalar value specifying the error allowed in the multipliers.
  - **Throws**
    - * `java.lang.IllegalArgumentException` – is thrown if smallw is less than or equal to 0.0

---

- *setPenaltyBound*
  `public void` **setPenaltyBound**`( double` **tau0** `)`
  - **Description**
    Set the universal bound for describing how much the unscaled penalty-term may deviate from zero. A small tau0 diminishes the efficiency of the solver because the iterates then will follow the boundary of the feasible set closely. Conversely, a large tau0 may degrade the reliability of the code. If this member function is not called, tau0 is set to 1.0.
  - **Parameters**
    - * `tau0` – a `double` scalar value specifying the universal bound for describing how much the unscaled penalty-term may deviate from zero.
  - **Throws**
    - * `java.lang.IllegalArgumentException` – is thrown if tau0 is less than or equal to 0.0

---

- *setScalingBound*
  `public void` **setScalingBound**`( double` **scbnd** `)`
  - **Description**
    Set the scaling bound for the internal automatic scaling of the objective function. If this member function is not called, scbnd is set to 1.0e4.
  - **Parameters**

* `scbnd` – a `double` scalar value specifying the scaling variable for the problem function.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if scbnd is less than or equal to 0.0

---

• *setViolationBound*

`public void` **setViolationBound**`( double` **delmin** `)`

– **Description**

Set the scalar which defines allowable constraint violations of the final accepted result. Constraints are satisfied if $|g_i(x)| \leq delmin$, and $g_i(x) \geq -delmin$ respectively. If this member function is not called, delmin is set to $min(del0/10, max(epsdif, min(del0/10, max((1.e-6)del0, small_w))))$.

– **Parameters**

* `delmin` – a `double` scalar value specifying the allowable constraint violations of the final accepted result.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if delmin is less than or equal to 0.0

---

• *setXlowerBound*

`public void` **setXlowerBound**`( double[]` **xlb** `)`

– **Description**

Set the lower bounds on the variables. If this member function is not called, the elements of this array are set to -1.79e308.

– **Parameters**

* `xlb` – a `double` array specifying the lower bounds on the variables

---

• *setXscale*

`public void` **setXscale**`( double[]` **xscale** `)`

– **Description**

Set the internal scaling of the variables. The initial value given and the objective function and gradient evaluations, however, are always given in the original unscaled variables. The first internal variable is obtained by dividing the values x[i] by xscale[i]. If this member function is not called, xscale[i] is set to 1.0.

– **Parameters**

* `xscale` – a `double` array specifying the internal scaling of the variables.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if xscale is less than or equal to 0.0

---

- *setXupperBound*
  `public void` **setXupperBound**`( double[]` **xub** `)`
  - **Description**
    Set the upper bounds on the variables. If this member function is not called, the elements of this array are set to 1.79e308.
  - **Parameters**
    * `xub` – a `double` array specifying the upper bounds on the variables

---

- *solve*
  `public double[]` **solve**`( MinConNLP.Function` **F** `) throws`
  `com.imsl.math.MinConNLP.ConstraintEvaluationException,`
  `com.imsl.math.MinConNLP.ObjectiveEvaluationException,`
  `com.imsl.math.MinConNLP.WorkingSetSingularException,`
  `com.imsl.math.MinConNLP.QPInfeasibleException,`
  `com.imsl.math.MinConNLP.PenaltyFunctionPointInfeasibleException,`
  `com.imsl.math.MinConNLP.LimitingAccuracyException,`
  `com.imsl.math.MinConNLP.TooManyIterationsException,`
  `com.imsl.math.MinConNLP.BadInitialGuessException,`
  `com.imsl.math.MinConNLP.IllConditionedException,`
  `com.imsl.math.MinConNLP.SingularException,`
  `com.imsl.math.MinConNLP.LinearlyDependentGradientsException,`
  `com.imsl.math.MinConNLP.NoAcceptableStepsizeException,`
  `com.imsl.math.MinConNLP.TerminationCriteriaNotSatisfiedException`
  - **Description**
    Solve a general nonlinear programming problem using the successive quadratic programming algorithm with a finite-difference gradient or with a user-supplied gradient.
  - **Parameters**
    * `F` – defines the user-supplied function to evaluate the function at a given point. F can be used to supply a gradient of the function. If F implements `Gradient` the user-supplied gradient is used. Otherwise,an attempt to solve the problem is made using a finite-difference gradient.
  - **Returns** – a `double` array containing the solution of the nonlinear programming problem.

### Example 1: Solving a general nonlinear programming problem

A general nonlinear programming problem is solved using a finite difference gradient.

```
import com.imsl.math.*;

public class MinConNLPEx1 implements MinConNLP.Function{

    public double f(double[] x, int iact, boolean[] ierr){
            double result;
            ierr[0] = false;
                if(iact == 0){
                    result = (x[0]-2.e0)*(x[0]-2.e0) + (x[1]-1.e0)*(x[1]-1.e0);
                    return result;
                } else {
                    switch (iact) {
                        case 1:
                            result = (x[0]-2.e0*x[1] + 1.e0);
                            return result;
                        case 2:
                            result = (-(x[0]*x[0])/4.e0 - (x[1]*x[1]) + 1.e0);
                            return result;
                        default:
                            ierr[0] = true;
                            return 0.e0;
                    }
        }
    }

    public static void main(String args[]) throws Exception {
        int     m = 2;
        int     me = 1;
        int     n = 2;
        double xinit[] = {2., 2.};
        double  x[] = {0.};
        MinConNLP minconnon = new MinConNLP(m, me, n);
        minconnon.setGuess(xinit);
        MinConNLPEx1 fcn = new MinConNLPEx1();
        x = minconnon.solve(fcn);
        System.out.println("x is "+x[0] +" "+x[1]);
    }
}
```

### Output

```
x is 0.8228756555325116 0.9114378277662559
```

### Example 2: Solving a general nonlinear programming problem

A general nonlinear programming problem is solved using a user-supplied gradient.

```
import com.imsl.math.*;

public class MinConNLPEx2 implements MinConNLP.Gradient{

    public double f(double[] x, int iact, boolean[] ierr){
            double result;
            ierr[0] = false;
                if(iact == 0){
                    result = (x[0]-2.e0)*(x[0]-2.e0) + (x[1]-1.e0)*(x[1]-1.e0);
                    return result;
                } else {
                    switch (iact) {
                        case 1:
                            result = (x[0]-2.e0*x[1] + 1.e0);
                            return result;
                        case 2:
                            result = (-(x[0]*x[0])/4.e0 - (x[1]*x[1]) + 1.e0);
                            return result;
                        default:
                            ierr[0] = true;
                            return 0.e0;
                    }
        }
    }


    public void gradient(double[] x, int iact, double[] result){
                if(iact == 0){
                    result[0] = 2.e0*(x[0]-2.e0);
                    result[1] = 2.e0*(x[1]-1.e0);
                    return;
                } else {
                    switch (iact) {
                        case 1:
```

```
                        result[0] = 1.e0;
    result[1] = -2.e0;
                        return;
                    case 2:
                        result[0] = -0.5e0*x[0];
    result[1] = -2.e0*x[1];
                        return;
                }
        }
    }

    public static void main(String args[]) throws Exception {
        int     m = 2;
        int     me = 1;
        int     n = 2;
        MinConNLP minconnon = new MinConNLP(m, me, n);
        minconnon.setGuess(new double[]{2.,2.});
        MinConNLPEx2 grad = new MinConNLPEx2();
        double x[] = minconnon.solve(grad);
        System.out.println("x is "+x[0] +" "+x[1]);
    }
}
```

### Output

```
x is 0.8228756555325117 0.9114378277662558
```

### Example 3: Solving a general nonlinear programming problem with logging

A general nonlinear programming problem is solved using a finite difference gradient. Intermediate output is captured in a file named MinConNLPlog.txt. The level of output requested is FINE.

```
import com.imsl.math.*;
import com.imsl.Messages;
import com.imsl.IMSLException;
import java.util.logging.Logger;
import java.util.logging.LogRecord;
import java.util.logging.Level;
import java.util.logging.Handler;
```

```
public class MinConNLPEx3 implements MinConNLP.Function{

    public double f(double[] x, int iact, boolean[] ierr){
            double result;
            ierr[0] = false;
                if(iact == 0){
                    result = (x[0]-2.e0)*(x[0]-2.e0) + (x[1]-1.e0)*(x[1]-1.e0);
                    return result;
                } else {
                    switch (iact) {
                        case 1:
                            result = (x[0]-2.e0*x[1] + 1.e0);
                            return result;
                        case 2:
                            result = (-(x[0]*x[0])/4.e0 - (x[1]*x[1]) + 1.e0);
                            return result;
                        default:
                            ierr[0] = true;
                            return 0.e0;
                    }
        }
    }

    public static void main(String args[]) throws Exception {
        int     m = 2;
        int     me = 1;
        int     n = 2;
        double xinit[] = {2., 2.};
        double  x[] = {0.};
        MinConNLP minconnon = new MinConNLP(m, me, n);
        minconnon.setGuess(xinit);
        MinConNLPEx3 fcn = new MinConNLPEx3();
        Logger logger = minconnon.getLogger();
        Handler h = new java.util.logging.FileHandler("MinConNLPlog.txt");
        logger.addHandler(h);
        logger.setLevel(Level.FINE);
        h.setFormatter(new MinConNLP.Formatter());
        x = minconnon.solve(fcn);
        System.out.println("x is "+x[0] +" "+x[1]);
    }
}
```

## Output

```
x is 0.8228756555325116 0.9114378277662559
Contents of the file MinConNLPlog.txt after execution:


  ITSTEP= 1  FX= 0.0   UPSI= 5.0  B2N=-1.0  UMI= 0.0  NR= 2  SI= -1


  ITSTEP= 2  FX= 0.47222222222222204   UPSI= 0.8055555555555558  B2N=7.447602459741819E-16  UMI= 0.0  NR


  ITSTEP= 3  FX= 1.2261822533163689   UPSI= 0.09653353175869195  B2N=3.3306690738754696E-16  UMI= 0.0  N


  ITSTEP= 4  FX= 1.393242278445973   UPSI= 1.2061157826948055E-4  B2N=1.336885555457667E-15  UMI= 0.0  N

     N= 2     M= 2    ME= 1

  EPSX= 1.0E-5     SIGSM= 1.4901161193847656E-8

STARTVALUE
 0.02.0

   EPS= 2.220446049250313E-16   TOL= 2.2250738585072014E-308   DEL0= 0.5   DELM= 5.0E-7  TAU0= 1.0
   TAU= 0.1   SD= 0.1   SW= 5.4782007307014466E-33   RHO= 1.0E-6   RHO1=1.0E-10
 SCFM= 10000.0   C1D= 0.01   EPDI= 2.220446049250313E-16
  NRE= 2  ANAL= false
 VBND= 1.0   EFCN= 2.220446049250313E-16   DIFF= 1
TERMINATION REASON:
KT-CONDITIONS SATISFIED, NO FURTHER CORRECTION COMPUTED
EVALUATIONS OF F                                    18
EVALUATIONS OF GRAD F                                0
EVALUATIONS OF CONSTRAINTS                          48
EVALUATIONS OF GRADS OF CONSTRAINTS                  0
FINAL SCALING OF OBJECTIVE                          1.0
NORM OF GRAD(F)                                     2.360902457120518
LAGRANGIAN VIOLATION                                9.992007221626409E-16
FEASIBILITY VIOLATION                               2.866595849582154E-13
DUAL FEASIBILITY VIOLATION                          0.0
OPTIMIZER RUNTIME SEC S
```

```
OPTIMAL VALUE OF F = 1.3934649806887736

OPTIMAL SOLUTION X =
0.8228756555325116   0.9114378277662559
MULTIPLIERS ARE RELATIVE TO SCF=1
NR.     CONSTRAINT          NORMGRAD (OR 1)     MULTIPLIER
 1  -2.220446049250313E-16  2.23606797749979  -1.5944911588359063
 2  -2.864375403532904E-13  1.8687312653198707  1.8465915320074269
EVALUATIONS OF RESTRICTIONS AND THEIR GRADIENTS
 ( 24.0, 0.0 )
 ( 24.0, 0.0 )
LAST ESTIMATE OF CONDITION OF ACTIVE GRADIENTS  1.958467797854007
LAST ESTIMATE OF CONDITION OF APPROX. HESSIAN    1.3588763739672172
 ITERATIVE STEPS TOTAL          4
 # OF RESTARTS                  0
 # OF FULL REGULAR UPDATES      3
 # OF UPDATES                   3
 # OF FULL REGULARIZED SQP-STEPS 0
  FX= 1  SCF= 5.0  PSI= 1.8687312653198707  UPS= 1.8465915320074269
 DEL= 5.0E-5  B20= 0.0  B2N= -1.0  NR= 2
      SI= -1            U-= 0.0  C-R= 1.5365907428821477  C-D= 1.0
      XN= 2.8284271247461903  DN= 1.0671873729054746  PHA= -1           CL= 0
      SKM= 0.0  SIG= 1.0  CF+= 0.0          DIR= -5.0
      DSC= 0.0  COS= 1.0  VIO= 0.0
      UPD= 0           TK= 0.0  XSI= 0.0
  FX= 2  SCF= 0.8055555555555558  PSI= 0.0  UPS= 0
 DEL= 0.05  B20= 0.0  B2N= 7.447602459741819E-16  NR= 2
      SI= -1            U-= 0.0  C-R= 1.4798927762262672  C-D= 1.0
      XN= 1.7716909687891085  DN= 0.49125734684608885  PHA= 1            CL= 1
      SKM= 1.4727272299765986  SIG= 1.0  CF+= 1.0          DIR= -0.6737373565183514
      DSC= 1.4727272299765986  COS= 1.0  VIO= 0.9079593845004515
      UPD= 1           TK= 0.24133378083025844  XSI= 0.0
  FX= 3  SCF= 0.09653353175869195  PSI= 0.0  UPS= 0
 DEL= 0.05  B20= 0.0  B2N= 3.3306690738754696E-16  NR= 2
      SI= -1            U-= 0.0  C-R= 1.9355267257931226  C-D= 1.4591929871177434
      XN= 1.302259296758884  DN= 0.07742644541830818  PHA= 1            CL= 1
      SKM= 3.4500000422411627  SIG= 1.0  CF+= 2.0          DIR= -0.17617369749845635
      DSC= 3.4500000422411627  COS= 1.0  VIO= 1.0000000000000002
      UPD= 1           TK= 0.005994854450114255  XSI= 0.0
  FX= 4  SCF= 1.2061157826948055E-4  PSI= 0.0  UPS= 0
 DEL= 0.05  B20= 0.0  B2N= 1.336885555457667E-15  NR= 2
```

```
SI= -1           U-= 0.0  C-R= 1.958467797854007  C-D= 1.3588763739672172
XN= 1.2280376253662906  DN= 1.0192836585976224E-4  PHA= 2           CL= 1
SKM= 3.892584026079591  SIG= 1.0  CF+= 2.0          DIR= -2.468065092929623E-4
DSC= 3.892584026079591  COS= 1.0  VIO= 1.0000000000000002
UPD= 1          TK= 1.0389391766841544E-8  XSI= 0.0
```

# Chapter 9

# Special Functions

## Classes



## *class* Sfun

Collection of special functions.

### Declaration

```
public class com.imsl.math.Sfun
extends java.lang.Object
```

## Fields

- public static final `double` **EPSILON_SMALL**
  - The smallest relative spacing for doubles.
- public static final `double` **EPSILON_LARGE**
  - The largest relative spacing for doubles.

## Methods

- *beta*
  `public static double` **beta**`( double` **a**`, double` **b** `)`
  - **Description**
    Returns the value of the Beta function. The beta function is defined to be

    $$\beta(a,b)=\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}=\int_0^1 t^{a-1}(1-t)^{b-1}dt$$

    See `gamma` for the definition of $\Gamma(x)$.
    The method `beta` requires that both arguments be positive.
  - **Parameters**
    * `a` – a `double` value
    * `b` – a `double` value
  - **Returns** – a `double` value specifying the Beta function

---

- *betaIncomplete*
  `public static double` **betaIncomplete**`( double` **x**`, double` **p**`, double` **q** `)`
  - **Description**
    Returns the incomplete Beta function ratio. The incomplete beta function is defined to be

    $$I_x(p,\ q)=\frac{\beta_x(p,\ q)}{\beta(p,\ q)}=\frac{1}{\beta(p,\ q)}\int_0^x t^{p-1}(1-t)^{q-1}dt \text{ for } 0\le x\le 1,\ p>0,\ q>0$$

    See `beta` for the definition of $\beta(p,\ q)$.
    The parameters $p$ and $q$ must both be greater than zero. The argument $x$ must lie in the range 0 to 1. The incomplete beta function can underflow for

sufficiently small $x$ and large $p$; however, this underflow is not reported as an error. Instead, the value zero is returned as the function value.
The method `betaIncomplete` is based on the work of Bosten and Battiste (1974).

- **Parameters**
  * `x` – a `double` value specifying the upper limit of integration It must be in the interval [0,1] inclusive.
  * `p` – a `double` value specifying the first Beta parameter. It must be positive.
  * `q` – a `double` value specifying the second Beta parameter. It must be positive.
- **Returns** – a `double` value specifying the incomplete Beta function ratio

---

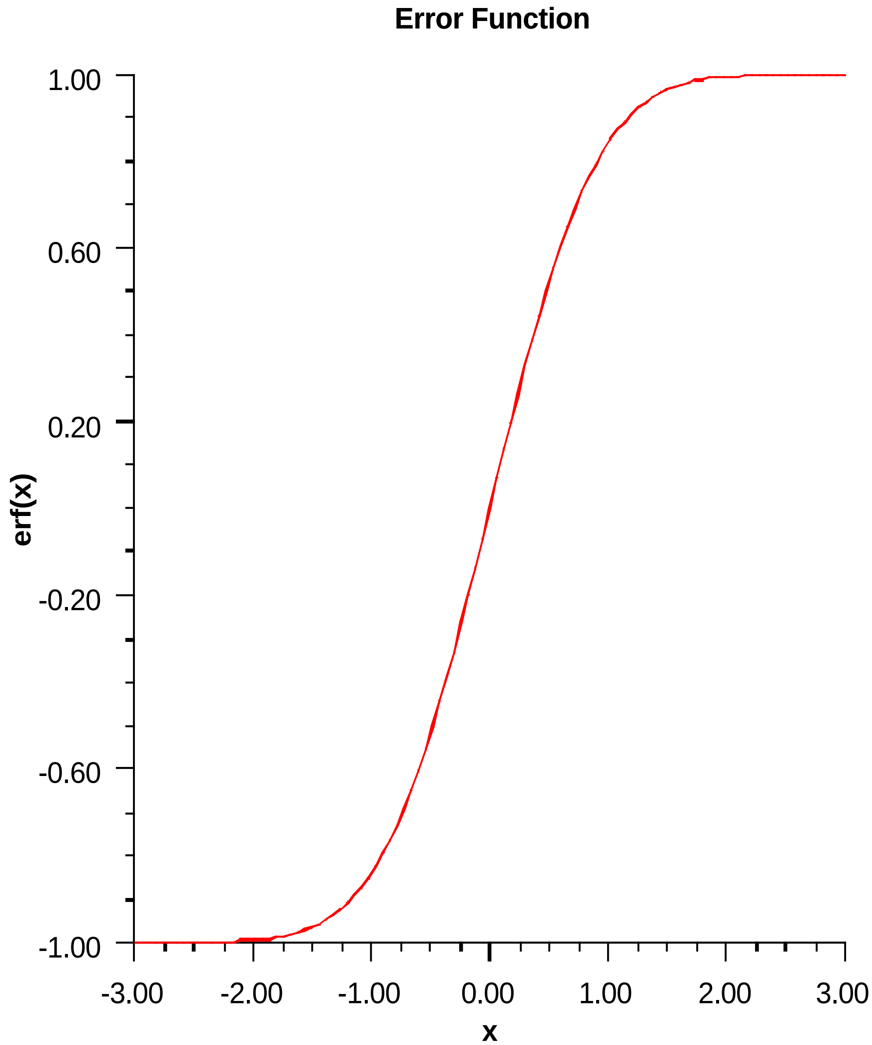
- *cot*
`public static double **cot**( double **x** )`
  - **Description**
    Returns the cotangent of a `double`.
  - **Parameters**
    * `x` – a `double` value
  - **Returns** – a `double` value specifying the cotangent of x. If x is NaN, the result is NaN.

---

- *erf*
`public static double **erf**( double **x** )`
  - **Description**
    Returns the error function of a `double`.
    The error function method, `erf`$(x)$, is defined to be

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

    All values of $x$ are legal.




– **Parameters**

  * `x` – a `double` value

– **Returns** – a `double` value specifying the error function of x

---

- *erfc*

`public static double erfc( double x )`

– **Description**

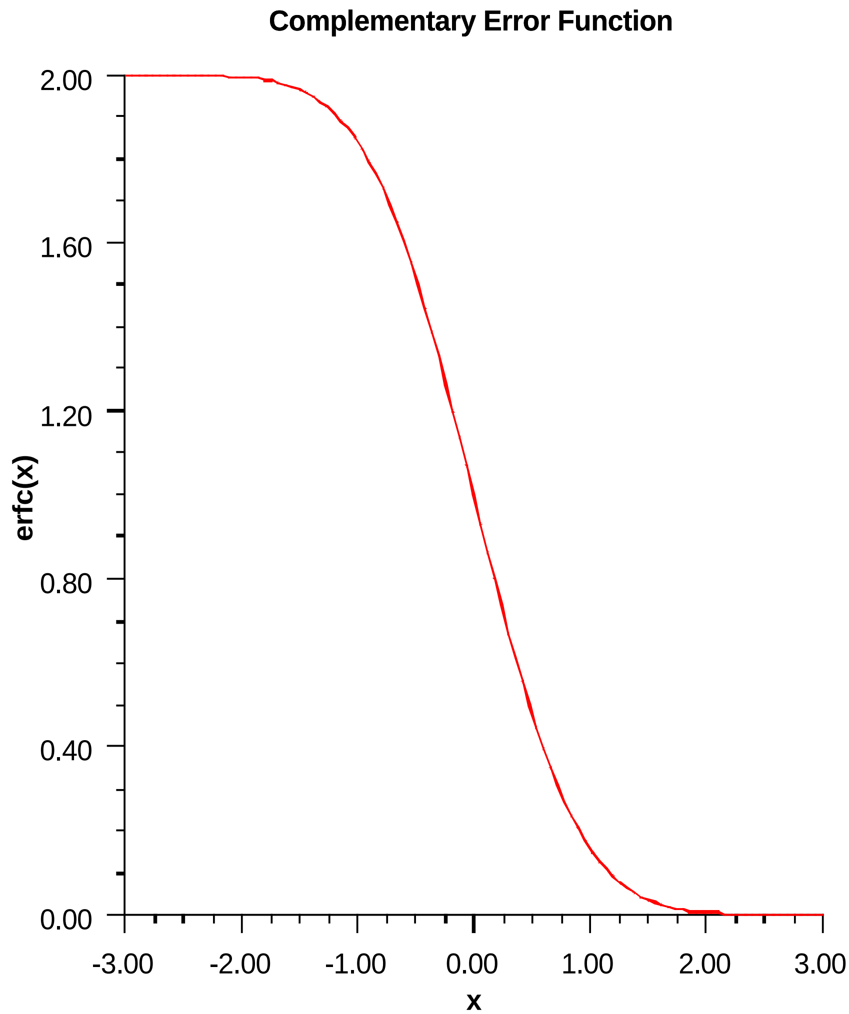
Returns the complementary error function of a `double`.
The complementary error function method, `erfc` $(x)$, is defined to be

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

The argument $x$ must not be so large that the result underflows.
Approximately, $x$ should be less than

$$\left[-ln\left(\sqrt{\pi}s\right)\right]^{1/2}$$

where $s = Double.MIN\_VALUE$ is the smallest representable positive floating-point number.




- **Parameters**
  - `x` – a `double` value
- **Returns** – a `double` value specifying the complementary error function of x

---

- *erfcInverse*

  `public static double` **erfcInverse**`( double x )`

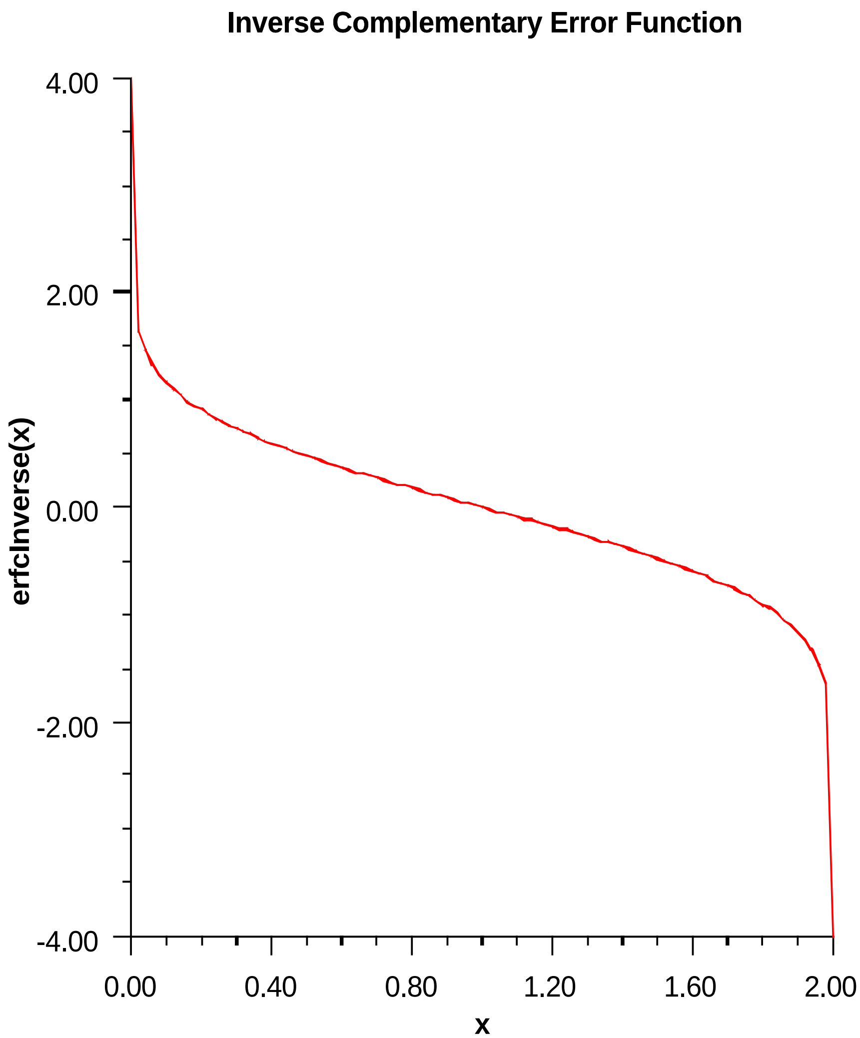
  - **Description**

    Returns the inverse of the complementary error function.

The `erfcinverse(x)` method computes the inverse of the complementary error function erfc x, defined in `erfc`.
`erfcinverse(x)` is defined for $0 < x < 2$. If $x_{\max} < x < 2$, then the answer will be less accurate than half precision. Very approximately,

$$x_{max} \approx 2 - \sqrt{\varepsilon/(4\pi)}$$

where $\varepsilon$ = machine precision (approximately 1.11e-16).




– **Parameters**

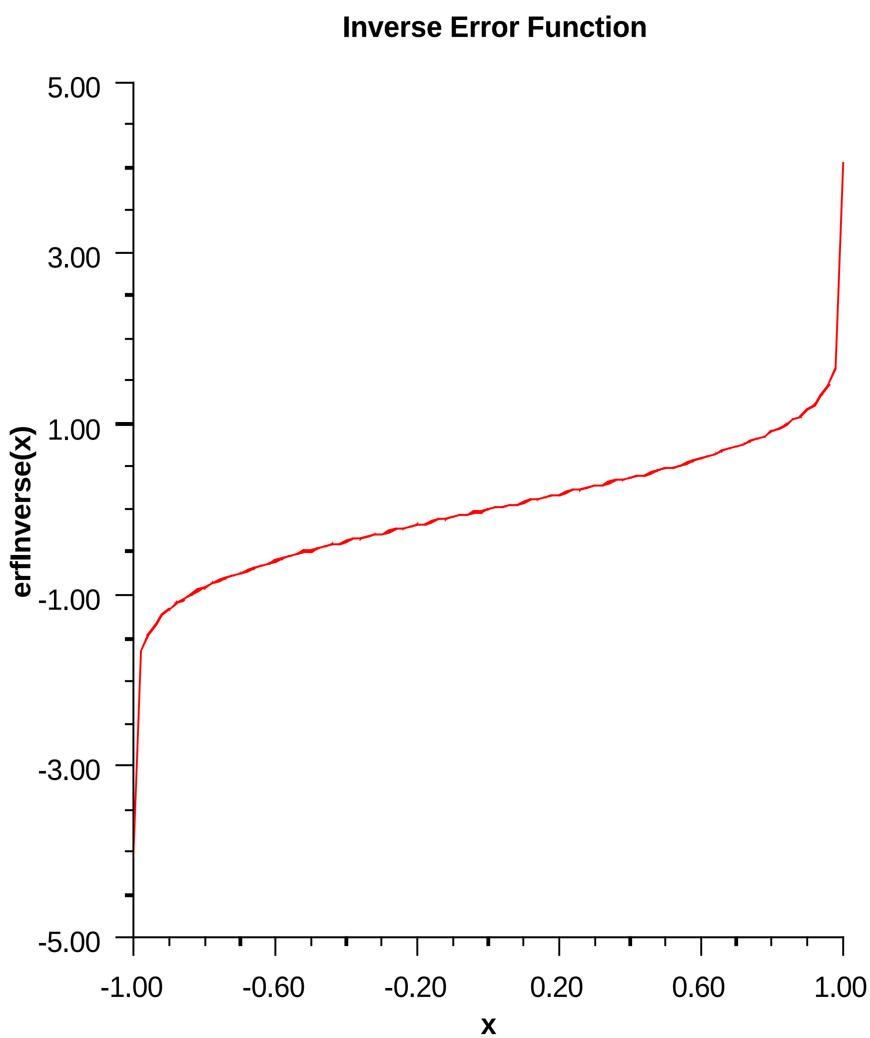
- * `x` – a `double` value, $0 \le x \le 2$.
  - **Returns** – a `double` value specifying the inverse of the error function of `x`.

---

- *erfInverse*
  `public static double` **erfInverse**`( double x )`
  - **Description**
    Returns the inverse of the error function.
    `erfInverse(X)` method computes the inverse of the error function erf $x$, defined in `erf`.
    The method `erfInverse(X)` is defined for $x_{max} < |x| < 1$, then the answer will be less accurate than half precision. Very approximately,

$$x_{\max} \approx 1 - \sqrt{\varepsilon / (4\pi)}$$

    where $\varepsilon$ is the machine precision (approximately 1.11e-16).




- **Parameters**
  - `x` – a `double` value
- **Returns** – a `double` value specifying the inverse of the error function of x

---

- *fact*

  `public static double` **fact**`( int` **n** `)`

  - **Description**

    Returns the factorial of an integer.

– **Parameters**

  * `n` – an `int` value

– **Returns** – a `double` value specifying the factorial of n, n!. If x is negative, the result is NaN.

---

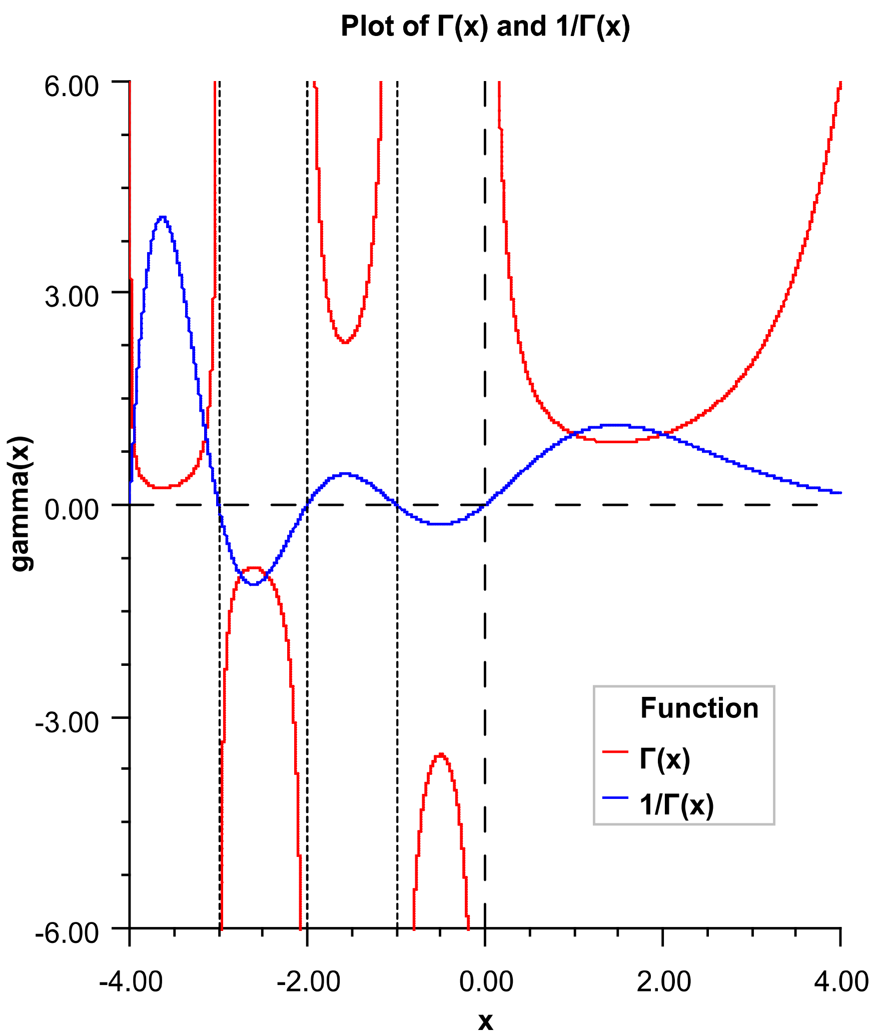
• *gamma*
`public static double` **`gamma`**`( double x )`

– **Description**

Returns the Gamma function of a `double`.
The gamma function, $\Gamma(x)$, is defined to be

$$\Gamma\left(x\right) = \int_0^\infty t^{x-1} e^{-t} dt \quad for\, x > 0$$

For $x < 0$, the above definition is extended by analytic continuation.
The gamma function is not defined for integers less than or equal to zero. Also, the argument $x$ must be greater than $-170.56$ so that $\Gamma(x)$ does not underflow, and $x$ must be less than *171.64* so that $\Gamma(x)$ does not overflow. The underflow limit occurs first for arguments that are close to large negative half integers. Even though other arguments away from these half integers may yield machine-representable values of $\Gamma(x)$, such arguments are considered illegal. Users who need such values should use the log gamma. Finally, the argument should not be so close to a negative integer that the result is less accurate than half precision.




– **Parameters**

  * `x` – a `double` value

– **Returns** – a `double` value specifying the Gamma function of x. If x is a negative integer, the result is NaN.

---

• *log10*

`public static double` **log10**`( double x )`

– **Description**

Returns the common (base 10) logarithm of a `double`.

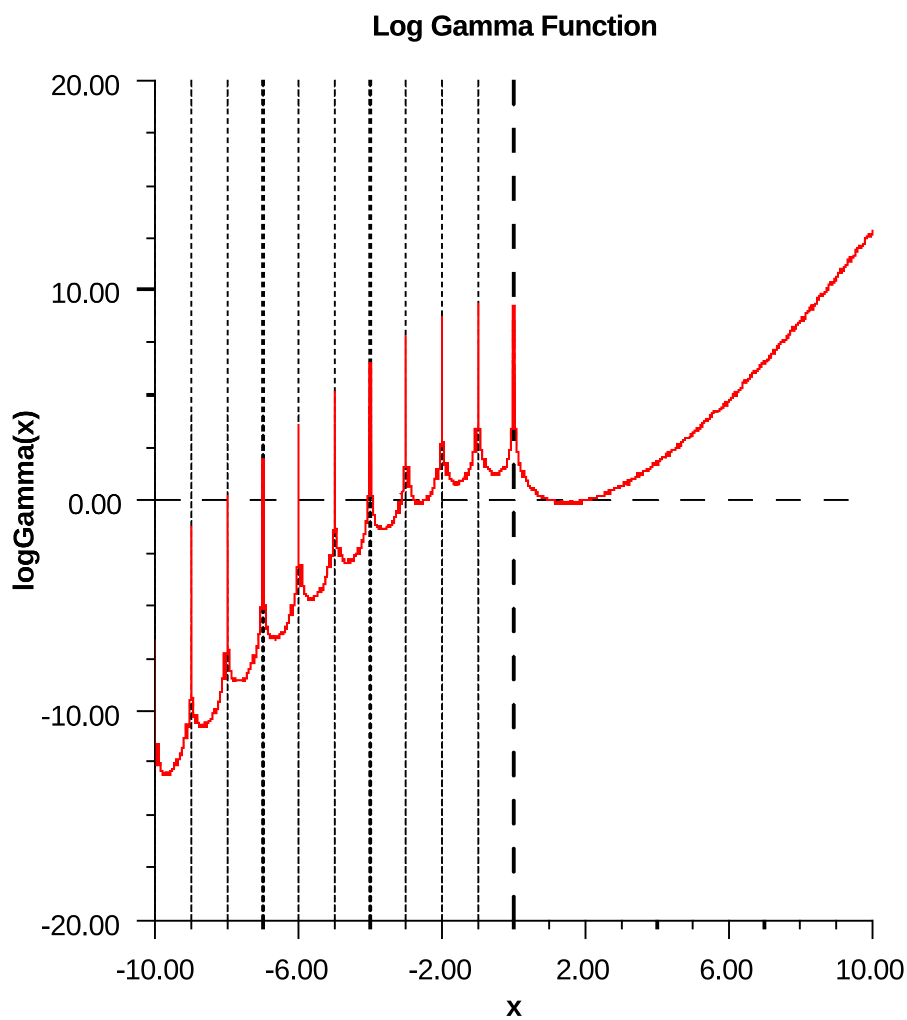
- **Parameters**
  - `x` – a `double` value
- **Returns** – a `double` value specifying the common logarithm of x

---

- *logBeta*
  `public static double` **logBeta**`( double a, double b )`
  - **Description**
    Returns the logarithm of the Beta function.
    Method `logBeta` computes $\ln \beta(a,b) = \ln \beta(b,a)$. See `beta` for the definition of $\beta(a,b)$.
    `logBeta` is defined for $a$ >0 and $b$ >0. It returns accurate results even when $a$ or $b$ is very small. It can overflow for very large arguments; this error condition is not detected except by the computer hardware.
  - **Parameters**
    - `a` – a `double` value
    - `b` – a `double` value
  - **Returns** – a `double` value specifying the natural logarithm of the Beta function

---

- *logGamma*
  `public static double` **logGamma**`( double x )`
  - **Description**
    Returns the logarithm of the Gamma function of the absolute value of a `double`.
    Method `logGamma` computes $\ln|\Gamma(x)|$. See `gamma` for the definition of $\Gamma(x)$.
    The gamma function is not defined for integers less than or equal to zero. Also, $|x|$ must not be so large that the result overflows. Neither should $x$ be so close to a negative integer that the accuracy is worse than half precision.




- **Parameters**
  - `x` – a `double` value
- **Returns** – a `double` value specifying the natural logarithm of the Gamma function of $|x|$. If x is a negative integer, the result is NaN.

---

- *poch*

`public static double` **poch**`( double` **a**`, double` **x** `)`

- **Description**

Returns a generalization of Pochhammer's symbol.
Method `poch` evaluates Pochhammer's symbol $(a)_n = (a)(a-1)\ldots(a-n+1)$ for n a nonnegative integer. Pochhammer's generalized symbol is defined to be

$$(a)_x = \frac{\Gamma(a+x)}{\Gamma(a)}$$

See `gamma` for the definition of $\Gamma(x)$.
Note that a straightforward evaluation of Pochhammer's generalized symbol with either gamma or log gamma functions can be especially unreliable when $a$ is large or $x$ is small.
Substantial loss can occur if $a + x$ or $a$ are close to a negative integer unless $|x|$ is sufficiently small. To insure that the result does not overflow or underflow, one can keep the arguments $a$ and $a + x$ well within the range dictated by the gamma function method gamma or one can keep $|x|$ small whenever $a$ is large. `poch` also works for a variety of arguments outside these rough limits, but any more general limits that are also useful are difficult to specify.

- **Parameters**
  * `a` – a `double` value specifying the first argument
  * `x` – a `double` value specifying the second, differential argument
- **Returns** – a `double` value specifying the generalized Pochhammer symbol, gamma(a+x)/gamma(a)

---

- *r9lgmc*
  `public static double` **r9lgmc**`( double x )`
  - **Description**
    Returns the log gamma correction term for argument values greater than or equal to 10.0.
  - **Parameters**
    * `x` – a `double` value
  - **Returns** – a `double` value specifying the log gamma correction term.

---

- *sign*
  `public static double` **sign**`( double x, double y )`
  - **Description**
    Returns the value of x with the sign of y.
  - **Parameters**
    * `x` – a `double` value
    * `y` – a `double` value
  - **Returns** – a `double` value specifying the absolute value of x and the sign of y

### Example: The Special Functions

Various special functions are exercised. Their use in this example typifies the manner in which other special functions in the Sfun class would be used.

```
import com.imsl.math.*;

public class SfunEx1 {
    public static void main(String args[]) {
        double result;

        // Log base 10 of x
        double x = 100.;
        result = Sfun.log10(x);
        System.out.println("The log base 10 of 100. is "+result);

        // Factorial of 10
        int n = 10;
        result = Sfun.fact(n);
        System.out.println("10 factorial is "+result);

        // Gamma of 5.0
        double x1 = 5.;
        result = Sfun.gamma(x1);
        System.out.println("The Gamma function at 5.0 is "+result);

        // LogGamma of 1.85
        double x2 = 1.85;
        result = Sfun.logGamma(x2);
        System.out.println("The logarithm of the absolute value of the " +
        "Gamma function \n    at 1.85 is " + result);

        // Beta of (2.2, 3.7)
        double a = 2.2;
        double b = 3.7;
        result = Sfun.beta(a, b);
        System.out.println("Beta(2.2, 3.7) is "+result);

        // LogBeta of (2.2, 3.7)
        double a1 = 2.2;
        double b1 = 3.7;
        result = Sfun.logBeta(a1, b1);
        System.out.println("logBeta(2.2, 3.7) is "+result + "\n");
```

```
    }
}
```

### Output

```
The log base 10 of 100. is 2.0
10 factorial is 3628800.0
The Gamma function at 5.0 is 24.0
The logarithm of the absolute value of the Gamma function
    at 1.85 is -0.05592381301965721
Beta(2.2, 3.7) is 0.045375983484708095
logBeta(2.2, 3.7) is -3.0927723120378947
```

## *class* **Bessel**

Collection of Bessel functions.

### Declaration

public class com.imsl.math.Bessel
**extends** java.lang.Object

### Methods

---

- *I*
  `public static double[] I( double xnu, double x, int n )`
  - **Description**
    Evaluates a sequence of modified Bessel functions of the first kind with real order and real argument. The Bessel function $I_v(x)$, is defined to be

    $$I_\nu(x) = \frac{1}{\pi}\int_0^\pi e^{x\cos\theta}\cos(\nu\theta)d\,\theta - \frac{\sin(\nu\pi)}{\pi}\int_0^\infty e^{-x\cosh t - vt}dt$$

    Here, argument `xnu` is represented by $\nu$ in the above equation.

The input x must be nonnegative and less than or equal to log(b) (b is the largest representable number). The argument $\nu$= `xnu` must satisfy $0 \leq \nu \leq 1$. This function is based on a code due to Cody (1983), which uses backward recursion.

– **Parameters**
  * `xnu` – a `double` representing the lowest order desired. `xnu` must be at least zero and less than 1
  * `x` – a `double` representing the argument of the Bessel functions to be evaluated
  * `n` – is the `int` order of the last element in the sequence

– **Returns** – a `double` array of length `n+1` containing the values of the function through the series. `Bessel.I[i]` contains the value of the Bessel function of order `i+xnu`.

---

• *I*

`public static double[] I( double x, int n )`

– **Description**

Evaluates a sequence of modified Bessel functions of the first kind with integer order and real argument. The Bessel function $I_n(x)$ is defined to be

$$I_n\,(x) = \frac{1}{\pi}\int_0^{\pi} e^{x\,\cos\,\theta}\,\cos\,(n\,\theta)\;d\,\theta$$

The input `x` must satisfy $|x| \leq \log(b)$ where b is the largest representable floating-point number. The algorithm is based on a code due to Sookne (1973b), which uses backward recursion.

– **Parameters**
  * `x` – a `double` representing the argument of the Bessel functions to be evaluated
  * `n` – is the `int` order of the last element in the sequence

– **Returns** – a `double` array of length `n+1` containing the values of the function through the series. `Bessel.I[i]` contains the value of the Bessel function of order i.

---

• *J*

`public static double[] J( double xnu, double x, int n )`

– **Description**

Evaluate a sequence of Bessel functions of the first kind with real order and real positive argument. The Bessel function $J_v(x)$, is defined to be

$$J_\nu(x) = \frac{(x/2)^\nu}{\sqrt{\pi}\Gamma(\nu+1/2)}\int_0^{\pi}\;\cos\,(x\;\cos\;\theta)\,\sin^{2\nu}\theta\;d\,\theta$$

This code is based on the work of Gautschi (1964) and Skovgaard (1975). It uses backward recursion.

- **Parameters**
  * `xnu` – a `double` representing the lowest order desired. `xnu` must be at least zero and less than 1.
  * `x` – a `double` representing the argument for which the sequence of Bessel functions is to be evaluated
  * `n` – an `int` representing the order of the last element in the sequence. If order is the highest order desired, set `n` to `int`(order).
- **Returns** – a `double` array of length `n+1` containing the values of the function through the series. `Bessel.J[I]` contains the value of the Bessel function of order I+v at `x` for I=0 to `n`.

---

- *J*

`public static double[] J( double x, int n )`

- **Description**

  Evaluates a sequence of Bessel functions of the first kind with integer order and real argument. The Bessel function $J_n(x)$, is defined to be

$$J_n\left(x\right) = \frac{1}{\pi}\int_0^{\pi} \cos\left(x\,\sin\,\theta - n\,\theta\right)d\,\theta$$

  The algorithm is based on a code due to Sookne (1973b) that uses backward recursion with strict error control.
- **Parameters**
  * `x` – a `double` representing the argument for which the sequence of Bessel functions is to be evaluated
  * `n` – an `int` which specifies the order of the last element in the sequence
- **Returns** – a `double` array of length `n+1` containing the values of the function through the series. `Bessel.J[i]` contains the value of the Bessel function of order i at `x` for i=0 to `n`.

---

- *K*

`public static double[] K( double xnu, double x, int n )`

- **Description**

  Evaluates a sequence of modified Bessel functions of the third kind with fractional order and real argument. The Bessel function $K_v(x)$ is defined to be

$$K_\nu(x) = \frac{\pi}{2}e^{\nu\pi i/2}\left[i\,J_\nu(ix) - Y_\nu(ix)\right] \quad \text{for} - \pi < \text{arg x} \le \frac{\pi}{2}$$

  Currently, `xnu` (represented by $\nu$ in the above equation) is restricted to be less than one in absolute value. A total of $n$ values is stored in the result, `K`.

K[0] $= K_v(x)$, K[1] $= K_{v+1}(x)$, ..., K$[n-1] = K_{v+n-1}(x)$.
This method is based on the work of Cody (1983).

– **Parameters**
  * `xnu` – a `double` representing the fractional order of the function. `xnu` must be less than one in absolute value.
  * `x` – a `double` representing the argument for which the sequence of Bessel functions is to be evaluated.
  * `n` – an `int` representing the order of the last element in the sequence. If order is the highest order desired, set `n` to int(order).

– **Returns** – a `double` array of length n+1 containing the values of the function through the series. Bessel.K[I] contains the value of the Bessel function of order I+v at `x` for I=0 to `n`.

---

• *K*

`public static double[] K( double x, int n )`

– **Description**
Evaluates a sequence of modified Bessel functions of the third kind with integer order and real argument. This function uses $e^x K_{\nu+k-1}$ for $k = 1, \ldots, n$ and $\nu = 0$. For the definition of $K_v(x)$, see above.

– **Parameters**
  * `x` – a `double` representing the argument for which the sequence of Bessel functions is to be evaluated
  * `n` – an `int` which specifies the order of the last element in the sequence

– **Returns** – a `double` array of length n+1 containing the values of the function through the series

---

• *scaledK*

`public static double[] scaledK( double v, double x, int n )`

– **Description**
Evaluate a sequence of exponentially scaled modified Bessel functions of the third kind with fractional order and real argument. This function evaluates $e^x K_{v+i-1}(x)$, for i=1,...,n where K is the modified Bessel function of the third kind. Currently, v is restricted to be less than 1 in absolute value. A total of $|n| + 1$ elements are returned in the array. This code is particularly useful for calculating sequences for large x provided n = x. (Overflow becomes a problem if $n << x$.) n must not be zero, and x must be greater than zero. $|v|$must be less than 1. Also, when $|n|$ is large compared with x, $|v + n|$ must not be so large that

$$e^x K_{\nu+n}(x) \approx e^x \frac{\Gamma(|\nu + n|}{2(x/2)^{|\nu+n|}}$$

overflows. The code is based on work of Cody (1983).

– **Parameters**

  * `v` – a `double` representing the fractional order of the function. `v` must be less than one in absolute value.
  * `x` – a `double` representing the argument for which the sequence of Bessel functions is to be evaluated.
  * `n` – an `int` representing the order of the last element in the sequence. If order is the highest order desired, set `n` to int(order).

– **Returns** – a `double` array of length `n+1` containing the values of the function through the series. If `n` is positive, `Bessel.K[I]` contains $e^x$ times the value of the Bessel function of order `I+v` at `x` for I=0 to `n`. If `n` is negative, `Bessel.K[I]` contains $e^x$ times the value of the Bessel function of order `v-I` at `x` for I=0 to `n`.

---

• *Y*

`public static double[] Y( double xnu, double x, int n )`

– **Description**

Evaluate a sequence of Bessel functions of the second kind with real nonnegative order and real positive argument. The Bessel function $Y_v(x)$ is defined to be

$$Y_\nu(x) = \frac{1}{\pi}\int_0^\pi \cos(x\sin\theta - \nu\theta)d\,\theta$$

$$-\frac{1}{\pi}\int_0^\infty \left[e^{\nu t} + e^{-\nu t}\cos(\nu\pi)\right]e^{-x\sinh t}\,dt$$

The variable `xnu` (represented by $\nu$in the above equation) must satisfy $0 \le \nu < 1$. If this condition is not met, then `Y` is set to `NaN`. In addition, `x` must be in $[x_m, x_M]$ where $x_m = 6(16^{-32})$ and $x_m = 16^9$. If $x < x_m$, then the largest representable number is returned; and if $x < x_M$, then zero is returned.
The algorithm is based on work of Cody and others, (see Cody et al. 1976; Cody 1969; NATS FUNPACK 1976). It uses a special series expansion for small arguments. For moderate arguments, an analytic continuation in the argument based on Taylor series with special rational minimax approximations providing starting values is employed. An asymptotic expansion is used for large arguments.

– **Parameters**

  * `xnu` – a `double` representing the lowest order desired. `xnu` must be at least zero and less than 1
  * `x` – a `double` representing the argument for which the sequence of Bessel functions is to be evaluated
  * `n` – an `int` such that `n+1` elements will be evaluated in the sequence

– **Returns** – a `double` array of length `n+1` containing the values of the function through the series. `Bessel.K[I]` contains the value of the Bessel function of order I+v at `x` for I=0 to `n`.

### Example: The Bessel Functions

The Bessel functions I, J, and K are exercised for orders 0, 1, 2, and 3 at argument 10.e0.

```
import com.imsl.math.*;

public class BesselEx1 {
    public static void main(String args[]) {
        double x = 10.e0;
        int hiorder = 4;
        // Exercise some of the Bessel functions with argument 10.0
        double bi[] = Bessel.I(x, hiorder);
        double bj[] = Bessel.J(x, hiorder);
        double bk[] = Bessel.K(x, hiorder);

        System.out.println("Order     Bessel.I              Bessel.J" +
        "              Bessel.K");
        for(int i = 0; i < 4; i++) {
            System.out.println(i+"     "+bi[i]+"     "+bj[i]+"     "+bk[i]);
        }
        System.out.println();
    }
}
```

### Output

```
Order     Bessel.I              Bessel.J              Bessel.K
0     2815.7166284662553     -0.24593576445134832     1.7780062316167654E-5
1     2670.9883037012555     0.043472746168861535     1.8648773453825585E-5
2     2281.5189677260046     0.2546303136851206     2.150981700693277E-5
3     1758.3807166108538     0.05837937930518672     2.725270025659869E-5
```

## *class* **JMath**

Pure Java implementation of the standard java.lang.Math class. This Java code is based on C code in the package fdlibm, which can be obtained from www.netlib.org.

### Declaration

public final class com.imsl.math.JMath
**extends** java.lang.Object

### Fields

---

- public static final `double` **PI**
- public static final `double` **E**

### Methods

---

- *abs*
  `public static double` **abs**`( double` **x** `)`
  - **Description**
    Returns the absolute value of a `double`.
  - **Parameters**
    * `x` – a `double`
  - **Returns** – a `double` representing $|x|$.

---

- *abs*
  `public static float` **abs**`( float` **x** `)`
  - **Description**
    Returns the absolute value of a `float`.
  - **Parameters**
    * `x` – a `float`
  - **Returns** – a `float` representing $|x|$.

---

- *abs*
  `public static int` **abs**`( int` **x** `)`
  - **Description**
    Returns the absolute value of an `int`.
  - **Parameters**
    * `x` – an `int`

- **Returns** – an `int` representing $|x|$.

---

- *abs*
  `public static long` **abs**`( long` **x** `)`
  - **Description**
    Returns the absolute value of a `long`.
  - **Parameters**
    * `x` – a `long`
  - **Returns** – a `long` representing $|x|$.

---

- *acos*
  `public static double` **acos**`( double` **x** `)`
  - **Description**
    Returns the inverse (arc) cosine of a `double`.
  - **Parameters**
    * `x` – a `double`
  - **Returns** – a `double` representing the angle, in radians, whose cosine is x. It is in the range $[0, \pi]$.

---

- *asin*
  `public static double` **asin**`( double` **x** `)`
  - **Description**
    Returns the inverse (arc) sine of a `double`.
  - **Parameters**
    * `x` – a `double`
  - **Returns** – a `double` representing the angle, in radians, whose sine is x. It is in the range $[-\pi/2, \pi/2]$.

---

- *atan*
  `public static double` **atan**`( double` **x** `)`
  - **Description**
    Returns the inverse (arc) tangent of a `double`.
  - **Parameters**
    * `x` – a `double`
  - **Returns** – a `double` representing the angle, in radians, whose tangent is x. It is in the range $[-\pi/2, \pi/2]$.

---

- *atan2*
  `public static double` **atan2**`( double` **y**`, double` **x** `)`

– **Description**
Returns the angle corresponding to a Cartesian point.

– **Parameters**
  * `x` – a `double`, the first argument
  * `y` – a `double`, the second argument

– **Returns** – a `double` representing the angle, in radians, the the line from (0,0) to (x,y) makes with the x-axis. It is in the range $[-\pi, \pi]$.

---

• *ceil*
`public static double` **ceil**`( double x )`

– **Description**
Returns the value of a `double` rounded toward positive infinity to an integral value.

– **Parameters**
  * `x` – a `double`

– **Returns** – the smallest `double`, not less than x, that is an integral value

---

• *cos*
`public static double` **cos**`( double x )`

– **Description**
Returns the cosine of a `double`.

– **Parameters**
  * `x` – a `double`, assumed to be in radians

– **Returns** – a `double`, the cosine of x

---

• *exp*
`public static double` **exp**`( double x )`

– **Description**
Returns the exponential of a `double`. Special cases: $e^{\infty}$ is $\infty$, $e^{\mathrm{NaN}}$ is NaN; $e^{-\infty}$ is 0, and for finite argument, only $e^0 = 1$ is exact.

– **Parameters**
  * `x` – a `double`.

– **Returns** – a `double` representing $e^x$.

---

• *floor*
`public static double` **floor**`( double x )`

– **Description**
Returns the value of a `double` rounded toward negative infinity to an integral value.

- **Parameters**
  - `x` – a `double`
- **Returns** – the smallest `double`, not greater than x, that is an integral value

---

- *IEEEremainder*
  `public static double` **IEEEremainder**`( double x, double p )`
  - **Description**
    Returns the IEEE remainder from $x$ divided by $p$. The IEEE remainder is $x\%p = x - [x/p] \times p$ as if in infinite precise arithmetic, where $[x/p]$ is the (infinite bit) integer nearest $x/p$ (in half way case choose the even one).
  - **Parameters**
    - `x` – a `double`, the dividend
    - `p` – a `double`, the divisor
  - **Returns** – a `double` representing the remainder computed according to the IEEE 754 standard.

---

- *log*
  `public static double` **log**`( double x )`
  - **Description**
    Returns the natural logarithm of a `double`.
  - **Parameters**
    - `x` – a `double`
  - **Returns** – a `double` representing the natural (base e) logarithm of x

---

- *max*
  `public static double` **max**`( double x, double y )`
  - **Description**
    Returns the larger of two `double`s.
  - **Parameters**
    - `x` – a `double`
    - `y` – a `double`
  - **Returns** – a `double`, the larger of x and y. This function considers -0.0 to be less than 0.0.

---

- *max*
  `public static float` **max**`( float x, float y )`
  - **Description**
    Returns the larger of two `float`s.
  - **Parameters**

* `x` – a `float`
* `y` – a `float`

– **Returns** – a `float`, the larger of x and y. This function considers -0.0f to be less than 0.0f.

---

- *max*

`public static int` **max**`( int` **x**`, int` **y** `)`

– **Description**
Returns the larger of two `int`s.

– **Parameters**

* `x` – an `int`
* `y` – an `int`

– **Returns** – an `int`, the larger of x and y

---

- *max*

`public static long` **max**`( long` **x**`, long` **y** `)`

– **Description**
Returns the larger of two `long`s.

– **Parameters**

* `x` – a `long`
* `y` – a `long`

– **Returns** – a `long`, the larger of x and y

---

- *min*

`public static double` **min**`( double` **x**`, double` **y** `)`

– **Description**
Returns the smaller of two `double`s.

– **Parameters**

* `x` – a `double`
* `y` – a `double`

– **Returns** – a `double`, the smaller of x and y. This function considers -0.0 to be less than 0.0.

---

- *min*

`public static float` **min**`( float` **x**`, float` **y** `)`

– **Description**
Returns the smaller of two `float`s.

– **Parameters**

* `x` – a `float`
* `y` – a `float`

– **Returns** – a `float`, the smaller of x and y. This function considers -0.0f to be less than 0.0f.

---

- *min*
  `public static int min( int x, int y )`
  - **Description**
    Returns the smaller of two `int`s.
  - **Parameters**
    * `x` – an `int`
    * `y` – an `int`
  - **Returns** – an `int` representing the smaller of x and y

---

- *min*
  `public static long min( long x, long y )`
  - **Description**
    Returns the smaller of two `long`s.
  - **Parameters**
    * `x` – a `long`
    * `y` – a `long`
  - **Returns** – a `long`, the smaller of x and y

---

- *pow*
  `public static double pow( double x, double y )`
  - **Description**
    Returns x to the power y.
  - **Parameters**
    * `x` – a `double`, the base
    * `y` – a `double`, the exponent
  - **Returns** – a `double`, x to the power y

---

- *random*
  `public static synchronized double random( )`
  - **Description**
    Returns a random number from a uniform distribution.
  - **Returns** – a `double` representing a random number from a uniform distribution

---

- *rint*
  `public static double rint( double x )`
  - **Description**
    Returns the value of a `double` rounded toward the closest integral value.

- **Parameters**
  - `x` – a `double`
- **Returns** – the `double` closest to x that is an integral value

---

- *round*
  `public static long` **round**`( double` **x** `)`
  - **Description**
    Returns the `long` closest to a given `double`.
  - **Parameters**
    - `x` – a `double`
  - **Returns** – the `long` closest to x

---

- *round*
  `public static int` **round**`( float` **x** `)`
  - **Description**
    Returns the integer closest to a given `float`.
  - **Parameters**
    - `x` – a `float`
  - **Returns** – the `int` closest to x

---

- *sin*
  `public static double` **sin**`( double` **x** `)`
  - **Description**
    Returns the sine of a `double`.
  - **Parameters**
    - `x` – a `double`, assumed to be in radians
  - **Returns** – a `double`, the sine of x

---

- *sqrt*
  `public static double` **sqrt**`( double` **x** `)`
  - **Description**
    Returns the square root of a `double`.
  - **Parameters**
    - `x` – a `double`
  - **Returns** – a `double` representing the square root of x

---

- *tan*
  `public static double` **tan**`( double` **x** `)`

– **Description**
Returns the tangent of a `double`.

– **Parameters**
  * `x` – a `double`, assumed to be in radians

– **Returns** – a `double`, the tangent of x

## *class* IEEE

Pure Java implementation of the IEEE 754 functions as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

This Java code is based on C code in the package fdlibm, which can be obtained from www.netlib.org. The original fdlibm C code contains the following notice.



### Declaration

public class com.imsl.math.IEEE
**extends** java.lang.Object

### Methods

---

- *copysign*
  `public static double` **copysign**`( double x, double y )`

  – **Description**
  Returns a value with the magnitude of x and with the sign bit of $y$. If $y$ is NaN then $|x|$ is returned.

  – **Parameters**
    * `x` – a `double` from which the magnitude will be gleaned
    * `y` – a `double` from which the sign will be gleaned

  – **Returns** – a `double` value with magnitude x and sign of y

---

- *finite*
  `public static boolean` **finite**`( double x )`

- **Description**
  Finite number test on an argument of type double.
- **Parameters**
  * `x` – the `double` which is to be tested
- **Returns** – true if x is a finite number, false if x is a NaN or an infinity

---

- *ilogb*
  `public static int` **ilogb**`( double` **x** `)`
  - **Description**
    Return the binary exponent of non-zero x.
  - **Parameters**
    * `x` – a `double`
  - **Returns** – an `int` representing the binary exponent of x. Special cases `ilogb(0) = -Integer.MAX_VALUE` and
    ilogb($\infty$) = ilogb($-\infty$) = ilogb($NaN$) = Integer.MAX_VALUE.

---

- *isNaN*
  `public static boolean` **isNaN**`( double` **x** `)`
  - **Description**
    NaN test on an argument of type double.
  - **Parameters**
    * `x` – the `double` which is to be tested
  - **Returns** – true if x is a NaN, false otherwise

---

- *nextAfter*
  `public static double` **nextAfter**`( double` **x**`, double` **y** `)`
  - **Description**
    Returns the next machine floating-point number next to x in the direction toward y.
  - **Parameters**
    * `x` – a `double`
    * `y` – a `double`
  - **Returns** – a `double` which represents the value which is closest to x in the interval bounded by x and y

---

- *scalbn*
  `public static double` **scalbn**`( double` **x**`, int` **n** `)`
  - **Description**
    Returns $2^n$ computed by exponent manipulation rather than by actually performing an exponentiation or a multiplication.

- **Parameters**
  - `x` – a `double`
  - `n` – an `int` representing the power to which 2 is raised
- **Returns** – a `double` representing $x2^n$.

---

• *unordered*

`public static boolean` **`unordered`**`( double x, double y )`

- **Description**

  Unordered test on a pair of `doubles`. Tests whether either of a pair of `doubles` is a NaN.
- **Parameters**
  - `x` – a `double`
  - `y` – a `double`
- **Returns** – true if either x or y is a NaN, false otherwise

## *class* Hyperbolic

Pure Java implementation of the hyperbolic functions and their inverses.

This Java code is based on C code in the package fdlibm, which can be obtained from www.netlib.org. The original fdlibm C code contains the following notice.



### Declaration

public class com.imsl.math.Hyperbolic
**extends** java.lang.Object

### Methods

---

• *acosh*

`public static double` **`acosh`**`( double x )`

– **Description**
Returns the inverse hyperbolic cosine of its argument. Specifically,
acosh(1) returns +0
acosh($\pm\infty$) returns $+\infty$
acosh($x$) returns NaN, if $|x| < 1$.

– **Parameters**
  * `x` – a `double` value representing the argument.

– **Returns** – a `double` value representing the number whose hyperbolic cosine is x.

---

• *asinh*
`public static double` **`asinh`**`( double x )`

– **Description**
Returns the inverse hyperbolic sine of its argument. Specifically,
asinh($\pm 0$) returns $\pm\infty$
asinh($\pm\infty$) returns $\pm\infty$

– **Parameters**
  * `x` – a `double` value representing the argument.

– **Returns** – a `double` value representing the number whose hyperbolic sine is x.

---

• *atanh*
`public static double` **`atanh`**`( double x )`

– **Description**
Returns the inverse hyperbolic tangent of its argument. Specifically,
atanh($\pm 0$) returns $\pm 0$
atanh($\pm 1$) returns $+\infty$
atanh(x) returns NaN, if $|x| > 1$.

– **Parameters**
  * `x` – a `double` value representing the argument.

– **Returns** – a `double` value representing the number whose hyperbolic tangent is x.

---

• *cosh*
`public static double` **`cosh`**`( double x )`

– **Description**
Returns the hyperbolic cosine of its argument. Specifically,
cosh($\pm 0$) returns 1.

cosh($\pm\infty$) returns $+\infty$

- **Parameters**
  - `x` – a `double` value representing the argument.
- **Returns** – a `double` value representing the hyperbolic cosine of x.

---

- *expm1*

`public static double expm1( double x )`

- **Description**
  Returns exp(x)-1, the exponential of x minus 1. Specifically,
  expm1($\pm 0$) returns $\pm 0$
  expm1($+\infty$) returns $\pm\infty$
  expm1($-\infty$) returns -1.
- **Parameters**
  - `x` – a `double` specifying the argument.
- **Returns** – a `double` value representing exp(x)-1.

---

- *log1p*

`public static double log1p( double x )`

- **Description**
  Returns log(1+x), the logarithm of (x plus 1). Specifically,
  log1p($\pm 0$) returns $\pm 0$
  log1p($-1$) returns $-\infty$
  log1p($x$) returns NaN, if $x < -1$.
  log1p($\pm\infty$) returns $\pm\infty$
- **Parameters**
  - `x` – a `double` value representing the argument.
- **Returns** – a `double` value representing log(1+x).

---

- *sinh*

`public static double sinh( double x )`

- **Description**
  Returns the hyperbolic sine of its argument. Specifically,
  sinh($\pm 0$) returns $\pm 0$
  sinh($\pm\infty$) returns $\pm\infty$
- **Parameters**
  - `x` – a `double` value representing the argument.

– **Returns** – a `double` value representing the hyperbolic sine of x.

---

- *tanh*

  `public static double tanh( double x )`

  – **Description**

  Returns the hyperbolic tangent of its argument. Specifically,
  tanh($\pm 0$) returns $\pm 0$
  tanh($\pm\infty$) returns $\pm 1$.

  – **Parameters**

  * `x` – a `double` value representing the argument.

  – **Returns** – a `double` value representing the hyperbolic tangent of x.

### Example: The Hyperbolic Functions

The Hyperbolic functions are exercised with argument 0.

```
import com.imsl.math.*;

public class HyperbolicEx1 {
    public static void main(String args[]) {
        // Exercise the hyperbolic functions with argument 0.0
        System.out.println("sinh(0.) is "+Hyperbolic.sinh(0.));
        System.out.println("cosh(0.) is "+Hyperbolic.cosh(0.));
        System.out.println("tanh(0.) is "+Hyperbolic.tanh(0.));
        System.out.println("asinh(0.) is "+Hyperbolic.asinh(0.));
        System.out.println("acosh(0.) is "+Hyperbolic.acosh(0.));
        System.out.println("atanh(0.) is "+Hyperbolic.atanh(0.));
    }
}
```

### Output

```
sinh(0.) is 0.0
cosh(0.) is 1.0
tanh(0.) is 0.0
asinh(0.) is 0.0
acosh(0.) is NaN
atanh(0.) is 0.0
```

# Chapter 10

# Miscellaneous

**Classes**



## *class* Complex

Set of mathematical functions for complex numbers. It provides the basic operations (addition, subtraction, multiplication, division) as well as a set of complex functions. The binary operations have the form, where op is `add`, `subtract`, `multiply` or `divide`.

```
public static Complex op(Complex x, Complex y)   // x op y
public static Complex op(Complex x, double y)    // x op y
public static Complex op(double x, Complex y)    // x op y
```

`Complex` objects are immutable. Once created there is no way to change their value. The functions in this class follow the rules for complex arithmetic as defined C9x *Annex G: IEC 559-compatible complex arithmetic.* The API is not the same, but handling of infinities, NaNs, and positive and negative zeros is intended to follow the same rules.

## Declaration

public class com.imsl.math.Complex
**extends** java.lang.Number
**implements** java.io.Serializable, java.lang.Cloneable

## Fields

- public static final `Complex` **i**
  - The imaginary unit. This constant is set to new Complex(0,1).
- public static `java.lang.String` **suffix**
  - String used in converting `Complex` to `String`. Default is *i*, but sometimes *j* is desired. Note that this is set for the class, not for a particular instance of a `Complex`.

## Constructors

- *Complex*
  `public` **Complex**`( )`
  - **Description**
    Constructs a `Complex` equal to zero.

- *Complex*
  `public` **Complex**`( Complex` **z** `)`
  - **Description**
    Constructs a `Complex` equal to the argument.
  - **Parameters**
    * `z` – a `Complex` object
  - **Throws**
    * `java.lang.NullPointerException` – is thrown if z is null

- *Complex*
  `public` **Complex**`( double` **re** `)`
  - **Description**
    Constructs a `Complex` with a zero imaginary part.

– **Parameters**
  * `re` – a `double` value equal to the real part of the `Complex` object

---

- *Complex*
  `public` **Complex**`( double` **re**`, double` **im** `)`
  – **Description**
    Constructs a `Complex` with real and imaginary parts given by the input arguments.
  – **Parameters**
    * `re` – a `double` value equal to the real part of the `Complex` object
    * `im` – a `double` value equal to the imaginary part of the `Complex` object

## Methods

---

- *abs*
  `public static double` **abs**`( Complex` **z** `)`
  – **Description**
    Returns the absolute value (modulus) of a `Complex`, `|`$z$`|`.
  – **Parameters**
    * `z` – a `Complex` object
  – **Returns** – a `double` value equal to the absolute value of the argument

---

- *acos*
  `public static Complex` **acos**`( Complex` **z** `)`
  – **Description**
    Returns the inverse cosine (arc cosine) of a `Complex`, with branch cuts outside the interval [-1,1] along the real axis.
    Specifically, if $z = x{+}iy$,
    $\mathrm{acos}(\bar{z}) = \overline{\mathrm{acos}(z)}$.
    $\mathrm{acos}(\pm 0 + i0)$ returns $\pi/2 - i0$.
    $\mathrm{acos}(-\infty + i\infty)$ returns $3\pi/4 - i\infty$.
    $\mathrm{acos}(+\infty + i\infty)$ returns $\pi/4 - i\infty$.
    $\mathrm{acos}(x + i\infty)$ returns $\pi/2 - i\infty$, for finite $x$.
    $\mathrm{acos}(-\infty + iy)$ returns $\pi - i\infty$, for positive-signed finite $y$.
    $\mathrm{acos}(+\infty + iy)$ returns $+0 - i\infty$, for positive-signed finite $y$.
    $\mathrm{acos}(\pm\infty + i\mathrm{NaN})$ returns $\mathrm{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).
    $\mathrm{acos}(\pm 0 + i\mathrm{NaN})$ returns $\pi/2 + i\mathrm{NaN}$.

acos(NaN + $i\infty$) returns NaN − $i\infty$.
acos($x + i$NaN) returns NaN + $i$NaN, for nonzero finite $x$.
acos(NaN + $iy$) returns NaN + $i$NaN, for finite $y$.
acos(NaN + $i$NaN) returns NaN + $i$NaN.

– **Parameters**
  * `z` – a `Complex` object
– **Returns** – A newly constructed `Complex` initialized to the inverse (arc) cosine of the argument. The real part of the result is in the interval $[0, \pi]$.

---

- *acosh*
  `public static Complex` **`acosh`**`( Complex` **`z`** `)`

  – **Description**
  Returns the inverse hyperbolic cosine (arc cosh) of a `Complex`, with a branch cut at values less than one along the real axis.
  Specifically, if $z = x+iy$,
  $\mathrm{acosh}(\bar{z}) = \overline{\mathrm{acosh}(z)}$.
  acosh($\pm 0 + i0$) returns $+0 + i\pi/2$.
  acosh($-\infty + i\infty$) returns $+\infty + i3\pi/4$.
  acosh($+\infty + i\infty$) returns $+\infty + i\pi/4$.
  acosh($x + i\infty$) returns $+\infty + i\pi/2$, for finite $x$.
  acosh($-\infty + iy$) returns $+\infty + i\pi$, for positive-signed finite $y$.
  acosh($+\infty + iy$) returns $+\infty + i0$, for positive-signed finite $y$.
  acosh(NaN + $i\infty$) returns $+\infty + i$NaN.
  acosh($\pm\infty + i$NaN) returns $+\infty + i$NaN.
  acosh($x + i$NaN) returns NaN + $i$NaN, for finite $x$.
  acosh(NaN + $iy$) returns NaN + $i$NaN, for finite $y$.
  acosh(NaN + $i$NaN) returns NaN + $i$NaN.

  – **Parameters**
    * `z` – a `Complex` object
  – **Returns** – A newly constructed `Complex` initialized to the inverse (arc) hyperbolic cosine of the argument. The real part of the result is non-negative and its imaginary part is in the interval $[-i\pi, i\pi]$.

---

- *add*
  `public static Complex` **`add`**`( Complex` **`x`**`, Complex` **`y`** `)`

  – **Description**
  Returns the sum of two `Complex` objects, $x+y$.
  – **Parameters**

* `x` – a `Complex` object
* `y` – a `Complex` object

– **Returns** – a newly constructed `Complex` initialized to $x+y$

---

- *add*

`public static Complex add( Complex x, double y )`

– **Description**
Returns the sum of a `Complex` and a `double`, $x+y$.

– **Parameters**

* `x` – a `Complex` object
* `y` – a `double` value

– **Returns** – a newly constructed `Complex` initialized to $x+y$

---

- *add*

`public static Complex add( double x, Complex y )`

– **Description**
Returns the sum of a `double` and a `Complex`, $x+y$.

– **Parameters**

* `x` – a `double` value
* `y` – a `Complex` object

– **Returns** – a newly constructed `Complex` initialized to $x+y$

---

- *argument*

`public static double argument( Complex z )`

– **Description**
Returns the argument (phase) of a `Complex`, in radians, with a branch cut along the negative real axis.

– **Parameters**

* `z` – a `Complex` object

– **Returns** – A `double` value equal to the argument (or phase) of a `Complex`. It is in the interval $[-\pi, \pi]$.

---

- *asin*

`public static Complex asin( Complex z )`

– **Description**
Returns the inverse sine (arc sine) of a `Complex`, with branch cuts outside the interval [-1,1] along the real axis. The value of asin is defined in terms of the function asinh, by $\mathrm{asin}(z) = -i\,\mathrm{asinh}(iz)$.

– **Parameters**

* `z` – a `Complex` object

– **Returns** – A newly constructed `Complex` initialized to the inverse (arc) sine of the argument. The real part of the result is in the interval $[-\pi/2, +\pi/2]$.

---

- *asinh*

`public static Complex asinh( Complex z )`

– **Description**

Returns the inverse hyperbolic sine (arc sinh) of a `Complex`, with branch cuts outside the interval [-*i*,*i*].
Specifically, if $z = x+iy$,
$\mathrm{asinh}(\bar{z}) = \overline{\mathrm{asinh}(z)}$ and asinh is odd.
$\mathrm{asinh}(+0 + i0)$ returns $0 + i0$.
$\mathrm{asinh}(\infty + i\infty)$ returns $+\infty + i\pi/4$.
$\mathrm{asinh}(x + i\infty)$ returns $+\infty + i\pi/2$ for positive-signed finite $x$.
$\mathrm{asinh}(+\infty + iy)$ returns $+\infty + i0$ for positive-signed finite $y$.
$\mathrm{asinh}(\mathrm{NaN} + i\infty)$ returns $\pm\infty + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
$\mathrm{asinh}(+\infty + i\mathrm{NaN})$ returns $+\infty + i\mathrm{NaN}$.
$\mathrm{asinh}(\mathrm{NaN} + i0)$ returns $\mathrm{NaN} + i0$.
$\mathrm{asinh}(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite nonzero $y$.
$\mathrm{asinh}(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $x$.
$\mathrm{asinh}(\mathrm{NaN} + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$.

– **Parameters**

* `z` – a `Complex` object

– **Returns** – A newly constructed `Complex` initialized to the inverse (arc) hyperbolic sine of the argument. Its imaginary part is in the interval $[-i\pi/2, i\pi/2]$.

---

- *atan*

`public static Complex atan( Complex z )`

– **Description**

Returns the inverse tangent (arc tangent) of a `Complex`, with branch cuts outside the interval [-*i*,*i*] along the imaginary axis. The value of atan is defined in terms of the function atanh, by $\mathrm{atan}(z) = -i\,\mathrm{atanh}(iz)$.

– **Parameters**

* `z` – a `Complex` object

– **Returns** – A newly constructed `Complex` initialized to the inverse (arc) tangent of the argument. Its real part is in the interval $[-\pi/2, \pi/2]$.

---

- *atanh*
  `public static Complex` **atanh**`( Complex z )`
  - **Description**
    Returns the inverse hyperbolic tangent (arc tanh) of a `Complex`, with branch cuts outside the interval [-1,1] on the real axis.
    Specifically, if $z = x+iy$,
    $\text{atanh}(\bar{z}) = \overline{\text{atanh}(z)}$ and atanh is odd.
    $\text{atanh}(+0 + i0)$ returns $+0 + i0$.
    $\text{atanh}(+\infty + i\infty)$ returns $+0 + i\pi/2$.
    $\text{atanh}(+\infty + iy)$ returns $+0 + i\pi/2$, for finite positive-signed $y$.
    $\text{atanh}(x + i\infty)$ returns $+0 + i\pi/2$, for finite positive-signed $x$.
    $\text{atanh}(+0 + i\text{NaN})$ returns $+0 + i\text{NaN}$.
    $\text{atanh}(\text{NaN} + i\infty)$ returns $\pm 0 + ipi/2$ (where the sign of the real part of the result is unspecified).
    $\text{atanh}(+\infty + i\text{NaN})$ returns $+0 + i\text{NaN}$.
    $\text{atanh}(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$, for finite $y$.
    $\text{atanh}(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$, for nonzero finite $x$.
    $\text{atanh}(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.
  - **Parameters**
    * `z` – a `Complex` object
  - **Returns** – A newly constructed `Complex` initialized to the inverse (arc) hyperbolic tangent of the argument. The imaginary part of the result is in the interval $[-i\pi/2, i\pi/2]$.

---

- *byteValue*
  `public byte` **byteValue**`( )`
  - **Description**
    Returns the value of the real part as a byte.
  - **Returns** – a byte representing the value of the real part of a `Complex` object

---

- *compareTo*
  `public int` **compareTo**`( Complex z )`
  - **Description**
    Compares two `Complex` objects.
    A lexagraphical ordering is used. First the real parts are compared in the sense of Double.compareTo. If the real parts are unequal this is the return value. If the return parts are equal then the comparison of the imaginary parts is returned.
  - **Parameters**

* `z` – a `Complex` to be compared

– **Returns** – The value 0 if z is equal to this `Complex`; a value less than 0 if this `Complex` is less than z; and a value greater than 0 if this `Complex` is greater than z.

---

- *compareTo*

  `public int` **compareTo**`( java.lang.Object` **obj** `)`

  – **Description**

  Compares this `Complex` to another Object. If the Object is a `Complex`, this function behaves like compareTo(`Complex`). Otherwise, it throws a ClassCastException (as `Complex` objects are comparable only to other `Complex` objects).

  – **Parameters**

  * `obj` – an `Object` to be compared

  – **Returns** – an `int`, 0 if obj is equal to this `Complex`; a value less than 0 if this `Complex` is less than obj; and a value greater than 0 if this `Complex` is greater than obj.

  – **Throws**

  * `java.lang.ClassCastException` – is thrown if obj is not a `Complex` object

---

- *conjugate*

  `public static Complex` **conjugate**`( Complex` **z** `)`

  – **Description**

  Returns the complex conjugate of a `Complex` object.

  – **Parameters**

  * `z` – a `Complex` object

  – **Returns** – a newly constructed `Complex` initialized to the complex conjugate of `Complex` argument, z

---

- *cos*

  `public static Complex` **cos**`( Complex` **z** `)`

  – **Description**

  Returns the cosine of a `Complex`. The value of cos is defined in terms of the function cosh, by $\cos(z) = \cosh(iz)$.

  – **Parameters**

  * `z` – a `Complex` object

  – **Returns** – a newly constructed `Complex` initialized to the cosine of the argument

---

- *cosh*

  `public static Complex` **`cosh`**`( Complex` **`z`** `)`

  - **Description**

    Returns the hyperbolic cosh of a `Complex`.
    If $z = x+iy$,
    $\cosh(\bar{z}) = \overline{\cosh(z)}$ and cosh is even.
    $\cosh(+0 + i0)$ returns $1 + i0$.
    $\cosh(+0 + i\infty)$ returns $\mathrm{NaN} \pm i0$ (where the sign of the imaginary part of the result is unspecified).
    $\cosh(+\infty + i0)$ returns $+\infty + i0$.
    $\cosh(+\infty + i\infty)$ returns $+\infty + i\mathrm{NaN}$.
    $\cosh(x + i\infty)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite nonzero $x$.
    $\cosh(+\infty + iy)$ returns $+\infty[\cos(y) + i\sin(y)]$, for finite nonzero $y$.
    $\cosh(+0 + i\mathrm{NaN})$ returns $\mathrm{NaN} \pm i0$ (where the sign of the imaginary part of the result is unspecified).
    $\cosh(+\infty + i\mathrm{NaN})$ returns $+\infty + i\mathrm{NaN}$.
    $\cosh(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite nonzero $x$.
    $\cosh(\mathrm{NaN} + i0)$ returns $\mathrm{NaN} \pm i0$ (where the sign of the imaginary part of the result is unspecified).
    $\cosh(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for all nonzero numbers $y$.
    $\cosh(\mathrm{NaN} + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$.

  - **Parameters**
    * `z` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to the hyperbolic cosine of the argument

- *divide*

  `public static Complex` **`divide`**`( Complex` **`x`**`, Complex` **`y`** `)`

  - **Description**

    Returns the result of a `Complex` object divided by a `Complex` object, $x/y$.
  - **Parameters**
    * `x` – a `Complex` object representing the numerator
    * `y` – a `Complex` object representing the denominator
  - **Returns** – a newly constructed `Complex` initialized to $x/y$

- *divide*

  `public static Complex` **`divide`**`( Complex` **`x`**`, double` **`y`** `)`

  - **Description**

    Returns the result of a `Complex` object divided by a `double`, $x/y$.

- **Parameters**
  - `x` – a `Complex` object representing the numerator
  - `y` – a `double` representing the denominator
- **Returns** – a newly constructed `Complex` initialized to $x/y$

---

- *divide*

  `public static Complex` **divide**`( double` **x**`, Complex` **y** `)`

  - **Description**

    Returns the result of a `double` divided by a `Complex` object, $x/y$.
  - **Parameters**
    - `x` – a `double` value
    - `y` – a `Complex` object representing the denominator
  - **Returns** – a newly constructed `Complex` initialized to $x/y$

---

- *doubleValue*

  `public double` **doubleValue**`( )`

  - **Description**

    Returns the value of the real part as a `double`.
  - **Returns** – a `double` representing the value of the real part of a `Complex` object

---

- *equals*

  `public boolean` **equals**`( Complex` **z** `)`

  - **Description**

    Compares with another `Complex`.

    *Note: To be useful in hashtables this method considers two NaN* `double` *values to be equal. This is not according to IEEE specification.*
  - **Parameters**
    - `z` – a `Complex` object
  - **Returns** – true if the real and imaginary parts of this object are equal to their counterparts in the argument; false, otherwise

---

- *equals*

  `public boolean` **equals**`( java.lang.Object` **obj** `)`

  - **Description**

    Compares this object against the specified object.

    *Note: To be useful in hashtables this method considers two NaN* `double` *values to be equal. This is not according to IEEE specification*
  - **Parameters**
    - `obj` – the object to compare with

- **Returns** – true if the objects are the same; false otherwise

---

- *exp*

  `public static Complex` **exp**`( Complex` **z** `)`

  - **Description**

    Returns the exponential of a `Complex` z, exp(z).
    Specifically, if $z = x+iy$,
    $\exp(\bar{z}) = \overline{\exp(z)}$.
    $\exp(\pm 0 + i0)$ returns $1 + i0$.
    $\exp(+\infty + i0)$ returns $+\infty + i0$.
    $\exp(-\infty + i\infty)$ returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).
    $\exp(+\infty + i\infty)$ returns $\pm\infty + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
    $\exp(x + i\infty)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $x$.
    $\exp(-\infty + iy)$ returns $+0[\cos(y) + i\sin(y)]$, for finite $y$.
    $\exp(+\infty + iy)$ returns $+\infty[\cos(y) + i\sin(y)]$, for finite nonzero $y$.
    $\exp(-\infty + i\mathrm{NaN})$ returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).
    $\exp(+\infty + i\mathrm{NaN})$ returns $\pm\infty + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
    $\exp(\mathrm{NaN} + i0)$ returns $\mathrm{NaN} + i0$.
    $\exp(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for all non-zero numbers $y$.
    $\exp(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $x$.

  - **Parameters**
    * `z` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to the exponential of the argument

---

- *floatValue*

  `public float` **floatValue**`( )`

  - **Description**

    Returns the value of the real part as a float.
  - **Returns** – a float representing the value of the real part of a `Complex` object

---

- *hashCode*

  `public int` **hashCode**`( )`

  - **Description**

    Returns a hashcode for this `Complex`.

- **Returns** – a hash code value for this object

---

- *imag*
  `public double imag( )`
  - **Description**
    Returns the imaginary part of a `Complex` object.
  - **Returns** – a `double` representing the imaginary part of a `Complex` object, z

---

- *imag*
  `public static double imag( Complex z )`
  - **Description**
    Returns the imaginary part of a `Complex` object.
  - **Parameters**
    * `z` – a `Complex` object
  - **Returns** – a `double` representing the imaginary part of the `Complex` object, z

---

- *intValue*
  `public int intValue( )`
  - **Description**
    Returns the value of the real part as an int.
  - **Returns** – an int representing the value of the real part of a `Complex` object

---

- *log*
  `public static Complex log( Complex z )`
  - **Description**
    Returns the logarithm of a `Complex` z, with a branch cut along the negative real axis.
    Specifically, if $z = x{+}iy$,
    $\log(\bar{z}) = \overline{\log(z)}$.
    $\log(0 + i0)$ returns $-\infty + i\pi$.
    $\log(+0 + i0)$ returns $-\infty + i0$.
    $\log(-\infty + i\infty)$ returns $+\infty + i3\pi/4$.
    $\log(+\infty + i\infty)$ returns $+\infty + i\pi/4$.
    $\log(x + i\infty)$ returns $+\infty + i\pi/2$, for finite $x$.
    $\log(-\infty + iy)$ returns $+\infty + i\pi$, for finite positive-signed $y$.
    $\log(+\infty + iy)$ returns $+\infty + i0$, for finite positive-signed $y$.
    $\log(\pm\infty + i\mathrm{NaN})$ returns $+\infty + i\mathrm{NaN}$.
    $\log(\mathrm{NaN} + i\infty)$ returns $+\infty + i\mathrm{NaN}$.
    $\log(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $x$.
    $\log(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $y$.

$\log(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

- **Parameters**
  * `z` – a `Complex` object
- **Returns** – A newly constructed `Complex` initialized to the logarithm of the argument. Its imaginary part is in the interval $[-i\pi, i\pi]$.

---

- *longValue*
`public long` **longValue**`( )`
  - **Description**
  Returns the value of the real part as a long.
  - **Returns** – a long representing the value of the real part of a `Complex` object

---

- *multiply*
`public static Complex` **multiply**`( Complex x, Complex y )`
  - **Description**
  Returns the product of two `Complex` objects, $x$ * $y$.
  - **Parameters**
    * `x` – a `Complex` object
    * `y` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to $x \times y$

---

- *multiply*
`public static Complex` **multiply**`( Complex x, double y )`
  - **Description**
  Returns the product of a `Complex` object and a `double`, $x$ * $y$.
  - **Parameters**
    * `x` – a `Complex` object
    * `y` – a `double` value
  - **Returns** – a newly constructed `Complex` initialized to $x \times y$

---

- *multiply*
`public static Complex` **multiply**`( double x, Complex y )`
  - **Description**
  Returns the product of a `double` and a `Complex` object, $x$ * $y$.
  - **Parameters**
    * `x` – a `double` value
    * `y` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to $x \times y$

- *multiplyImag*
  public static Complex **multiplyImag**( Complex **x**, double **y** )
  - **Description**
    Returns the product of a `Complex` object and a pure imaginary `double`, x * iy.
  - **Parameters**
    * `x` – a `Complex` object
    * `y` – a `double` value representing a pure imaginary
  - **Returns** – a newly constructed `Complex` initialized to x * iy

- *multiplyImag*
  public static Complex **multiplyImag**( double **x**, Complex **y** )
  - **Description**
    Returns the product of a pure imaginary `double` and a `Complex` object, $ix * y$.
  - **Parameters**
    * `x` – a `double` value representing a pure imaginary
    * `y` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to $ix \times y$.

- *negate*
  public static Complex **negate**( Complex **z** )
  - **Description**
    Returns the negative of a `Complex` object, -z.
  - **Parameters**
    * `z` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to the negative of the `Complex` argument, z

- *pow*
  public static Complex **pow**( Complex **x**, Complex **y** )
  - **Description**
    Returns the `Complex` x raised to the `Complex` y power. The value of pow is defined in terms of the functions exp and log, by $\mathrm{pow}(x, y) = \exp(y \log(x))$.
  - **Parameters**
    * `x` – a `Complex` object
    * `y` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to $x^y$.

- *pow*
  public static Complex **pow**( Complex **z**, double **x** )

– **Description**
Returns the `Complex` z raised to the x power, with a branch cut for the first parameter (z) along the negative real axis.

– **Parameters**
  * **z** – a `Complex` object
  * **x** – a `double` value

– **Returns** – a newly constructed `Complex` initialized to z to the power x

---

- *real*
`public double` **real**`( )`

  – **Description**
  Returns the real part of a `Complex` object.

  – **Returns** – a `double` representing the real part of a `Complex` object, z

---

- *real*
`public static double` **real**`( Complex` **z** `)`

  – **Description**
  Returns the real part of a `Complex` object.

  – **Parameters**
    * **z** – a `Complex` object

  – **Returns** – a `double` representing the real part of the `Complex` object, z

---

- *shortValue*
`public short` **shortValue**`( )`

  – **Description**
  Returns the value of the real part as a short.

  – **Returns** – a short representing the value of the real part of a `Complex` object

---

- *sin*
`public static Complex` **sin**`( Complex` **z** `)`

  – **Description**
  Returns the sine of a `Complex`. The value of sin is defined in terms of the function sinh, by $\sin(z) = -i \sinh(iz)$.

  – **Parameters**
    * **z** – a `Complex` object

  – **Returns** – a newly constructed `Complex` initialized to the sine of the argument

---

- *sinh*
`public static Complex` **sinh**`( Complex` **z** `)`

– **Description**
Returns the hyperbolic sine of a `Complex`.
If $z = x+iy$,
$\sinh(\bar{z}) = \overline{\sinh(z)}$ and sinh is odd.
$\sinh(+0 + i0)$ returns $+0 + i0$.
$\sinh(+0 + i\infty)$ returns $\pm 0 + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(+\infty + i0)$ returns $+\infty + i0$.
$\sinh(+\infty + i\infty)$ returns $\pm\infty + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(+\infty + iy)$ returns $+\infty[\cos(y) + i\sin(y)]$, for positive finite $y$.
$\sinh(x + i\infty)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for positive finite $x$.
$\sinh(+0 + i\mathrm{NaN})$ returns $\pm 0 + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(+\infty + i\mathrm{NaN})$ returns $\pm\infty + i\mathrm{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite nonzero $x$.
$\sinh(\mathrm{NaN} + i0)$ returns $\mathrm{NaN} + i0$.
$\sinh(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for all nonzero numbers $y$.
$\sinh(\mathrm{NaN} + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$.

– **Parameters**
  * **z** – a `Complex` object

– **Returns** – a newly constructed `Complex` initialized to the hyperbolic sine of the argument

---

- *sqrt*

`public static Complex` **`sqrt`**`( Complex` **`z`** `)`

– **Description**
Returns the square root of a `Complex`, with a branch cut along the negative real axis.
Specifically, if $z = x+iy$,
$\mathrm{sqrt}(\bar{z}) = \overline{\mathrm{sqrt}(z)}$.
$\mathrm{sqrt}(\pm 0 + i0)$ returns $+0 + i0$.
$\mathrm{sqrt}(-\infty + iy)$ returns $+0 + i\infty$, for finite positive-signed $y$.
$\mathrm{sqrt}(+\infty + iy)$ returns $+\infty + i0$, for finite positive-signed $y$.
$\mathrm{sqrt}(x + i\infty)$ returns $+\infty + i\infty$, for all x (including NaN).
$\mathrm{sqrt}(-\infty + i\mathrm{NaN})$ returns $\mathrm{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).
$\mathrm{sqrt}(+\infty + i\mathrm{NaN})$ returns $+\infty + i\mathrm{NaN}$.
$\mathrm{sqrt}(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$ and optionally raises the invalid exception, for finite $x$.

sqrt(NaN + $iy$) returns NaN + $i$NaN and optionally raises the invalid exception, for finite $y$.
sqrt(NaN + $i$NaN) returns NaN + $i$NaN.

- **Parameters**
  - `z` – a `Complex` object
- **Returns** – A newly constructed `Complex` initialized to square root of z.

---

- *subtract*
`public static Complex` **`subtract`**`( Complex` **`x`**`, Complex` **`y`** `)`
  - **Description**
  Returns the difference of two `Complex` objects, $x$-$y$.
  - **Parameters**
    - `x` – a `Complex` object
    - `y` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to $x$-$y$

---

- *subtract*
`public static Complex` **`subtract`**`( Complex` **`x`**`, double` **`y`** `)`
  - **Description**
  Returns the difference of a `Complex` object and a `double`, $x$-$y$.
  - **Parameters**
    - `x` – a `Complex` object
    - `y` – a `double` value
  - **Returns** – a newly constructed `Complex` initialized to $x$-$y$

---

- *subtract*
`public static Complex` **`subtract`**`( double` **`x`**`, Complex` **`y`** `)`
  - **Description**
  Returns the difference of a `double` and a `Complex` object, $x$-$y$.
  - **Parameters**
    - `x` – a `double` value
    - `y` – a `Complex` object
  - **Returns** – a newly constructed `Complex` initialized to $x$-$y$

---

- *tan*
`public static Complex` **`tan`**`( Complex` **`z`** `)`
  - **Description**
  Returns the tangent of a `Complex`. The value of tan is defined in terms of the function tanh, by $\tan(z) = -i \tanh(iz)$.

– **Parameters**
  * z – a `Complex` object
– **Returns** – a newly constructed `Complex` initialized to the tangent of the argument

---

• *tanh*
`public static Complex tanh( Complex z )`

– **Description**
Returns the hyperbolic tanh of a `Complex`.
If $z = x{+}iy$,
$\tanh(\bar{z}) = \overline{\tanh(z)}$ and tanh is odd.
$\tanh(+0 + i0)$ returns $+0 + i0$.
$\tanh(+\infty + iy)$ returns $1 + i0$, for all positive-signed numbers $y$.
$\tanh(x + i\infty)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $x$.
$\tanh(+\infty + i\mathrm{NaN})$ returns $1 \pm i0$ (where the sign of the imaginary part of the result is unspecified).
$\tanh(\mathrm{NaN} + i0)$ returns $\mathrm{NaN} + i0$.
$\tanh(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for all nonzero numbers $y$.
$\tanh(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite $x$.
$\tanh(\mathrm{NaN} + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$.

– **Parameters**
  * z – a `Complex` object
– **Returns** – a newly constructed `Complex` initialized to the hyperbolic tangent of the argument

---

• *toString*
`public java.lang.String toString( )`

– **Description**
Returns a `String` representation for the specified `Complex`.
– **Returns** – a `String` representation for this object

---

• *valueOf*
`public static Complex valueOf( java.lang.String s ) throws java.lang.NumberFormatException`

– **Description**
Parses a `String` into a `Complex`.
– **Parameters**
  * s – the `String` to be parsed

- **Returns** – a newly constructed `Complex` initialized to the value represented by the `String` argument
- **Throws**
  * `java.lang.NumberFormatException` – if the string does not contain a parsable `Complex` number
  * `java.lang.NullPointerException` – if the input argument is null

### Example: LU Decomposition of a Complex Matrix

The Complex class is used to convert a real matrix to a Complex matrix. An LU decomposition of the matrix is performed and the determinant and condition number of the matrix are obtained.

```
import com.imsl.math.*;

public class ComplexEx1 {
    public static void main(String args[]) throws SingularMatrixException {
        double ar[][] = {
            {1, 3, 3},
            {1, 3, 4},
            {1, 4, 3}
        };
        double br[] = {12, 13, 14};

        Complex a[][] = new Complex[3][3];
        Complex b[] = new Complex[3];

        for (int i = 0; i < 3; i++){
            b[i] = new Complex(br[i]);
            for (int j = 0; j < 3; j++) {
                a[i][j] = new Complex(ar[i][j]);
            }
        }

        // Compute the LU factorization of A
        ComplexLU clu = new ComplexLU(a);

        // Solve Ax = b
        Complex x[] = clu.solve(b);
        System.out.println("The solution is:");
        System.out.println(" ");
        new PrintMatrix("x").print(x);
```

```
        // Find the condition number of A.
        double condition = clu.condition(a);
        System.out.println("The condition number = "+condition);
        System.out.println();

        // Find the determinant of A.
        Complex determinant = clu.determinant();
        System.out.println("The determinant = "+determinant);
    }
}
```

**Output**

```
The solution is:

  x
   0
0  3
1  2
2  1

The condition number = 0.014886731391585757

The determinant = -0.9999999999999998
```

## *class* Physical

Return the value of various mathematical and physical constants. The case of the `String` specifying the name of the physical constant does not matter. The names 'PI', 'Pi', 'pI' and 'pi' are equivalent. The units of the physical constants are in SI units, (meter-kilogram-second). The names allowed are as follows:

| Name | Description | Value | Reference |
|---|---|---|---|
| AMU | Atomic mass unit | 1.6605402E-27 kg | [1] |
| ATM | Standard atm pressure | 1.01325E+5 N/m$^2$ | E[2] |
| AU | Astronomical unit | 1.496E+11 m | [ ] |
| Avogadro | Avogadro's number | 6.0221367E+23 1/mole | [1] |
| Boltzman | Boltzman's constant | 1.380658E-23 J/K | [1] |
| C | Speed of light | 2.997924580E+8 m/s | E[1] |
| Catalan | Catalan's constant | 0.915965... | E[3] |
| E | Base of natural logs | 2.718... | E[3] |
| ElectronCharge | Electron change | 1.60217733E-19 C | [1] |
| ElectronMass | Electron mass | 9.1093897E-31 kg | [1] |
| ElectronVolt | Electron volt | 1.60217733E-19 J | [1] |
| Euler | Euler's constant gamma | 0.577... | E[3] |
| Faraday | Faraday constant | 9.6485309E+4 C/mole | [1] |
| FineStructure | Fine structure | 7.29735308E-3 | [1] |
| Gamma | Euler's constant | 0.577... | E[3] |
| Gas | Gas constant | 8.314510 J/mole/K | [1] |
| Gravity | Gravitational constant | 6.67259E-11 Nm$^2$/kg$^2$ | [1] |
| Hbar | Planck constant / 2 pi | 1.05457266E-34 J*s | [1] |
| PerfectGasVolume | Std vol ideal gas | 2.241383E-2 m$^3$/mole | [*] |
| Pi | Pi | 3.141... | E[3] |
| Planck | Planck's constant h | 6.6260755E-34 J*s | [1] |
| ProtonMass | Proton mass | 1.6726231E-27 kg | [1] |
| Rydberg | Rydberg's constant | 1.0973731534E+7 /m | [1] |
| SpeedLight | Speed of light | 2.997924580E+8 m/s | E[1] |
| StandardGravity | Standard g | 9.80665 m/s$^2$ | E[2] |
| StandardPressure | Standard atm pressure | 1.01325E+5 N/m$^2$ | E[2] |
| StefanBoltzmann | Stefan-Boltzman | 5.67051E-8 W/K$^4$/m$^2$ | [1] |
| WaterTriple | Triple point of water | 2.7316E+2 K | E[2] |

1. Units strings have the form U1*U2*...*Um/V1/.../Vn, where Ui and Vi are the names of basic units or are the names of basic units raised to a power. Examples are, 'METER*KILOGRAM/SECOND', 'M*KG/S', 'METER', or 'M/KG$^2$'. These strings are case insensitive.

2. The basic unit names allowed are as follows.

   Units of time
   day, hour = hr, min = minute, s = sec = second, year

   Units of frequency
   Hertz = Hz

   Units of mass
   AMU, g = gram, lb = pound, ounce = oz, slug

   Units of distance
   Angstrom, AU, ft = feet = foot, in = inch, m = meter = metre, micron, mile, mill, parsec, yard

   Units of area
   acre

   Units of volume
   l = liter = litre

   Units of force
   dyne, N = Newton, poundal

   Units of energy
   BTU(thermochemical), Erg, J = Joule

   Units of work
   W = watt

   Units of pressure
   ATM = atomosphere, bar, Pascal

   Units of temperature
   degC = Celsius, degF = Fahrenheit, degK = Kelvin

   Units of viscosity
   poise, stoke

   Units of charge
   Abcoulomb, C = Coulomb, statcoulomb

   Units of current
   A = ampere, abampere, statampere

   Units of voltage
   Abvolt, V = volt

Units of magnetic induction
T = Tesla, Wb = Weber

Other units
1, farad, mole, Gauss, Henry, Maxwell, Ohm

The following metric prefixes may be used with the above units. Note that the one or two letter prefixes may only be used with one letter unit abbreviations.

A = atto = 1.E-18
F = femto = 1.E-15
P = pico = 1.E-12
N = nano = 1.E-9
U = micro = 1.E-6
M = milli = 1.E-3
C = centi = 1.E-2
D = deci = 1.E-1
DK = deca = 1.E+1
K = kilo = 1.E+3
myria = 1.E+4 (no single letter prefix; M means milli)
mega = 1.E+6 (no single letter prefix; M means milli)
G = giga = 1.E+9
T = tera = 1.E+12

### Declaration

public class com.imsl.math.Physical
**extends** java.lang.Number
**implements** java.io.Serializable, java.lang.Cloneable

### Fields

---

- protected `double` **value**
- protected `int[]` **dim**
- protected static final `int` **LENGTH**
- protected static final `int` **MASS**
- protected static final `int` **TIME**

- protected static final `int` **CURRENT**
- protected static final `int` **TEMPERATURE**

## Constructors

---

- *Physical*
  `public` **Physical**`( )`
  - **Description**
    Constructs a new 0-valued, dimensionless object.

---

- *Physical*
  `public` **Physical**`( double` **value**`, int` **length**`, int` **mass**`, int` **time**`, int` **current**`, int` **temperature** `)`
  - **Description**
    Constructs a new `Physical` object and initializes this object to a `double` value along with `int` values for length, mass, time, current, and temperature.
  - **Parameters**
    * `value` – `double` value to which this object is initialized
    * `length` – `int` value assigned to this object's length
    * `mass` – `int` value assigned to this object's mass
    * `time` – `int` value assigned to this object's time
    * `current` – `int` value assigned to this object's current
    * `temperature` – `int` value assigned to this object's temperature

---

- *Physical*
  `public` **Physical**`( double` **value**`, java.lang.String` **units** `)`
  - **Description**
    Constructs a new `Physical` object and initializes this object to a `double` value.
  - **Parameters**
    * `value` – `double` value to which the copy of the object is initialized
    * `units` – `String` specifying the unit

---

- *Physical*
  `public` **Physical**`( Physical` **copy** `)`
  - **Description**
    Constructs a copy of a `Physical` object.
  - **Parameters**
    * `copy` – `Physical` object from which a copy is made

## Methods

- *add*
`public static Physical` **add**`( Physical` **x**`, Physical` **y** `)`
  - **Description**
    Add two compatible `Physical` objects.
  - **Parameters**
    * `x` – `Physical` object which is to be added
    * `y` – `Physical` object which is to be added
  - **Returns** – `Physical` object which is the sum of x + y
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if x and y are not compatible

- *checkCompatibility*
`public static void` **checkCompatibility**`( Physical` **x**`, Physical` **y** `)`
  - **Description**
    Checks the compatibility of two `Physical` objects.
  - **Parameters**
    * `x` – a `Physical` object
    * `y` – a `Physical` object to be checked against x
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the two `Physical` objects are incompatible

- *constant*
`public static Physical` **constant**`( java.lang.String` **name** `)`
  - **Description**
    Returns the value of a constant, given its name.
  - **Parameters**
    * `name` – is a `String` representing the name of the constant to be returned
  - **Returns** – the `Physical` object containing the value of the constant, in its default units
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when the name given is undefined

- *constant*
  `public static double` **constant**`( java.lang.String` **name**`,`
  `java.lang.String` **units** `)`

  – **Description**
    Returns the value of a constant, given its name, in the specified units.
  – **Parameters**
    * `name` – is a `String` representing the name of the constant to be returned.
    * `units` – is a `String` representing the units in which the constant is to be returned.
  – **Returns** – a `double` containing the value of the constant in the specified units
  – **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the constant name is undefined

---

- *convert*
  `public static Physical` **convert**`( Physical` **pOld**`, java.lang.String`
  **unitsNew** `)`

  – **Description**
    Converts a value to a different set of units.
  – **Parameters**
    * `pOld` – a `Physical` object specifying the value to be converted
    * `unitsNew` – a `String` specifying the units to which `pOld` is to be converted
  – **Returns** – a `Physical` object containing the value of `pOld` converted to the new units
  – **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the new and old units are incompatible

---

- *defineConstant*
  `public static void` **defineConstant**`( java.lang.String` **name**`, Physical`
  **value** `)`

  – **Description**
    Defines a new constant.
  – **Parameters**
    * `name` – a `String` specifying the name of the constant to be defined
    * `value` – a `Physical` object defining the value of the new constant

---

- *definePrefix*
  `public static void` **definePrefix**`( java.lang.String` **name**`, double` **value** `)`

– **Description**
Defines a new prefix.

– **Parameters**

* `name` – a `String` specifying the name of the prefix to be defined
* `value` – is the `double` value of the prefix

---

- *defineUnit*
`public static void` **defineUnit**`( java.lang.String` **name**`, Physical` **value** `)`

– **Description**
Defines a new unit.

– **Parameters**

* `name` – a `String` specifying the name of the unit to be defined
* `value` – a `Physical` object defining the value of one unit in terms of SI units

---

- *divide*
`public static Physical` **divide**`( double` **x**`, Physical` **y** `)`

– **Description**
Divide a `double` by a `Physical` object.

– **Parameters**

* `x` – `double` which is the numerator
* `y` – `Physical` object which is the divisor

– **Returns** – `Physical` object which is the result of x/y

---

- *divide*
`public static Physical` **divide**`( Physical` **x**`, double` **y** `)`

– **Description**
Divide a `Physical` object by a `double`.

– **Parameters**

* `x` – `Physical` object which is the numerator
* `y` – `double` object which is the divisor

– **Returns** – `Physical` object which is the result of x/y

---

- *divide*
`public static Physical` **divide**`( Physical` **x**`, Physical` **y** `)`

– **Description**
Divide two `Physical` objects.

– **Parameters**

* `x` – `Physical` object which is the numerator
* `y` – `Physical` object which is the divisor

– **Returns** – `Physical` object which is the result of x/y

---

- *doubleValue*
  `public double` **doubleValue**`( )`
  - **Description**
    Returns the value of this dimensionless object.
  - **Returns** – the `double` value of the dimensionless object
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the this object is not dimensionless

---

- *floatValue*
  `public float` **floatValue**`( )`
  - **Description**
    Returns the value of this dimensionless object.
  - **Returns** – the `float` value of the dimensionless object
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the this object is not dimensionless

---

- *intValue*
  `public int` **intValue**`( )`
  - **Description**
    Returns the value of this dimensionless object.
  - **Returns** – the `int` value of the dimensionless object
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the this object is not dimensionless

---

- *longValue*
  `public long` **longValue**`( )`
  - **Description**
    Returns the value of this dimensionless object.
  - **Returns** – the `long` value of the dimensionless object
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the this object is not dimensionless

---

- *multiply*
  `public static Physical` **multiply**`( double x, Physical y )`

– **Description**
  Multiply a `double` and a `Physical` object
– **Parameters**
  * `x` – `double` which is to be multiplied
  * `y` – `Physical` object which is to be multiplied
– **Returns** – `Physical` object which is the product of x and y

---

• *multiply*
`public static Physical` **multiply**`( Physical` **x**`, double` **y** `)`

– **Description**
  Multiply a `Physical` object and a `double`
– **Parameters**
  * `x` – `Physical` object which is to be multiplied
  * `y` – `double` which is to be multiplied
– **Returns** – `Physical` object which is the product of x and y

---

• *multiply*
`public static Physical` **multiply**`( Physical` **x**`, Physical` **y** `)`

– **Description**
  Multiply two `Physical` objects.
– **Parameters**
  * `x` – `Physical` object which is to be multiplied
  * `y` – `Physical` object which is to be multiplied
– **Returns** – `Physical` object which is the product of x and y

---

• *negate*
`public static Physical` **negate**`( Physical` **x** `)`

– **Description**
  Negate a `Physical` object.
– **Parameters**
  * `x` – `Physical` object which is to be negated
– **Returns** – `Physical` object which has been negated

---

• *subtract*
`public static Physical` **subtract**`( Physical` **x**`, Physical` **y** `)`

– **Description**
  Subtract two compatible `Physical` objects.
– **Parameters**
  * `x` – `Physical` object

    * `y` – `Physical` object which is to be subtracted from x
  - **Returns** – `Physical` object which is the result of x - y
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if x and y are not compatible

---

- *toString*
  `public java.lang.String` **toString**`( )`
  - **Description**
    Returns a String containing the value and units, if any.
  - **Returns** – a `String` specifying the value and units, if any, of this `Physical` object

---

- *unitsString*
  `public java.lang.String` **unitsString**`( )`
  - **Description**
    Returns a String containing the units only.
  - **Returns** – a `String` specifying the units of this `Physical` object

### Example: The Physical Constants

The value of the physical constant PI is printed.

```
import com.imsl.math.*;

public class PhysicalEx1 {
    public static void main(String args[]) {
        System.out.println("The value of the physical constant PI is " +
        Physical.constant("PI"));
    }
}
```

### Output

```
The value of the physical constant PI is 3.141592653589793
```

## *class* **EpsilonAlgorithm**

The class is used to determine the limit of a sequence of approximations, by means of the Epsilon algorithm of P. Wynn. An estimate of the absolute error is also given. The condensed Epsilon table is computed. Only those elements needed for the computation of the next diagonal are preserved.

### Declaration

```
public class com.imsl.math.EpsilonAlgorithm
extends java.lang.Object
```

### Constructors

---

- *EpsilonAlgorithm*
  `public` **EpsilonAlgorithm**`( )`
  - **Description**
    Initializes an EpsilonAlgorithm with a maximum table size of 50.

---

- *EpsilonAlgorithm*
  `public` **EpsilonAlgorithm**`( int` **maxTableSize** `)`
  - **Description**
    Initializes an EpsilonAlgorithm.
  - **Parameters**
    * `maxTableSize` – The maximum table size.

### Methods

---

- *extrapolate*
  `public double` **extrapolate**`( double` **x** `)`
  - **Description**
    Extrapolates the convergence limit of a sequence.
  - **Parameters**
    * `x` – is the next point in the original series.

– **Returns** – an estimate of the limit of the series.

---

- *getErrorEstimate*
  `public double` **getErrorEstimate**`( )`

  – **Description**
  Returns the current error estimate.

# Chapter 11

# Printing Functions

## Classes



## *class* PrintMatrix

Matrix printing utilities.

### Declaration

public class com.imsl.math.PrintMatrix
**extends** java.lang.Object

### Fields

- public static final `int` **FULL**
  - This flag as the argument to setMatrixType, indicates that the full matrix is to be printed.

- public static final `int` **UPPER_TRIANGULAR**
  - This flag as the argument to setMatrixType, indicates that only the upper triangular elements of the matrix are to be printed. The matrix still must be a rectangular matrix.
- public static final `int` **LOWER_TRIANGULAR**
  - This flag as the argument to setMatrixType, indicates that only the lower triangular elements of the matrix are to be printed. The matrix still must be a rectangular matrix.
- public static final `int` **STRICT_UPPER_TRIANGULAR**
  - This flag as the argument to setMatrixType, indicates that only the strict upper triangular elements of the matrix are to be printed. The matrix still must be a rectangular matrix.
- public static final `int` **STRICT_LOWER_TRIANGULAR**
  - This flag as the argument to setMatrixType, indicates that only the strict lower triangular elements of the matrix are to be printed. The matrix still must be a rectangular matrix.

## Constructors

---

- *PrintMatrix*
  `public` **PrintMatrix**`( )`
  - **Description**
    Creates an instance of the PrintMatrix class.

---

- *PrintMatrix*
  `public` **PrintMatrix**`( java.io.PrintStream` **out** `)`
  - **Description**
    Creates an instance of the PrintMatrix class with the specified PrintStream.
  - **Parameters**
    * `out` – a `PrintStream`

---

- *PrintMatrix*
  `public` **PrintMatrix**`( java.io.PrintStream` **out**`, java.lang.String` **title** `)`
  - **Description**
    Creates a PrintMatrix object with the specified PrintStream and sets its title.

– **Parameters**

  * `out` – a `PrintStream`
  * `title` – a `String` specifying the title

---

• *PrintMatrix*
`public` **`PrintMatrix`**`( java.lang.String` **`title`** `)`

– **Description**
Creates a PrintMatrix object and sets its title.

– **Parameters**

  * `title` – a `String` specifying the title

## Methods

---

• *print*
`public void` **`print`**`( java.lang.Object` **`array`** `)`

– **Description**
Prints an nRows by nColumns matrix with specified format.

– **Parameters**

  * `array` – a two-dimensional, non-empty, rectangular array

---

• *print*
`public void` **`print`**`( PrintMatrixFormat` **`pmf`**`, java.lang.Object` **`array`** `)`

– **Description**
Prints an nRows by nColumns matrix with specified format.

– **Parameters**

  * `pmf` – a `PrintMatrixFormat` matrix format
  * `array` – a two-dimensional, non-empty, rectangular array

---

• *print*
`protected void` **`print`**`( java.lang.String` **`string`** `)`

– **Description**
Print a string. This function can be overridden to print to something other than a PrintStream.

– **Parameters**

  * `string` – the `String` to be printed

---

- *printHTML*
  `public void` **printHTML**( `PrintMatrixFormat` **pmf**, `java.lang.Object` **array**, `int` **nRows**, `int` **nColumns** )

  – **Description**
    Prints an nRows by nColumns matrix with specified format for HTML output.

  – **Parameters**
    * `pmf` – a `PrintMatrixFormat` matrix format
    * `nRows` – an `int` specifying the number of rows in the matrix
    * `nColumns` – an `int` specifying the number of columns in the matrix

---

- *println*
  `protected void` **println**( )

  – **Description**
    Print a newline. This function can be overridden to print to something other than a PrintStream.

---

- *setColumnSpacing*
  `public PrintMatrix` **setColumnSpacing**( `int` **columnSpacing** )

  – **Description**
    Sets the number of spaces between columns. The default value is 2.

  – **Parameters**
    * `columnSpacing` – an `int` specifying the number of spaces between columns, default is 2

  – **Returns** – the `PrintMatrix` object

---

- *setEqualColumnWidths*
  `public PrintMatrix` **setEqualColumnWidths**( `boolean` **equalColumnWidths** )

  – **Description**
    Force all of the columns to have the same width.

  – **Parameters**
    * `equalColumnWidths` – a `boolean` which specifies that all column widths will be equal

  – **Returns** – the `PrintMatrix` object

---

- *setMatrixType*
  `public PrintMatrix` **setMatrixType**( `int` **matrixType** )

  – **Description**
    Set matrix type.

- **Parameters**
  - `matrixType` – `int` specifying the matrix type. Values for matrixType are:

| | |
|---|---|
| 0 | FULL |
| 1 | UPPER_TRIANGULAR |
| 2 | LOWER_TRIANGULAR |
| 3 | STRICT_UPPER_TRIANGULAR |
| 4 | STRICT_LOWER_TRIANGULAR |

- **Returns** – the `PrintMatrix` object

---

- *setPageWidth*
  `public PrintMatrix` **`setPageWidth`**`( int` **`pageWidth`** `)`
  - **Description**
    Sets the page width. The default value is the largest possible integer.
  - **Parameters**
    - `pageWidth` – an `int` specifying the page width, default is the largest possible integer
  - **Returns** – the `PrintMatrix` object

---

- *setTitle*
  `public PrintMatrix` **`setTitle`**`( java.lang.String` **`title`** `)`
  - **Description**
    Sets matrix title
  - **Parameters**
    - `title` – a `String` specifying the title of the matrix
  - **Returns** – the `PrintMatrix` object

### Example: Matrix and PrintMatrix

The 1 norm of a matrix is found using a method from the Matrix class. The matrix is printed using the PrintMatrix class.

```
import com.imsl.math.*;

public class  PrintMatrixEx1 {
    public static void main(String args[]) {
        double nrm1;
        double a[][] = {
            {0., 1., 2., 3.},
            {4., 5., 6., 7.},
            {8., 9., 8., 1.},
```

```
            {6., 3., 4., 3.}
        };

        // Get the 1 norm of matrix a
        nrm1 = Matrix.oneNorm(a);

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Simple Matrix");

        // Print the matrix and its 1 norm
        p.print(a);
        System.out.println("The 1 norm of the matrix is "+nrm1);
    }
}
```

### Output

```
A Simple Matrix
   0  1  2  3
0  0  1  2  3
1  4  5  6  7
2  8  9  8  1
3  6  3  4  3

The 1 norm of the matrix is 20.0
```

## *class* **PrintMatrixFormat**

This class can be used to customize the actions of PrintMatrix. By default, entries are formatted using the default NumberFormat for the current default locale. As of JDK1.3, none of these NumberFormat objects support scientific notation. To enable scientific notation, set the NumberFormat property to null. There is no way to simultaneously support scientific notation and locale-correct formatting.

### Declaration

public class com.imsl.math.PrintMatrixFormat
**extends** java.lang.Object

### Fields

- public static final `int` **BEGIN_MATRIX**
  - This flag as the type argument to format, indicates that the formatting string for beginning a matrix is to be returned.
- public static final `int` **END_MATRIX**
  - This flag as the type argument to format, indicates that the formatting string for ending a matrix is to be returned.
- public static final `int` **BEGIN_COLUMN_LABELS**
  - This flag as the type argument to format, indicates that the formatting string for beginning a column label row is to be returned.
- public static final `int` **END_COLUMN_LABELS**
  - This flag as the type argument to format, indicates that the formatting string for ending a column label row is to be returned.
- public static final `int` **BEGIN_COLUMN_LABEL**
  - This flag as the type argument to format, indicates that the formatting string for ending a column label is to be returned.
- public static final `int` **COLUMN_LABEL**
  - This flag as the type argument to format, indicates that the formatted string for a given column label is to be returned.
- public static final `int` **END_COLUMN_LABEL**
  - This flag as the type argument to format, indicates that the formatting string for ending a column label is to be returned.
- public static final `int` **BEGIN_ROW**
  - This flag as the type argument to format, indicates that the formatting string for beginning a row is to be returned.
- public static final `int` **END_ROW**
  - This flag as the type argument to format, indicates that the formatting string for ending a row is to be returned.
- public static final `int` **BEGIN_ROW_LABEL**
  - This flag as the type argument to format, indicates that the formatting string for beginning a row label is to be returned.

- public static final `int` **ROW_LABEL**
  - This flag as the type argument to format, indicates that the formatted string for a given row label is to be returned.
- public static final `int` **END_ROW_LABEL**
  - This flag as the type argument to format, indicates that the formatting string for ending a row label is to be returned.
- public static final `int` **BEGIN_ENTRY**
  - This flag as the type argument to format, indicates that the formatted string for beginning an entry is to be returned.
- public static final `int` **ENTRY**
  - This flag as the type argument to format, indicates that the formatted string for a given entry is to be returned.
- public static final `int` **END_ENTRY**
  - This flag as the type argument to format, indicates that the formatted string for ending an entry is to be returned.
- protected `java.text.NumberFormat` **numberFormat**
  - The NumberFormat to be used in formatting double and Complex entries.

## Constructor

---

- *PrintMatrixFormat*
  `public` **PrintMatrixFormat**`( )`
  - **Description**
    Constructs a PrintMatrixFormat object.

## Methods

---

- *format*
  `public java.lang.String` **format**`( int` **type**`, java.lang.Object` **entry**`, int` **row**`, int` **col**`, java.text.ParsePosition` **pos** `)`
  - **Description**
    Returns a formatted string.

– **Parameters**

  * `type` – is the type of string requested.

| *type* | *return value* |
|---|---|
| BEGIN_MATRIX | Tag for the beginning of the matrix. |
| END_MATRIX | Tag for the end of the matrix. |
| BEGIN_COLUMN_LABELS | Tag for the beginning of the column labels row. |
| END_COLUMN_LABELS | Tag for the end of the column labels row. |
| BEGIN_COLUMN_LABEL | Tag for the beginning of a column label. |
| END_COLUMN_LABEL | Tag for the end of a column label. |
| COLUMN_LABEL | The label of the specified column. |
| BEGIN_ROW | Tag for the beginning of a row. |
| END_ROW | Tag for the end of a row. |
| BEGIN_ROW_LABEL | Tag for the beginning of a row label. |
| END_ROW_LABEL | Tag for the end of a row label. |
| ROW_LABEL | The label of the specified row. |
| ENTRY | The row-col entry of the matrix |

  * `entry` – is the entry to be formatted. This is only used if type equals ENTRY. For other values of type, this can be set to null.
  * `row` – is the (0-based) row number of the element to be formatted. This is -1 if there is no row number associated with this request.
  * `col` – is the (0-based) column number of the element to be formatted. This is -1 if there is no column number associated with this request.
  * `pos` – is a ParsePosition object used to indicate the alignment center of the return string. This is used only if type==ENTRY. If the entry is a `double` then the index is the position of the decimal point. If the entry is an `int` then the index is the position of the end of the formatted integer.

– **Returns** – is the `String` to be put into the printed table.

---

• *getNumberFormat*
`public java.text.NumberFormat` **getNumberFormat**`( )`

– **Description**
Returns the NumberFormat to be used in formatting double and Complex entries.

---

• *setColumnLabels*
`public void` **setColumnLabels**`( java.lang.String[]` **columnLabels** `)`

– **Description**
Turns on column labeling using the given labels.

- **Parameters**
  - `columnLabels` – is an array of `String`s to be used as column labels. If there are more columns than labels, the labels are reused.

---

- *setFirstColumnNumber*
  `public void` **setFirstColumnNumber**`( int` **firstColumnNumber** `)`
  - **Description**
    Turns on column labeling with index numbers and sets the index for the label of the first column.
  - **Parameters**
    - `firstColumnNumber` – is the number for the first column label. This is usually 0 or 1. The default is 0.

---

- *setFirstRowNumber*
  `public void` **setFirstRowNumber**`( int` **firstRowNumber** `)`
  - **Description**
    Turns on row labeling with index numbers and sets the index for the label of the first row.
  - **Parameters**
    - `firstRowNumber` – is the number for the first row label. This is usually 0 or 1. The default is 0.

---

- *setNoColumnLabels*
  `public void` **setNoColumnLabels**`( )`
  - **Description**
    Turns off column labels.

---

- *setNoRowLabels*
  `public void` **setNoRowLabels**`( )`
  - **Description**
    Turns off row labels.

---

- *setNumberFormat*
  `public void` **setNumberFormat**`( java.text.NumberFormat` **numberFormat**
  `)`
  - **Description**
    Sets the NumberFormat to be used in formatting double and Complex entries.
  - **Parameters**
    - `numberFormat` – a `NumberFormat` or null. If `null` then numbers will be formatted using toString(int), or toString().

### Example: Matrix Formatting

A simple matrix is printed using the default format with the PrintMatrix class. The PrintMatrixFormat class is then used to change the default format.

```
import com.imsl.math.*;
import java.text.*;

public class  PrintMatrixFormatEx1 {
    public static void main(String args[]) {
        double a[][] = {
            {0., 1., 2., 3.},
            {4., 5., 6., 7.},
            {8., 9., 8., 1.},
            {6., 3., 4., 3.}
        };

        // Construct a PrintMatrix object with a title
        PrintMatrix p = new PrintMatrix("A Simple Matrix");

        // Print the matrix
        p.print(a);

        // Turn row and column labels off
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();

        // Print the matrix
        p.print(mf, a);
    }
}
```

### Output

```
A Simple Matrix
   0  1  2  3
0  0  1  2  3
1  4  5  6  7
2  8  9  8  1
3  6  3  4  3
```

```
A Simple Matrix

0  1  2  3
4  5  6  7
8  9  8  1
6  3  4  3
```

# Chapter 12

# Basic Statistics

Classes

## Usage Notes

The methods/classes for the computations of basic statistics generally have relatively simple arguments. Most of the methods/classes in this chapter allow for missing values. Missing value codes can be set by using `Double.NaN`.

Several methods/classes in this chapter perform statistical tests. These methods in the classes generally return a "$p$-value" for the test. The $p$-value is between 0 and 1 and is the probability of observing data that would yield a test statistic as extreme or more extreme under the assumption of the null hypothesis. Hence, a small $p$-value is evidence for the rejection of the null hypothesis.

# *class* Summary

Computes basic univariate statistics.

For the data in `x`. `Summary` computes the sample mean, variance, minimum, maximum, and ther basic statistics. It also computes confidence intervals for the mean and variance if the sample is assumed to be from a normal population.

Missing values, that is, values equal to NaN (not a number), are excluded from the computations. The sum of the weights is used only in computing the mean (of course, then the weighted mean is used in computing the central moments). The definitions of some of the statistics are given below in terms of a single variable $x$. The $i$-th datum is $x_i$, with corresponding weight $w_i$. If weights are not specified, the $w_i$ are identically one. The summation in each case is over the set of valid observations, based on the presence of missing values in the data.

Number of nonmissing observations,

$$n = \sum f_i$$

Mean,

$$\bar{x}_w = \frac{\sum f_i w_i x_i}{\sum f_i w_i}$$

Variance,

$$s_w^2 = \frac{\sum f_i w_i \left(x_i - \bar{x}_w\right)^2}{n-1}$$

Skewness,

$$\frac{\sum f_i w_i \left(x_i - \bar{x}_w\right)^3 / n}{\left[\sum f_i w_i \left(x_i - \bar{x}_w\right)^2 / n\right]^{3/2}}$$

Excess or Kurtosis,

$$\frac{\sum f_i w_i \left(x_i - \bar{x}_w\right)^4 / n}{\left[\sum f_i w_i \left(x_i - \bar{x}_w\right)^2 / n\right]^2} - 3$$

Minimum,

$$x_{\min} = \min(x_i)$$

Maximum,

$$x_{\max} = \max(x_i)$$

## Declaration

public class com.imsl.stat.Summary
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

---

- *Summary*
  `public` **Summary**`( )`
  - **Description**
    Constructs a new summary statistics object.

## Methods

---

- *confidenceMean*
  `public double[]` **confidenceMean**`( double` **p** `)`
  - **Description**
    Returns the confidence interval for the mean (assuming normality).

- **Parameters**
  - `p` – a `double`, the confidence level desired, usually 0.90, 0.95 or 0.99.
- **Returns** – a `double` array of length 2 which contains the lower and upper confidence limits for the mean

---

- *confidenceVariance*
  `public double[]` **confidenceVariance**`( double` **p** `)`
  - **Description**
    Returns the confidence interval for the variance (assuming normality).
  - **Parameters**
    - `p` – a `double`, the confidence level desired, usually 0.90, 0.95 or 0.99.
  - **Returns** – a `double` array of length 2 which contains the lower and upper confidence limits for the variance

---

- *getKurtosis*
  `public double` **getKurtosis**`( )`
  - **Description**
    Returns the kurtosis.
  - **Returns** – a `double` representing the kurtosis

---

- *getMaximum*
  `public double` **getMaximum**`( )`
  - **Description**
    Returns the maximum.
  - **Returns** – a `double` representing the maximum

---

- *getMean*
  `public double` **getMean**`( )`
  - **Description**
    Returns the population mean.
  - **Returns** – a `double` representing the population mean

---

- *getMinimum*
  `public double` **getMinimum**`( )`
  - **Description**
    Returns the minimum.
  - **Returns** – a `double` representing the minimum

---

- *getSampleStandardDeviation*
  `public double` **getSampleStandardDeviation**`( )`
  - **Description**
    Returns the sample standard deviation.
  - **Returns** – a `double` representing the sample standard deviation

---

- *getSampleVariance*
  `public double` **getSampleVariance**`( )`
  - **Description**
    Returns the sample variance.
  - **Returns** – a `double` representing the sample variance

---

- *getSkewness*
  `public double` **getSkewness**`( )`
  - **Description**
    Returns the skewness.
  - **Returns** – a `double` representing the skewness

---

- *getStandardDeviation*
  `public double` **getStandardDeviation**`( )`
  - **Description**
    Returns the population standard deviation.
  - **Returns** – a `double` representing the population standard deviation

---

- *getVariance*
  `public double` **getVariance**`( )`
  - **Description**
    Returns the population variance.
  - **Returns** – a `double` representing the population variance

---

- *kurtosis*
  `public static double` **kurtosis**`( double[] x )`
  - **Description**
    Returns the kurtosis of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose kurtosis is to be found
  - **Returns** – a `double`, the kurtosis of the given data set

---

- *kurtosis*
  `public static double` **kurtosis**`( double[] x, double[]` **weight** `)`
  - **Description**
    Returns the kurtosis of the given data set and associated weights.
  - **Parameters**
    * `x` – a `double` array containing the data set whose kurtosis is to be found
    * `weight` – a `double` array containing the weights associated with the data points x
  - **Returns** – a `double`, the kurtosis of the given data set

---

- *maximum*
  `public static double` **maximum**`( double[] x )`
  - **Description**
    Returns the maximum of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose maximum is to be found
  - **Returns** – a `double`, the maximum of the given data set

---

- *mean*
  `public static double` **mean**`( double[] x )`
  - **Description**
    Returns the mean of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose mean is to be found
  - **Returns** – a `double`, the mean of the given data set

---

- *mean*
  `public static double` **mean**`( double[] x, double[]` **weight** `)`
  - **Description**
    Returns the mean of the given data set with associated weights.
  - **Parameters**
    * `x` – a `double` array containing the data set whose mean is to be found
    * `weight` – a `double` array containing the weights associated with the data points x
  - **Returns** – a `double`, the mean of the given data set

---

- *median*
  `public static double` **median**`( double[] x )`

- **Description**
  Returns the median of the given data set.
- **Parameters**
  * `x` – a `double` array containing the data set whose median is to be found
- **Returns** – a `double`, the median of the given data set

---

- *minimum*
  `public static double` **minimum**`( double[] x )`
  - **Description**
    Returns the minimum of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose minimum is to be found
  - **Returns** – a `double`, the minimum of the given data set

---

- *mode*
  `public static double` **mode**`( double[] x )`
  - **Description**
    Returns the mode of the given data set. Ties are broken at random.
  - **Parameters**
    * `x` – a `double` array containing the data set whose mode is to be found
  - **Returns** – a `double`, the mode of the given data set

---

- *sampleStandardDeviation*
  `public static double` **sampleStandardDeviation**`( double[] x )`
  - **Description**
    Returns the sample standard deviation of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose sample standard deviation is to be found
  - **Returns** – a `double`, the sample standard deviation of the given data set

---

- *sampleStandardDeviation*
  `public static double` **sampleStandardDeviation**`( double[] x, double[]` **weight** `)`
  - **Description**
    Returns the sample standard deviation of the given data set and associated weights.
  - **Parameters**

- * `x` – a `double` array containing the data set whose sample standard deviation is to be found
  * `weight` – a `double` array containing the weights associated with the data points x.
- **Returns** – a `double`, the sample standard deviation of the given data set

---

- *sampleVariance*
  `public static double` **sampleVariance**`( double[]` **x** `)`
  - **Description**
    Returns the sample variance of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose sample variance is to be found
  - **Returns** – a `double`, the sample variance of the given data set

---

- *sampleVariance*
  `public static double` **sampleVariance**`( double[]` **x**`, double[]` **weight** `)`
  - **Description**
    Returns the sample variance of the given data set and associated weights.
  - **Parameters**
    * `x` – a `double` array containing the data set whose sample variance is to be found
    * `weight` – a `double` array containing the weights associated with the data points x
  - **Returns** – a `double`, the sample variance of the given data set

---

- *skewness*
  `public static double` **skewness**`( double[]` **x** `)`
  - **Description**
    Returns the skewness of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose skewness is to be found
  - **Returns** – a `double`, the skewness of the given data set

---

- *skewness*
  `public static double` **skewness**`( double[]` **x**`, double[]` **weight** `)`
  - **Description**
    Returns the skewness of the given data set and associated weights.
  - **Parameters**

* `x` – a `double` array containing the data set whose skewness is to be found
* `weight` – a `double` array containing the weights associated with the data points x

– **Returns** – a `double`, the skewness of the given data set

---

• *standardDeviation*

`public static double` **standardDeviation**`( double[] x )`

– **Description**

Returns the population standard deviation of the given data set.

– **Parameters**

* `x` – a `double` array containing the data set whose standard deviation is to be found

– **Returns** – a `double`, the population standard deviation of the given data set

---

• *standardDeviation*

`public static double` **standardDeviation**`( double[] x, double[]` **weight** `)`

– **Description**

Returns the population standard deviation of the given data set and associated weights.

– **Parameters**

* `x` – a `double` array containing the data set whose standard deviation is to be found
* `weight` – a `double` array containing the weights associated with the data points x

– **Returns** – a `double`, the population standard deviation of the given data set

---

• *update*

`public synchronized void` **update**`( double x )`

– **Description**

Adds an observation to the `Summary` object.

– **Parameters**

* `x` – a `double`, the data observation to be added

---

• *update*

`public synchronized void` **update**`( double[] x )`

– **Description**

Adds a set of observations to the `Summary` object.

– **Parameters**

* `x` – a `double` array of data observations to be added

---

- *update*
  `public synchronized void` **`update`**`( double[] x, double[]` **`weight`** `)`
  - **Description**
    Adds a set of observations and associated weights to the `Summary` object.
  - **Parameters**
    * `x` – a `double` array of data observations to be added
    * `weight` – a `double` array of weights associated with the observations

---

- *update*
  `public synchronized void` **`update`**`( double x, double` **`weight`** `)`
  - **Description**
    Adds an observation and associated weight to the `Summary` object.
  - **Parameters**
    * `x` – a `double`, the data observation to be added
    * `weight` – a `double`, the weight associated with the observation

---

- *variance*
  `public static double` **`variance`**`( double[]` **`x`** `)`
  - **Description**
    Returns the population variance of the given data set.
  - **Parameters**
    * `x` – a `double` array containing the data set whose population variance is to be found
  - **Returns** – a `double`, the population variance of the given data set

---

- *variance*
  `public static double` **`variance`**`( double[] x, double[]` **`weight`** `)`
  - **Description**
    Returns the population variance of the given data set and associated weights.
  - **Parameters**
    * `x` – a `double` array containing the data set whose population variance is to be found
    * `weight` – a `double` array containing the weights associated with the data points x
  - **Returns** – a `double`, the population variance of the given data set

### Example: Summary Statistics

Summary statistics for a small data set are computed.

```
import com.imsl.stat.*;

public class SummaryEx1 {
    static final double data1[] = {3, 6.4, 2, 1.6, -8, 12, -7,
    6.4, 22, 1, 0, -3.2};

    public static void main(String args[]) {
        Summary summary = new Summary();
        summary.update(data1);

        System.out.println("The minimum is "+summary.getMinimum());
        System.out.println();

        System.out.println("The maximum is "+summary.getMaximum());
        System.out.println();

        System.out.println("The mean is "+summary.getMean());
        System.out.println();

        System.out.println("The variance is "+summary.getVariance());
        System.out.println();

        System.out.println("The sample variance is " +
        summary.getSampleVariance());
        System.out.println();

        System.out.println("The standard deviation is " +
        summary.getStandardDeviation());
        System.out.println();

        System.out.println("The skewness is "+summary.getSkewness());
        System.out.println();

        System.out.println("The kurtosis is "+summary.getKurtosis());
        System.out.println();

        double confmn[] = new double[2];
        confmn = summary.confidenceMean(0.95);
        System.out.println("The confidence Mean is {" + confmn[0] +
```

```
        ", " + confmn[1]+"}");
        System.out.println();

        double confvr[] = new double[2];
        confvr = summary.confidenceVariance(0.95);
        System.out.println("The confidence Variance is {" + confvr[0] +
        ", " + confvr[1]+"}");
    }
}
```

### Output

```
The minimum is -8.0

The maximum is 22.0

The mean is 3.016666666666666

The variance is 61.70972222222223

The sample variance is 67.31969696969698

The standard deviation is 7.855553591073148

The skewness is 0.8632224134285833

The kurtosis is 0.5677060483851211

The confidence Mean is {-2.1964514686012353, 8.229784801934567}

The confidence Variance is {33.78261872720627, 194.0685332772439}
```

## *class* Covariances

Computes the sample variance-covariance or correlation matrix.

Class `covariances` computes estimates of correlations, covariances, or sums of squares and crossproducts for a data matrix $x$. Weights and frequencies are allowed but not required.

The means, (corrected) sums of squares, and (corrected) sums of crossproducts are computed using the method of provisional means. Let $x_{ki}$ denote the mean based on $i$ observations for the $k$-th variable, $f_i$ denote the frequency of the $i$-th observation, $w_i$ denote the weight of the $i$-th observations, and $c_{jki}$ denote the sum of crossproducts (or sum of squares if $j = k$) based on $i$ observations. Then the method of provisional means finds new means and sums of crossproducts as shown in the example below.

The means and crossproducts are initialized as follows:

$$x_{k0} = 0.0 \quad for\ k = 1, \ldots, p$$

$$c_{jk0} = 0.0 \ \ for\ j,\ k = 1, \ldots, p$$

where $p$ denotes the number of variables. Letting $x_{k,i+1}$ denote the $k$-th variable of observation $i + 1$, each new observation leads to the following updates for $x_{ki}$ and $c_{jki}$ using the update constant $r_{i+1}$:

$$r_{i+1} = \frac{f_{i+1} w_{i+1}}{\sum_{l=1}^{i+1} f_l w_l}$$

$$\bar{x}_{k,\, i+1} = \bar{x}_{ki} + \left(x_{k,\, i+1} - \bar{x}_{ki}\right) r_{i+1}$$

$$c_{jk,\, i+1} = c_{jki} + f_{i+1} w_{i+1} \left(x_{j,\, i+1} - \bar{x}_{ji}\right) \left(x_{k,\, i+1} - \bar{x}_{ki}\right) \left(1 - r_{i+1}\right)$$

The default value for weights and frequencies is 1. Means and variances are computed based on the valid data for each variable or, if required, based on all the valid data for each pair of variables.

## Declaration

public class com.imsl.stat.Covariances
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* **Covariances.NonnegativeFreqException**

Frequencies must be nonnegative.

#### Declaration

public static class com.imsl.stat.Covariances.NonnegativeFreqException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *Covariances.NonnegativeFreqException*
  `public` **Covariances.NonnegativeFreqException**`(` `java.lang.String` **message** `)`

- *Covariances.NonnegativeFreqException*
  `public` **Covariances.NonnegativeFreqException**`(` `java.lang.String` **key**, `java.lang.Object[]` **arguments** `)`

### *class* Covariances.NonnegativeWeightException

Weights must be nonnegative.

#### Declaration

public static class com.imsl.stat.Covariances.NonnegativeWeightException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *Covariances.NonnegativeWeightException*
  `public` **Covariances.NonnegativeWeightException**`(` `java.lang.String` **message** `)`

- *Covariances.NonnegativeWeightException*
  `public` **Covariances.NonnegativeWeightException**`(` `java.lang.String` **key**, `java.lang.Object[]` **arguments** `)`

### *class* Covariances.TooManyObsDeletedException

More observations have been deleted than were originally entered (the sum of frequencies has become negative).

### Declaration

public static class com.imsl.stat.Covariances.TooManyObsDeletedException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *Covariances.TooManyObsDeletedException*
  `public` **Covariances.TooManyObsDeletedException**`( java.lang.String` **message** `)`
- *Covariances.TooManyObsDeletedException*
  `public` **Covariances.TooManyObsDeletedException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* Covariances.MoreObsDelThanEnteredException

More observations are being deleted from the output covariance matrix than were originally entered (the corresponding row, column of the incidence matrix is less than zero).

### Declaration

public static class com.imsl.stat.Covariances.MoreObsDelThanEnteredException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *Covariances.MoreObsDelThanEnteredException*
  `public` **Covariances.MoreObsDelThanEnteredException**`( java.lang.String` **message** `)`
- *Covariances.MoreObsDelThanEnteredException*
  `public` **Covariances.MoreObsDelThanEnteredException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* Covariances.DiffObsDeletedException

Different observations are being deleted from return matrix than were originally entered.

### Declaration

public static class com.imsl.stat.Covariances.DiffObsDeletedException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *Covariances.DiffObsDeletedException*
  `public` **Covariances.DiffObsDeletedException**`( java.lang.String` **message** `)`
- *Covariances.DiffObsDeletedException*
  `public` **Covariances.DiffObsDeletedException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### Fields

---

- public static final `int` **VARIANCE_COVARIANCE_MATRIX**
  - Indicates variance-covariance matrix.
- public static final `int` **CORRECTED_SSCP_MATRIX**
  - Indicates corrected sums of squares and crossproducts matrix.
- public static final `int` **CORRELATION_MATRIX**
  - Indicates correlation matrix.
- public static final `int` **STDEV_CORRELATION_MATRIX**
  - Indicates correlation matrix except for the diagonal elements which are the standard deviations

### Constructor

---

- *Covariances*
  `public` **Covariances**`( double[][] x )`
  - **Description**
    Constructor for `Covariances`.

– **Parameters**
  * `x` – A `double` matrix containing the data.
– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if `x.length`, and `x[0].length` are equal to 0.

## Methods

---

- *compute*
  `public final double[][]` **compute**`( int` **matrixType** `) throws com.imsl.stat.Covariances.NonnegativeFreqException, com.imsl.stat.Covariances.NonnegativeWeightException, com.imsl.stat.Covariances.TooManyObsDeletedException, com.imsl.stat.Covariances.MoreObsDelThanEnteredException, com.imsl.stat.Covariances.DiffObsDeletedException`
  – **Description**
  Computes the matrix.
  – **Parameters**
    * `matrixType` – An `int` scalar indicating the type of matrix to compute. Uses class member `VARIANCE_COVARIANCE_MATRIX`, `CORRECTED_SSCP_MATRIX`, `CORRELATION_MATRIX`, `STDEV_CORRELATION_MATRIX` for `matrixType`.
  – **Returns** – A `double` matrix containing computed result.
  – **Throws**
    * `com.imsl.stat.Covariances.NonnegativeFreqException` – is thrown if the frequencies are negative.
    * `com.imsl.stat.Covariances.NonnegativeWeightException` – is thrown if the weights sre negative.
    * `com.imsl.stat.Covariances.TooManyObsDeletedException` – is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative.
    * `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – is thrown if more observations are being deleted from “variance-covariance” matrix than were originally entered. The corresponding row,column of the incidence matrix is less than zero.
    * `com.imsl.stat.Covariances.DiffObsDeletedException` – is thrown if different observations are being deleted than were originally entered.

- *getIncidenceMatrix*
  `public int[][]` **getIncidenceMatrix**`( )`

- **Description**
  Returns the incidence matrix.
- **Returns** – An `int` matrix containing the incidence matrix. If `method` is 0, incidence matrix is $1 \times 1$ and contains the number of valid observations; otherwise, incidence matrix is $x\,[0]\,.\text{length} \times x\,[0]\,.\text{length}$ and contains the number of pairs of valid observations used in calculating the crossproducts for covariance.

---

- *getMeans*
  `public double[]` **getMeans**`( )`
  - **Description**
    Returns the means of the variables in `x`.
  - **Returns** – A `double` array containing the means of the variables in `x`. The components of the array correspond to the columns of `x`.

---

- *getNumRowMissing*
  `public int` **getNumRowMissing**`( )`
  - **Description**
    Returns the total number of observations that contain any missing values (`Double.NaN`).
  - **Returns** – An `int` scalar containing the total number of observations that contain any missing values (`Double.NaN`).

---

- *getObservations*
  `public int` **getObservations**`( )`
  - **Description**
    Returns the sum of the frequencies.
  - **Returns** – An `int` scalar containing the sum of the frequencies. If `missingValueMethod` = 0, observations with missing values are not included; otherwise, all observations are included except for observations with missing values for the weight or the frequency.

---

- *getSumOfWeights*
  `public double` **getSumOfWeights**`( )`
  - **Description**
    Returns the sum of the weights of all observations.
  - **Returns** – A `double` scalar containing the sum of the weights of all observations. If `missingValueMethod` = 0, observations with missing values are not included. Otherwise, all observations are included except for observations with missing values for the weight or the frequency.

- *setFrequencies*
  `public void` **setFrequencies**`( double[]` **frequencies** `)`

  – **Description**
    Sets the frequency for each observation.

  – **Parameters**
    * `frequencies` – A `double` array of size `x.length` containing the frequency for each observation. Default: `frequencies[]` = 1.

- *setMissingValueMethod*
  `public void` **setMissingValueMethod**`( int` **missingValueMethod** `)`

  – **Description**
    Sets the method used to exclude missing values in `x` from the computations, where `Double.NaN` is interpreted as the missing value code.

  – **Parameters**
    * `missingValueMethod` – An `int` scalar indicating the method to use. The methods are as follows:

| missingValueMethod | **Action** |
|---|---|
| 0 | The exclusion is listwise, default. (The entire row of x is excluded if any of the values of the row is equal to the missing value code.) |
| 1 | Raw crossproducts are computed from all valid pairs and means, and variances are computed from all valid data on the individual variables. Corrected crossproducts, covariances, and correlations are computed using these quantities. |
| 2 | Raw crossproducts, means, and variances are computed as in the case of method = 1. However, corrected crossproducts and covariances are computed only from the valid pairs of data. Correlations are computed using these covariances and the variances from all valid data. |
| 3 | Raw crossproducts, means, variances, and covariances are computed as in the case of method = 2. Correlations are computed using these covariances, but the variances used are computed from the valid pairs of data. |

---

- *setWeights*
  `public void` **setWeights**`( double[]` **weights** `)`
  - **Description**
    Sets the weight for each observation.
  - **Parameters**
    - `weights` – A `double` array of size `x.length` containing the weight for each observation. Default: `weights[]` = 1.

### Example: Covariances

This example illustrates the use of Covariances class for the first 50 observations in the Fisher iris data (Fisher 1936). Note that the first variable is constant over the first 50 observations.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class CovariancesEx1 {
    public static void main(String args[]) throws Exception {
        double[][] x = {
            {1.0, 5.1, 3.5, 1.4, .2}, {1.0, 4.9, 3.0, 1.4, .2},
            {1.0, 4.7, 3.2, 1.3, .2}, {1.0, 4.6, 3.1, 1.5, .2},
            {1.0, 5.0, 3.6, 1.4, .2}, {1.0, 5.4, 3.9, 1.7, .4},
            {1.0, 4.6, 3.4, 1.4, .3}, {1.0, 5.0, 3.4, 1.5, .2},
            {1.0, 4.4, 2.9, 1.4, .2}, {1.0, 4.9, 3.1, 1.5, .1},
            {1.0, 5.4, 3.7, 1.5, .2}, {1.0, 4.8, 3.4, 1.6, .2},
            {1.0, 4.8, 3.0, 1.4, .1}, {1.0, 4.3, 3.0, 1.1, .1},
            {1.0, 5.8, 4.0, 1.2, .2}, {1.0, 5.7, 4.4, 1.5, .4},
            {1.0, 5.4, 3.9, 1.3, .4}, {1.0, 5.1, 3.5, 1.4, .3},
            {1.0, 5.7, 3.8, 1.7, .3}, {1.0, 5.1, 3.8, 1.5, .3},
            {1.0, 5.4, 3.4, 1.7, .2}, {1.0, 5.1, 3.7, 1.5, .4},
            {1.0, 4.6, 3.6, 1.0, .2}, {1.0, 5.1, 3.3, 1.7, .5},
            {1.0, 4.8, 3.4, 1.9, .2}, {1.0, 5.0, 3.0, 1.6, .2},
            {1.0, 5.0, 3.4, 1.6, .4}, {1.0, 5.2, 3.5, 1.5, .2},
            {1.0, 5.2, 3.4, 1.4, .2}, {1.0, 4.7, 3.2, 1.6, .2},
            {1.0, 4.8, 3.1, 1.6, .2}, {1.0, 5.4, 3.4, 1.5, .4},
            {1.0, 5.2, 4.1, 1.5, .1}, {1.0, 5.5, 4.2, 1.4, .2},
            {1.0, 4.9, 3.1, 1.5, .2}, {1.0, 5.0, 3.2, 1.2, .2},
            {1.0, 5.5, 3.5, 1.3, .2}, {1.0, 4.9, 3.6, 1.4, .1},
            {1.0, 4.4, 3.0, 1.3, .2}, {1.0, 5.1, 3.4, 1.5, .2},
            {1.0, 5.0, 3.5, 1.3, .3}, {1.0, 4.5, 2.3, 1.3, .3},
            {1.0, 4.4, 3.2, 1.3, .2}, {1.0, 5.0, 3.5, 1.6, .6},
            {1.0, 5.1, 3.8, 1.9, .4}, {1.0, 4.8, 3.0, 1.4, .3},
            {1.0, 5.1, 3.8, 1.6, .2}, {1.0, 4.6, 3.2, 1.4, .2},
            {1.0, 5.3, 3.7, 1.5, .2}, {1.0, 5.0, 3.3, 1.4, .2}
        };
        Covariances co = new Covariances(x);
```

```
        PrintMatrix pm =
        new PrintMatrix("Sample Variances-covariances Matrix");

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(4);
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        pm.setMatrixType(PrintMatrix.UPPER_TRIANGULAR);

        pm.print(pmf, co.compute(Covariances.VARIANCE_COVARIANCE_MATRIX));
    }
}
```

### Output

```
    Sample Variances-covariances Matrix
     0       1       2       3       4
0  0.0000  0.0000  0.0000  0.0000  0.0000
1          0.1242  0.0992  0.0164  0.0103
2                  0.1437  0.0117  0.0093
3                          0.0302  0.0061
4                                  0.0111
```

## *class* NormOneSample

Computes statistics for mean and variance inferences using a sample from a normal population.

The statistics for mean and variance inferences are computed by using a sample from a normal population, including methods for the confidence intervals and tests for both mean and variance. The definitions of mean and variance are given below. The summation in each case is over the set of valid observations, based on the presence of missing values in the data.

Method `getMean`, returns value

$$\bar{x} = \frac{\sum x_i}{n}$$

$$\Delta_s^d Z_t$$

Method `getStandardDeviation`, returns value

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

The method `getTTestStat` returns the $t$ statistic for the two-sided test concerning the population mean which is given by

$$t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$$

where $s$ and $\bar{x}$ are given above. This quantity has a $T$ distribution with *n - 1* degrees of freedom. The method `getTTestDF` returns the degree of freedom.

The method `getChiSquaredTestStat` returns the chi-squared statistic for the two-sided test concerning the population variance which is given by

$$\chi^2 = \frac{(n-1)\, s^2}{\sigma_0^2}$$

where $s$ is given above. This quantity has a $\chi^2$ distribution with *n - 1* degrees of freedom. The method `getChiSquaredTestDF` returns the degrees of freedom.

## Declaration

public class com.imsl.stat.NormOneSample
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *NormOneSample*
  `public` **NormOneSample**`( double[] x )`
  - **Description**
    Constructor to compute statistics for mean and variance inferences using a sample from a normal population.
  - **Parameters**
    - `x` – is a one-dimension `double` array containing the observations.

## Methods

- *getChiSquaredTest*
  `public double` **getChiSquaredTest**`( )`
  - **Description**
    Returns the test statistic associated with the chi-squared test for variances. The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in `setChiSquaredTestNull`.
  - **Returns** – a `double` containing the test statistic for the chi-squared test.

- *getChiSquaredTestDF*
  `public int` **getChiSquaredTestDF**`( )`
  - **Description**
    Returns the degrees of freedom associated with the chi-squared test for variances. The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in `setChiSquaredTestNull`.
  - **Returns** – an `int` the degrees of freedom for the chi-squared test.

- *getChiSquaredTestP*
  `public double` **getChiSquaredTestP**`( )`
  - **Description**
    Returns the probability of a larger chi-squared associated with the chi-squared test for variances. The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in `setChiSquaredTestNull`.
  - **Returns** – a `double` containing the probability of a larger chi-squared for the chi-squared test for variances.

- *getLowerCIMean*
  `public double` **getLowerCIMean**`( )`
  - **Description**
    Returns the lower confidence limit for the mean.
  - **Returns** – a `double` containing the lower confidence limit for the mean.

- *getLowerCIVariance*
  `public double` **getLowerCIVariance**`( )`
  - **Description**
    Returns the lower confidence limits for the variance.
  - **Returns** – a `double` containing the lower confidence limits for the variance.

- *getMean*
  `public double` **getMean**`( )`
  - **Description**
    Returns the mean of the sample.
  - **Returns** – a `double` containing the mean.

- *getStdDev*
  `public double` **getStdDev**`( )`
  - **Description**
    Returns the standard deviation of the sample.
  - **Returns** – a `double` containing the standard deviation of the sample.

- *getTTest*
  `public double` **getTTest**`( )`
  - **Description**
    Returns the test statistic associated with the $t$ test. The $t$ test is a test, against a two-sided alternative, of the null hypothesis value described in `setTTestNull`.
  - **Returns** – a `double` containing the test statistic for the $t$ test.

- *getTTestDF*
  `public int` **getTTestDF**`( )`
  - **Description**
    Returns the degrees of freedom associated with the $t$ test for the mean. The $t$ test is a test, against a two-sided alternative, of the null hypothesis value described in `setTTestNull`.
  - **Returns** – an `int` containing the degrees of freedom for the $t$ test.

- *getTTestP*
  `public double` **getTTestP**`( )`
  - **Description**
    Returns the probability associated with the $t$ test of a larger $t$ in absolute value. The $t$ test is a test, against a two-sided alternative, of the null hypothesis value described in `setTTestNull`.
  - **Returns** – a `double` containing the probability for the $t$ test.

- *getUpperCIMean*
  `public double` **getUpperCIMean**`( )`

- **Description**
  Returns the upper confidence limit for the mean.
- **Returns** – a `double` containing the upper confidence limit for the mean.

---

- *getUpperCIVariance*
  `public double` **getUpperCIVariance**`( )`
  - **Description**
    Returns the upper confidence limits for the variance.
  - **Returns** – a `double` the upper confidence limits for the variance.

---

- *setChiSquaredTestNull*
  `public void` **setChiSquaredTestNull**`( double` **chiSqrTestNull** `)`
  - **Description**
    Sets the null hypothesis value for the chi-squared test. The default is 1.0.
  - **Parameters**
    * `chiSqrTestNull` – `double` containing the null hypothesis value for the chi-squared test.

---

- *setConfidenceMean*
  `public void` **setConfidenceMean**`( double` **confidenceMean** `)`
  - **Description**
    Sets the confidence level (in percent) for a two-sided interval estimate of the mean. Argument `confidenceMean` must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. For a one-sided confidence interval with confidence level c (at least 50 percent), set `confidenceMean`=1.0-2.0 * (1.0 - `c`). If the confidence mean is not specified, a 95-percent confidence interval is computed.
  - **Parameters**
    * `confidenceMean` – `double` containing the confidence level of the mean.

---

- *setConfidenceVariance*
  `public void` **setConfidenceVariance**`( double` **confidenceVariance** `)`
  - **Description**
    Sets the confidence level (in percent) for two-sided interval estimate of the variances. Argument `confidenceVariance` must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. For a one-sided confidence interval with confidence level c (at least 50 percent), set `confidenceVariance`=1.0-2.0 * (1.0 - `c`). If the confidence mean is not specified, a 95-percent confidence interval is computed.
  - **Parameters**
    * `confidenceVariance` – `double` containing the confidence level of the variance.

- *setTTestNull*
  `public void` **setTTestNull**`( double` **meanHypothesis** `)`
  - **Description**
    Sets the Null hypothesis value for $t$ test for the mean. `meanHypothesis`=0.0 by default.
  - **Parameters**
    * `meanHypothesis` – `double` containing the hypothesis value.

### Example 1: NormOneSample

This example uses data from Devore (1982, p335), which is based on data published in the *Journal of Materials*. There are 15 observations. The hypothesis $H0 : \mu = 20.0$ is tested. The extremely large $t$ value and the correspondingly small $p$-value provide strong evidence to reject the null hypothesis.

```
import com.imsl.stat.*;

public class NormOneSampleEx1 {
    public static void main(String args[]) {

        double mean, stdev, lomean, upmean;
        int df;
        double t, pvalue;
        double[] x = {
            26.7, 25.8, 24.0, 24.9, 26.4,
            25.9, 24.4, 21.7, 24.1, 25.9,
            27.3, 26.9, 27.3, 24.8, 23.6
        };

        /* Perform Analysis*/

        NormOneSample n1samp = new NormOneSample(x);

        mean = n1samp.getMean();
        stdev = n1samp.getStdDev();
        lomean = n1samp.getLowerCIMean();
        upmean = n1samp.getUpperCIMean();
        n1samp.setTTestNull(20.0);
        df = n1samp.getTTestDF();
        t = n1samp.getTTest();
        pvalue = n1samp.getTTestP();
```

```
        /* Print results */

        System.out.println("Sample Mean = "+ mean);
        System.out.println("Sample Standard Deviation = "+ stdev);
        System.out.println("95% CI for the mean is "+ lomean +"   "+ upmean);
        System.out.println("T Test results");
        System.out.println("df = " + df);
        System.out.println("t = " + t);
        System.out.println("pvalue = " + pvalue);
        System.out.println("");

                /* CI variance */
        double ciLoVar = n1samp.getLowerCIVariance();
        double ciUpVar = n1samp.getUpperCIVariance();
        System.out.println("CI variance is "+ciLoVar+"    "+ciUpVar);
        /*chi-squared test */
        df = n1samp.getChiSquaredTestDF();
        t = n1samp.getChiSquaredTest();
        pvalue = n1samp.getChiSquaredTestP();
        System.out.println("Chi-squared Test results");
        System.out.println("Chi-squared df = " + df);
        System.out.println("Chi-squared t = " + t);
        System.out.println("Chi-squared pvalue = " + pvalue);
    }
}
```

### Output

```
Sample Mean = 25.313333333333336
Sample Standard Deviation = 1.5788181233652814
95% CI for the mean is 24.43901299970965   26.187653666957022
T Test results
df = 14
t = 13.03408619922945
pvalue = 3.2147173811836183E-9

CI variance is 1.3360926049992239    6.199863467239496
```

```
Chi-squared Test results
Chi-squared df = 14
Chi-squared t = 34.897333333333332
Chi-squared pvalue = 0.0015223176141822004
```

## *class* **NormTwoSample**

Computes statistics for mean and variance inferences using samples from two normal populations.

Class `NormTwoSample` computes statistics for making inferences about the means and variances of two normal populations, using independent samples in `x1` and `x2`. For inferences concerning parameters of a single normal population, see class `NormOneSample`.

Let $\mu_1$ and $\sigma_1^2$ be the mean and variance of the first population, and let $\mu_2$ and $\sigma_2^2$ be the corresponding quantities of the second population. The function contains test confidence intervals for difference in means, equality of variances, and the pooled variance.

The means and variances for the two samples are as follows:

$$\bar{x}_1 = \left(\sum x_{1i}/n_1\right), \qquad \bar{x}_2 = \left(\sum x_{2i}\right)/n_2$$

and

$$s_1^2 = \sum \left(x_{1i} - \bar{x}_1\right)^2 / \left(n_1 - 1\right), \qquad s_2^2 = \sum \left(x_{2i} - \bar{x}_2\right)^2 / \left(n_2 - 1\right)$$

**Inferences about the Means**

The test that the difference in means equals a certain value, for example, $\mu_0$, depends on whether or not the variances of the two populations can be considered equal. If the variances are equal and `meanHypothesis` equals 0, the test is the two-sample $t$-test, which is equivalent to an analysis-of-variance test. The pooled variance for the difference-in-means test is as follows:

$$s^2 = \frac{\left(n_1 - 1\right) s_1 + \left(n_2 - 1\right) s_2}{n_1 + n_2 - 2}$$

The $t$ statistic is as follows:

$$t = \frac{\bar{x}_1 - \bar{x}_2 - \mu_0}{s\sqrt{(1/n_1) + (1/n_2)}}$$

Also, the confidence interval for the difference in means can be obtained by first assigning the unequal variances flag to false. This can be done by calling the `setUnequalVariances` method. The confidence interval can then be obtained by the `getLowerCIDiff` and `getUpperCIDiff` methods.

If the population variances are not equal, the ordinary $t$ statistic does not have a $t$ distribution and several approximate tests for the equality of means have been proposed. (See, for example, Anderson and Bancroft 1952, and Kendall and Stuart 1979.) One of the earliest tests devised for this situation is the Fisher-Behrens test, based on Fisher's concept of fiducial probability. A procedure used in the `getTTest`, `getLowerCIDiff` and `getUpperCIDiff` methods assuming unequal variances are specified is the Satterthwaite's procedure, as suggested by H.F. Smith and modified by F.E. Satterthwaite (Anderson and Bancroft 1952, p. 83). Use `setUnequalVariances` true to obtain results assuming unequal variances.

The test statistic is

$$t' = \left(\bar{x}_1 - \bar{x}_2 - \mu_0\right)/s_d$$

where

$$s_d = \sqrt{\left(s_1^2/n_1\right) + \left(s_2^2/n_2\right)}$$

Under the null hypothesis of $\mu_1 - \mu_2 = c$, this quantity has an approximate $t$ distribution with degrees of freedom `df`, given by the following equation:

$$\mathrm{df} = \frac{s_d^4}{\frac{\left(s_1^2/n_1\right)^2}{n_1-1} + \frac{\left(s_2^2/n_2\right)^2}{n_2-1}}$$

**Inferences about Variances**

The $F$ statistic for testing the equality of variances is given by $F = s_{\max}^2/s_{\min}^2$, where $s_{\max}^2$ is the larger of $s_1^2$ and $s_2^2$. If the variances are equal, this quantity has an F distribution with $n_1 - 1$ and $n_2 - 1$ degrees of freedom.

It is generally not recommended that the results of the $F$ test be used to decide whether to use the regular $t$-test or the modified $t'$ on a single set of data. The modified $t'$ (Satterthwaite's procedure) is the more conservative approach to use if there is doubt about the equality of the variances.

### Declaration

public class com.imsl.stat.NormTwoSample
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Constructor

---

- *NormTwoSample*
  `public` **NormTwoSample**`( double[] x, double[] y )`
  - **Description**
    Constructor to compute statistics for mean and variance inferences using samples from two normal populations.
  - **Parameters**
    * `x` – is a `double` array containing the first sample.
    * `y` – is a `double` array containing the second sample.

### Methods

---

- *downdateX*
  `public void` **downdateX**`( double[] x )`
  - **Description**
    Removes the observations in `x` from the first sample.
  - **Parameters**
    * `x` – is a `double` array containing the values to remove from the first sample.

---

- *downdateY*
  `public void` **downdateY**`( double[] y )`
  - **Description**
    Removes the observations in `y` from the second sample.
  - **Parameters**
    * `y` – is a `double` array containing the values to remove from the second sample.

---

- *getChiSquaredTest*
  `public double` **getChiSquaredTest**`( )`

– **Description**
Returns the test statistic associated with the chi-squared test for common, or pooled, variances. The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in `setChiSquaredTestNull`.

– **Returns** – a `double` containing the test statistic for the chi-squared test.

---

- *getChiSquaredTestDF*
`public int` **getChiSquaredTestDF**`( )`

  – **Description**
  Returns the degrees of freedom associated with the chi-squared test for the common, or pooled, variances. The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in `setChiSquaredTestNull`.

  – **Returns** – an `int` containing the degrees of freedom for the chi-squared test.

---

- *getChiSquaredTestP*
`public double` **getChiSquaredTestP**`( )`

  – **Description**
  Returns the probability of a larger chi-squared associated with the chi-squared test for common, or pooled, variances. The chi-squared test is a test of the hypothesis $\omega^2 = \omega_0^2$ where $\omega_0^2$ is the null hypothesis value as described in `setChiSquaredTestNull`.

  – **Returns** – a `double` containing the probability of a larger chi-squared for the chi-squared test for variances.

---

- *getDiffMean*
`public double` **getDiffMean**`( )`

  – **Description**
  Returns the difference in means, mean of `x` - mean of `y`.

  – **Returns** – a `double` containing the difference in mean.

---

- *getFTest*
`public double` **getFTest**`( )`

  – **Description**
  Returns the $F$ test value of the $F$ test for equality of variances.

  – **Returns** – a `double` containing the $F$ test value of the $F$ test for equality of variances.

---

- *getFTestDFdenominator*
`public int` **getFTestDFdenominator**`( )`

– **Description**
Returns the denominator degrees of freedom of the $F$ test for equality of variances.

– **Returns** – a `int` containing the denominator degrees of freedom.

---

- *getFTestDFnumerator*
`public int` **getFTestDFnumerator**`( )`

  – **Description**
  Returns the numerator degrees of freedom of the F test for equality of variances.

  – **Returns** – a `int` containing the numerator degrees of freedom.

---

- *getFTestP*
`public double` **getFTestP**`( )`

  – **Description**
  Returns the probability of a larger $F$ in absolute value for the $F$ test for equality of variances, assuming equal variances.

  – **Returns** – a `double` containing the probability of a larger $F$ in absolute value, assuming equal variances.

---

- *getLowerCICommonVariance*
`public double` **getLowerCICommonVariance**`( )`

  – **Description**
  Returns the lower confidence limits for the common, or pooled, variance.

  – **Returns** – a `double` containing the lower confidence limits for the variance.

---

- *getLowerCIDiff*
`public double` **getLowerCIDiff**`( )`

  – **Description**
  Returns the lower confidence limit for the mean of the first population minus the mean of the second for equal or unequal variances depending on the value set by setUnequalVariances. setUnequalVariances

  – **Returns** – a `double` containing the lower confidence limit for the mean of the first sample minus the mean of the second sample.

---

- *getLowerCIRatioVariance*
`public double` **getLowerCIRatioVariance**`( )`

  – **Description**
  Returns the approximate lower confidence limit for the ratio of the variance of the first population to the second.

– **Returns** – a `double` containing the approximate lower confidence limit variance.

---

- *getMeanX*
  `public double` **getMeanX**`( )`

  – **Description**
  Returns the mean of the first sample, `x`.
  – **Returns** – a `double` containing the mean.

---

- *getMeanY*
  `public double` **getMeanY**`( )`

  – **Description**
  Returns the mean of the second sample, `y`.
  – **Returns** – a `double` containing the mean.

---

- *getPooledVariance*
  `public double` **getPooledVariance**`( )`

  – **Description**
  Returns the Pooled variance for the two samples.
  – **Returns** – a `double` containing the Pooled variance for the two samples.

---

- *getStdDevX*
  `public double` **getStdDevX**`( )`

  – **Description**
  Returns the standard deviation of the first sample.
  – **Returns** – a `double` containing the standard deviation of the first sample.

---

- *getStdDevY*
  `public double` **getStdDevY**`( )`

  – **Description**
  Returns the standard deviation of the second sample.
  – **Returns** – a `double` containing the standard deviation of the second sample.

---

- *getTTest*
  `public double` **getTTest**`( )`

  – **Description**
  Returns the test statistic for the Satterthwaite's approximation. The value returned will be based on assumption of equal or unequal variances based on the the value set by `setUnequalVariances`. setUnequalVariances

– **Returns** – a `double` containing the test statistic for the $t$-test.

---

• *getTTestDF*
`public double` **getTTestDF**`( )`

– **Description**
Returns the degrees of freedom for the Satterthwaite's approximation for $t$-test for either equal or unequal variances, depending on the value set by `setUnequalVariances`. setUnequalVariances

– **Returns** – an `double` containing the degrees of freedom for the $t$-test.

---

• *getTTestP*
`public double` **getTTestP**`( )`

– **Description**
Returns the approximate probability of a larger `t` for the Satterthwaite's approximation for equal or unequal variances. setUnequalVariances

– **Returns** – a `double` containing the probability for the $t$-test.

---

• *getUpperCICommonVariance*
`public double` **getUpperCICommonVariance**`( )`

– **Description**
Returns the upper confidence limits for the common, or pooled, variance.

– **Returns** – a `double` containing the upper confidence limits for the variance.

---

• *getUpperCIDiff*
`public double` **getUpperCIDiff**`( )`

– **Description**
Returns the upper confidence limit for the mean of the first population minus the mean of the second for equal or unequal variances depending on the value set by `setUnequalVariances`. setUnequalVariances

– **Returns** – a `double` containing the upper confidence limit for the mean of the first sample minus the mean of the second sample.

---

• *getUpperCIRatioVariance*
`public double` **getUpperCIRatioVariance**`( )`

– **Description**
Returns the approximate upper confidence limit for the ratio of the variance of the first population to the second.

– **Returns** – a `double` containing the approximate upper confidence limit variance.

- *setChiSquaredTestNull*
  `public void` **setChiSquaredTestNull**( `double` **varianceHypothesisValue** )

  – **Description**
    Sets the null hypothesis value for the chi-squared test. The default is 1.0.
  – **Parameters**
    * `varianceHypothesisValue` – a `double` containing the null hypothesis value for the chi-squared test.

- *setConfidenceMean*
  `public void` **setConfidenceMean**( `double` **confidenceMean** )

  – **Description**
    Sets the confidence level (in percent) for a two-sided interval estimate of the mean of `x` - the mean of `y`, in percent. Argument `confidenceMean` must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. For a one-sided confidence interval with confidence level c (at least 50 percent), set confidenceMean $= 1.0 - 2.0(1.0 - c)$. If the confidence mean is not specified, a 95-percent confidence interval is computed. Default: confidenceMean = .95
  – **Parameters**
    * `confidenceMean` – `double` containing the confidence level of the mean.

- *setConfidenceVariance*
  `public void` **setConfidenceVariance**( `double` **confidenceVariance** )

  – **Description**
    Sets the confidence level (in percent) for two-sided interval estimate of the variances. Under the assumption of equal variances, the pooled variance is used to obtain a two-sided `confidenceVariance` percent confidence interval for the common variance with getLowerCICommonVariance or getUpperCICommonVariance. Without making the assumption of equal variances, setUnequalVariances , the ratio of the variances is of interest. A two-sided `confidenceVariance` percent confidence interval for the ratio of the variance of the first sample to that of the second sample is given by the `getLowerCIRatioVariance` and `getUpperCIRatioVariance`. See setUnequalVariances and getUpperCIRatioVariance. The confidence intervals are symmetric in probability. Argument `confidenceVariance` must be between 0.0 and 1.0 and is often 0.90, 0.95 or 0.99. The default is 0.95.
  – **Parameters**
    * `confidenceVariance` – `double` containing the confidence level of the variance.

- *setTTestNull*
  `public void` **setTTestNull**`( double` **meanHypothesis** `)`
  - **Description**
    Sets the Null hypothesis value for $t$-test for the mean. `meanHypothesis`=0.0 by default.
  - **Parameters**
    * `meanHypothesis` – `double` containing the hypothesis value.

---

- *setUnequalVariances*
  `public void` **setUnequalVariances**`( boolean` **eqVar** `)`
  - **Description**
    Specifies whether to return statistics based on equal or unequal variances. The default is to return statistics for equal variances. if `eqVar` is True then statistics for unequal variances will be returned.
  - **Parameters**
    * `eqVar` – a `boolean` containing a true or false value. A value of true will cause results for unequal variances to be returned. A value of false will cause results for equal variances to be returned.

---

- *update*
  `public void` **update**`( double[]` **x**`, double[]` **y** `)`
  - **Description**
    Concatenates samples x and y to the samples provided in the constructor.
  - **Parameters**
    * `x` – is a `double` array containing updates to the first sample.
    * `y` – is a `double` array containing updates to the second sample.

---

- *updateX*
  `public void` **updateX**`( double[]` **x** `)`
  - **Description**
    Concatenates the values in `x` to the first sample provided in the constructor.
  - **Parameters**
    * `x` – is a `double` array containing updates for the first sample.

---

- *updateY*
  `public void` **updateY**`( double[]` **y** `)`
  - **Description**
    Concatenates the values in `y` to the second sample provided in the constructor.
  - **Parameters**
    * `y` – is a `double` array containing updates for the second sample.

### Example 1: NormTwoSample

This example taken from Conover and Iman(1983, p294), involves scores on arithmetic tests of two grade-school classes.

| Scores for Standard Group | Scores for Experimental Group |
|---|---|
| 72 | 111 |
| 75 | 118 |
| 77 | 128 |
| 80 | 138 |
| 104 | 140 |
| 110 | 150 |
| 125 | 163 |
| | 164 |
| | 169 |

The question is whether a group taught by an experimental method has a higher mean score. The difference in means and the $t$ test are ouput. The variances of the two populations are assumed to be equal. It is seen from the output that there is strong reason to believe that the two means are different ($t$ value of -4.804). Since the lower 97.5-percent confidence limit does not include 0, the null hypothesis is that $\mu_1 \leq \mu_2$ would be rejected at the 0.05 significance level. (The closeness of the values of the sample variances provides some qualitative substantiation of the assumption of equal variances.)

```
import com.imsl.stat.*;

public class NormTwoSampleEx1 {
    public static void main(String args[]) {
        double mean;
        double x1[] = {72.0, 75.0, 77.0, 80.0, 104.0, 110.0, 125.0 };
        double x2[] = {111.0, 118.0, 128.0, 138.0, 140.0, 150.0,
        163.0, 164.0, 169.0 };

        /* Perform Analysis for one sample x2*/
        NormTwoSample n2samp = new NormTwoSample(x1,x2);
        mean = n2samp.getDiffMean();

        System.out.println("x1mean-x2mean  = "+mean);
        System.out.println("X1 mean ="+n2samp.getMeanX() );
        System.out.println("X2 mean ="+n2samp.getMeanY() );

        double pVar = n2samp.getPooledVariance();
        System.out.println("pooledVar = " + pVar);
```

```
        double loCI = n2samp.getLowerCIDiff();
        double upCI = n2samp.getUpperCIDiff();
        System.out.println("95% CI for the mean is " +
        loCI + " " + upCI);

        loCI = n2samp.getLowerCIDiff();
        upCI = n2samp.getUpperCIDiff();
        System.out.println("95% CI for the ueq mean is " +
        loCI + " " + upCI);

        System.out.println("T Test Results");
        double tDF = n2samp.getTTestDF();
        double tT = n2samp.getTTest();
        double tPval = n2samp.getTTestP();
        System.out.println("T default = "+tDF);
        System.out.println("t = "+tT);
        System.out.println("p-value = "+tPval);

        double stdevX = n2samp.getStdDevX();
        double stdevY = n2samp.getStdDevY();
        System.out.println("stdev x1 ="+stdevX);
        System.out.println("stdev x2 ="+stdevY);
    }
}
```

### Output

```
x1mean-x2mean  = -50.476190476190496
X1 mean =91.85714285714285
X2 mean =142.33333333333334
pooledVar = 434.6326530612244
95% CI for the mean is -73.01001962529507 -27.942361327085916
95% CI for the ueq mean is -73.01001962529507 -27.942361327085916
T Test Results
T default = 14.0
t = -4.8043615047163355
p-value = 2.8025836567727923E-4
stdev x1 =20.87605144201182
stdev x2 =20.826665599658526
```

## *class* **Sort**

A collection of sorting functions.

Class `Sort` contains ascending and descending methods for sorting elements of an array or a matrix. The array ascending method sorts the elements of an array, $A$, into ascending order by algebraic value. The array $A$ is divided into two parts by picking a central element $T$ of the array. The first and last elements of $A$ are compared with $T$ and exchanged until the three values appear in the array in ascending order. The elements of the array are rearranged until all elements greater than or equal to the central element appear in the second part of the array and all those less than or equal to the central element appear in the first part. The upper and lower subscripts of one of the segments are saved, and the process continues iteratively on the other segment. When one segment is finally sorted, the process begins again by retrieving the subscripts of another unsorted portion of the array. On completion, $A_j \le A_i$ for $j < i$. For more details, see Singleton (1969), Griffin and Redish (1970), and Petro (1970).

The matrix ascending method sorts the rows of real matrix `x` using a particular row in `x` as the keys. The sort is algebraic with the first key as the most significant, the second key as the next most significant, etc. When x is sorted in ascending order, the resulting sorted array is such that the following is true:

- For $i = 0, 1, \ldots, \text{n\_observations} - 2, \text{x}[i][\text{indices\_keys }[0]] \le \text{x}[i+1][\text{indices\_keys}[0]]$
- For $k = 1, \ldots, \text{n\_keys} - 1, \text{if} \text{x[i][indices\_keys[j]]} = \text{x[i + 1][indices\_keys[j]]}$ for $j = 0, 1, \ldots, k-1$, then $\text{x}[i][\text{indices\_keys}[k]] = \text{x}[i+1][\text{indices\_keys}[k]]$

The observations also can be sorted in descending order.The rows of `x` containing the missing value code NaN in at least one of the specified columns are considered as an additional group. These rows are moved to the end of the sorted `x`. The sorting algorithm is based on a quicksort method given by Singleton (1969) with modifications by Griffen and Redish (1970) and Petro (1970).

All other methods in this class work off of the ascending methods.

### Declaration

public class com.imsl.stat.Sort
**extends** java.lang.Object

### Constructor

- *Sort*
  `public` **`Sort( )`**

### Methods

---

- *ascending*
  `public static void` **`ascending(`** `double[]` **`ra`** `)`
  - **Description**
    Sort an array into ascending order.
  - **Parameters**
    * `ra` – `double` array to be sorted into ascending order

---

- *ascending*
  `public static void` **`ascending(`** `double[][]` **`ra`**, `int` **`nKeys`** `)`
  - **Description**
    Sort a matrix into ascending order by specified keys.
  - **Parameters**
    * `ra` – `double` matrix to be sorted into ascending order.
    * `nKeys` – `int` containing the first `nKeys` columns of `ra` to be used as the sorting keys.

---

- *ascending*
  `public static void` **`ascending(`** `double[][]` **`ra`**, `int[]` **`indkeys`** `)`
  - **Description**
    Sort a matrix into ascending order by specified keys.
  - **Parameters**
    * `ra` – `double` matrix to be sorted into ascending order.
    * `indkeys` – `int` array containing the order the columns of `ra` are to be sorted.

---

- *ascending*
  `public static void` **`ascending(`** `double[][]` **`ra`**, `int[]` **`indkeys`**, `int[]` **`iperm`** `)`
  - **Description**
    Sort a matrix into ascending order by specified keys.
  - **Parameters**
    * `ra` – `double` matrix to be sorted into ascending order.

- * `indkeys` – `int` array containing the order the columns of `ra` are to be sorted.
  * `iperm` – `int` array to be sorted using the same permutations applied to `ra`. Typically, you would initialize this to 0, 1, ...

---

- *ascending*
  `public static void` **`ascending`**`( double[][]` **`ra`**`, int` **`nKeys`**`, int[]` **`iperm`** `)`
  - **Description**
    Sort an array into ascending order by specified keys.
  - **Parameters**
    * `ra` – `double` array to be sorted into ascending order.
    * `nKeys` – `int` containing the first `nKeys` columns of `ra` to be used as the sorting keys.
    * `iperm` – `int` array to be sorted using the same permutations applied to `ra`. Typically, you would initialize this to 0, 1, ...

---

- *ascending*
  `public static void` **`ascending`**`( double[]` **`ra`**`, int[]` **`iperm`** `)`
  - **Description**
    Sort an array into ascending order.
  - **Parameters**
    * `ra` – `double` array to be sorted into ascending order
    * `iperm` – `int` array to be sorted using the same permutations applied to ra. Typically, you would initialize this to 0, 1, ...

---

- *ascending*
  `public static void` **`ascending`**`( int[]` **`ra`** `)`
  - **Description**
    Function to sort an integer array into ascending order.
  - **Parameters**
    * `ra` – `int` array to be sorted into ascending order

---

- *ascending*
  `public static void` **`ascending`**`( int[]` **`ra`**`, int[]` **`iperm`** `)`
  - **Description**
    Sort an array into ascending order.
  - **Parameters**
    * `ra` – `int` array to be sorted into ascending order

* iperm – int array to be sorted using the same permutations applied to ra. Typically, you would initialize this to 0, 1, ...

---

- *descending*
`public static void` **descending**`( double[]` **ra** `)`
  - **Description**
  Sort an array into descending order.
  - **Parameters**
    * ra – double array to be sorted into descending order

---

- *descending*
`public static void` **descending**`( double[][]` **ra**`, int` **nKeys** `)`
  - **Description**
  Function to sort a matrix into descending order by specified keys.
  - **Parameters**
    * ra – double matrix to be sorted into descending order.
    * nKeys – int containing the first nKeys columns of ra to be used as the sorting keys.

---

- *descending*
`public static void` **descending**`( double[][]` **ra**`, int[]` **indkeys** `)`
  - **Description**
  Function to sort a matrix into descending order by specified keys.
  - **Parameters**
    * ra – double matrix to be sorted into descending order.
    * indkeys – int array containing the order the columns of ra are to be sorted.

---

- *descending*
`public static void` **descending**`( double[][]` **ra**`, int[]` **indkeys**`, int[]` **iperm** `)`
  - **Description**
  Function to sort a matrix into descending order by specified keys.
  - **Parameters**
    * ra – double matrix to be sorted into descending order.
    * indkeys – int array containing the order the columns of ra are to be sorted.
    * iperm – int array to be sorted using the same permutations applied to ra. Typically, you would initialize this to 0, 1, ...

---

- *descending*
  `public static void` **descending**`( double[][]` **ra**`, int` **nKeys**`, int[]` **iperm** `)`

  - **Description**
    Function to sort an array into descending order by specified keys.
  - **Parameters**
    * `ra` – `double` array to be sorted into descending order.
    * `nKeys` – `int` containing the first `nKeys` columns of `ra` to be used as the sorting keys.
    * `iperm` – `int` array to be sorted using the same permutations applied to `ra`. Typically, you would initialize this to 0, 1, ...

---

- *descending*
  `public static void` **descending**`( double[]` **ra**`, int[]` **iperm** `)`

  - **Description**
    Sort an array into descending order.
  - **Parameters**
    * `ra` – `double` array to be sorted into descending order
    * `iperm` – an `int` array to be sorted using the same permutations applied to ra. Typically, you would initialize this to 0, 1, ...

### Example 1: Sorting

An array is sorted by increasing value. A permutation array is also computed. Note that the permutation array begins at 0 in this example.

```
import com.imsl.math.*;
import com.imsl.stat.*;

public class SortEx1 {
    public static void main(String args[]) {
        double ra[] = { 10., -9., 8., -7., 6., 5., 4., -3., -2., -1.};
        int iperm[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

        PrintMatrix pm = new PrintMatrix("The Input Array");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        // Print the array
        pm.print(mf, ra);
        System.out.println();
```

```
        //  Sort the array
        Sort.ascending(ra, iperm);

        pm = new PrintMatrix("The Sorted Array - Lowest to Highest");
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();

        // Print the array
        pm.print(mf, ra);

        pm = new PrintMatrix("The Resulting Permutation Array");
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        // Print the array
        pm.print(mf, iperm);
    }
}
```

## Output

```
The Input Array

10
-9
 8
-7
 6
 5
 4
-3
-2
-1


The Sorted Array - Lowest to Highest

-9
-7
-3
```

```
-2
-1
 4
 5
 6
 8
10

The Resulting Permutation Array

1
3
7
8
9
6
5
4
2
0
```

## Example 2: Sorting

The rows of a 10 x 3 matrix x are sorted in ascending order using Columns 0 and 1 as the keys. There are two missing values (NaNs) in the keys. The observations containing these values are moved to the end of the sorted array.

```
import com.imsl.math.*;
import com.imsl.stat.*;

public class SortEx2 {
    public static void main(String args[]) {

        int nKeys=2;
        double x[][] = {{1.0, 1.0, 1.0},
                        {2.0, 1.0, 2.0},
                        {1.0, 1.0, 3.0},
                        {1.0, 1.0, 4.0},
                        {2.0, 2.0, 5.0},
                        {1.0, 2.0, 6.0},
                        {1.0, 2.0, 7.0},
```

```
                {1.0, 1.0, 8.0},
                {2.0, 2.0, 9.0},
                {1.0, 1.0, 9.0}};

        int iperm[] = {0, 1, 2, 3, 4, 5, 6, 7,8,9};
        x[4][1] = Double.NaN;
        x[6][0] = Double.NaN;

        PrintMatrix pm = new PrintMatrix("The Input Array");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        // Print the array
        pm.print(mf, x);
        System.out.println();

        try {
        Sort.ascending(x, nKeys, iperm);
        } catch (Exception e) {

        }

        pm = new PrintMatrix("The Sorted Array - Lowest to Highest");
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();

        // Print the array
        pm.print(mf, x);

        pm = new PrintMatrix("The permutation array");
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        pm.print(mf, iperm);
    }
}
```

## Output

```
The Input Array

1  1  1
2  1  2
1  1  3
1  1  4
2  ?  5
1  2  6
?  2  7
1  1  8
2  2  9
1  1  9


The Sorted Array - Lowest to Highest

1  1  1
1  1  9
1  1  3
1  1  4
1  1  8
1  2  6
2  1  2
2  2  9
?  2  7
2  ?  5

The permutation array

0
9
2
3
7
5
1
8
6
4
```

## *class* **Ranks**

Compute the ranks, normal scores, or exponential scores for a vector of observations.

The class Ranks can be used to compute the ranks, normal scores, or exponential scores of the data in $X$. Ties in the data can be resolved in four different ways, as specified by member function `setTieBreaker`. The type of values returned can vary depending on the member function called:

**GetRanks: Ordinary Ranks**

For this member function, the values output are the ordinary ranks of the data in $X$. If *X[i]* has the smallest value among those in $X$ and there is no other element in $X$ with this value, then `getRanks(i)` = 1. If both *X[i]* and *X[j]* have the same smallest value, then

if *TieBreaker* = 0, *Ranks[i]* = `getRanks([j]` = 1.5<br>
if *TieBreaker* = 1, *Ranks[i]* = *Ranks[j]* = 2.0<br>
if *TieBreaker* = 2, *Ranks[i]* = *Ranks[j]* = 1.0<br>
if *TieBreaker* = 3, *Ranks[i]* = 1.0 and *Ranks[j]* = 2.0<br>
or *Ranks[i]* = 2.0 and *Ranks[j]* = 1.0.

When the ties are resolved by use of function `setRandom`, different results may occur when running the same program at different times unless the "seed" of the random number generator is set explicitly by use of `Random` method `setSeed`. Ordinarily, there is no need to call the routine to set the seed, even if there are ties in the data.

**getBlomScores: Normal Scores, Blom Version**

Normal scores are expected values, or approximations to the expected values, of order statistics from a normal distribution. The simplest approximations are obtained by evaluating the inverse cumulative normal distribution function, `inverseNormal`, at the ranks scaled into the open interval (0, 1). In the Blom version (see Blom 1958), the scaling transformation for the rank $r_i(1 \le r_i \le n$, where $n$ is the sample size is $(r_i - 3/8)/(n + 1/4)$. The Blom normal score corresponding to the observation with rank $r_i$ is

$$\Phi^{-1}\left(\frac{r_i - 3/8}{n + 1/4}\right)$$

where $\Phi(\cdot)$ is the normal cumulative distribution function.

Adjustments for ties are made after the normal score transformation. That is, if *X[i]* equals *X[j]* (within fuzz) and their value is the $k$-th smallest in the data set, the Blom normal scores are determined for ranks of $k$ and $k + 1$, and then these normal scores are averaged or selected in the manner specified by *TieBreaker*, which is set by the method `setTieBreaker`. (Whether the transformations are made first or ties are resolved first makes no difference except when averaging is done.)

### getTukeyScores: Normal Scores, Tukey Version

In the Tukey version (see Tukey 1962), the scaling transformation for the rank $r_i$ is $(r_i - 1/3)/(n + 1/3)$. The Tukey normal score corresponding to the observation with rank $r_i$ is

$$\Phi^{-1}\left(\frac{r_i - 1/3}{n + 1/3}\right)$$

Ties are handled in the same way as discussed above for the Blom normal scores.

### getVanDerWaerdenScores: Normal Scores, Van der Waerden Version

In the Van der Waerden version (see Lehmann 1975, page 97), the scaling transformation for the rank $r_i$ is $r_i/(n + 1)$. The Van der Waerden normal score corresponding to the observation with rank $r_i$ is

$$\Phi^{-1}\left(\frac{r_i}{n + 1}\right)$$

Ties are handled in the same way as discussed above for the Blom normal scores.

### getNormalScores: Expected Value of Normal Order Statistics

The method `getNormalScores` returns the expected values of the normal order statistics. If the value in *X[i]* is the $k$-th smallest, then the value `getNormalScores[i]` is $E(Z_k)$, where $E(\cdot)$ is the expectation operator and $Z_k$ is the $k$-th order statistic in a sample of size `NOBS` from a standard normal distribution. Ties are handled in the same way as discussed above for the Blom normal scores.

### getSavageScores: Savage Scores

The method `getSavageScores` returns the expected values of the exponential order statistics. These values are called Savage scores because of their use in a test discussed by Savage (1956) (see Lehman 1975). If the value in *X[i]* is the $k$-th smallest, then the $i$-th output value output is $E(Y_k)$, where $Y_k$ is the $k$-th order statistic in a sample of size $n$ from a standard exponential distribution. The expected value of the $k$-th order statistic from an exponential sample of size $n$ is

$$\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-k+1}$$

Ties are handled in the same way as discussed above for the Blom normal scores.

## Declaration

```
public class com.imsl.stat.Ranks
extends java.lang.Object
```

## Fields

- public static final `int` **TIE_AVERAGE**
  - In case of ties, use the average of the scores of the tied observations.
- public static final `int` **TIE_HIGHEST**
  - In case of ties, use the highest score in the group of ties.
- public static final `int` **TIE_LOWEST**
  - In case of ties, use the lowest score in the group of ties.
- public static final `int` **TIE_RANDOM**
  - In case of ties, use one of the group of ties chosen at random.

## Constructor

- *Ranks*
  `public` **Ranks**`( )`
  - **Description**
    Constructor for the Ranks class.

## Methods

- *expectedNormalOrderStatistic*
  `public static double` **expectedNormalOrderStatistic**`( int` **i**`, int` **n** `)`
  - **Description**
    Returns the expected value of a normal order statistic.

– **Parameters**
  * `i` – an `int`, the rank of the order statistic
  * `n` – an `int`, the sample size
– **Returns** – a `double`, the expected value of the i-th order statistic in a sample of size n from the standard normal distribution

---

• *getBlomScores*
`public double[]` **getBlomScores**`( double[] x )`

– **Description**
  Gets the Blom version of normal scores for each observation.
– **Parameters**
  * `x` – a `double` array which contains the observations to be ranked
– **Returns** – a `double` array which contains the Blom version of normal scores for each observation in x

---

• *getNormalScores*
`public double[]` **getNormalScores**`( double[] x )`

– **Description**
  Gets the expected value of normal order statistics (for tied observations, the average of the expected normal scores).
– **Parameters**
  * `x` – a `double` array which contains the observations
– **Returns** – a `double` array which contains the expected value of normal order statistics for the observations in x (for tied observations, the average of the expected normal scores)

---

• *getRanks*
`public double[]` **getRanks**`( double[] x )`

– **Description**
  Gets the rank for each observation.
– **Parameters**
  * `x` – a `double` array which contains the observations to be ranked
– **Returns** – a `double` array which contains the rank for each observation in x

---

• *getSavageScores*
`public double[]` **getSavageScores**`( double[] x )`

– **Description**
  Gets the Savage scores (the expected value of exponential order statistics).
– **Parameters**

∗ `x` – a `double` array which contains the observations

– **Returns** – a `double` array which contains the Savage scores for the observations in x. (the expected value of exponential order statistics)

---

• *getTukeyScores*
`public double[]` **getTukeyScores**`( double[] x )`

– **Description**
Gets the Tukey version of normal scores for each observation.

– **Parameters**

∗ `x` – a `double` array which contains the observations to be ranked

– **Returns** – a `double` array which contains the Tukey version of normal scores for each observation in x

---

• *getVanDerWaerdenScores*
`public double[]` **getVanDerWaerdenScores**`( double[] x )`

– **Description**
Gets the Van der Waerden version of normal scores for each observation.

– **Parameters**

∗ `x` – a `double` array which contains the observations to be ranked

– **Returns** – a `double` array which contains the Van der Waerden version of normal scores for each observation in x

---

• *setFuzz*
`public void` **setFuzz**`( double` **fuzz** `)`

– **Description**
Sets the fuzz factor used in determining ties.

– **Parameters**

∗ `fuzz` – a `double` which represents the fuzz factor

---

• *setRandom*
`public void` **setRandom**`( java.util.Random` **random** `)`

– **Description**
Sets the `Random` object.

– **Parameters**

∗ `random` – a `Random` object used in breaking ties

---

• *setTieBreaker*
`public void` **setTieBreaker**`( int` **iTie** `)`

- **Description**
  Sets the tie breaker for Ranks.
- **Parameters**
  * `iTie` – an `int` which represents the tie breaker

### Example: Ranks

In this data from Hinkley (1977) note that the fourth and sixth observations are tied and that the third and twentieth are tied.

```
import com.imsl.stat.*;
import com.imsl.math.*;

public class RanksEx1 {
    public static void main(String args[]) {
        double x[] = {
            0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43,
            3.37, 2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62,
            1.31, 0.32, 0.59, 0.81, 2.81, 1.87, 1.18, 1.35,
            4.75, 2.48, 0.96, 1.89, 0.90, 2.05};

            PrintMatrixFormat mf = new PrintMatrixFormat();
            mf.setNoRowLabels();
            mf.setNoColumnLabels();

            Ranks ranks = new Ranks();
            double score[] = ranks.getRanks(x);
            new PrintMatrix("The Ranks of the Observations - " +
            "Ties Averaged").print(mf, score);
            System.out.println();

            ranks = new Ranks();
            ranks.setTieBreaker(ranks.TIE_HIGHEST);
            score = ranks.getBlomScores(x);
            new PrintMatrix("The Blom Scores of the Observations - " +
            "Highest Score used in Ties").print(mf, score);
            System.out.println();

            ranks = new Ranks();
            ranks.setTieBreaker(ranks.TIE_LOWEST);
            score = ranks.getTukeyScores(x);
            new PrintMatrix("The Tukey Scores of the Observations - " +
            "Lowest Score used in Ties").print(mf, score);
```

```
            System.out.println();

            ranks = new Ranks();
            ranks.setTieBreaker(ranks.TIE_RANDOM);
            Random random = new Random();
            random.setSeed(123457);
            random.setMultiplier(16807);
            ranks.setRandom(random);
            score = ranks.getVanDerWaerdenScores(x);
            new PrintMatrix("The Van Der Waerden Scores of the " +
            "Observations - Ties untied by Random").print(mf, score);
    }
}
```

## Output

```
The Ranks of the Observations - Ties Averaged

 5
18
 6.5
11.5
21
11.5
 2
15
29
24
27
28
16
23
 3
17
13
 1
 4
 6.5
26
19
10
```

```
14
30
25
 9
20
 8
22


The Blom Scores of the Observations - Highest Score used in Ties

-1.024
 0.209
-0.776
-0.294
 0.473
-0.294
-1.61
-0.041
 1.61
 0.776
 1.176
 1.361
 0.041
 0.668
-1.361
 0.125
-0.209
-2.04
-1.176
-0.776
 1.024
 0.294
-0.473
-0.125
 2.04
 0.893
-0.568
 0.382
-0.668
 0.568
```

```
The Tukey Scores of the Observations - Lowest Score used in Ties

-1.02
 0.208
-0.89
-0.381
 0.471
-0.381
-1.599
-0.041
 1.599
 0.773
 1.171
 1.354
 0.041
 0.666
-1.354
 0.124
-0.208
-2.015
-1.171
-0.89
 1.02
 0.293
-0.471
-0.124
 2.015
 0.89
-0.566
 0.381
-0.666
 0.566


The Van Der Waerden Scores of the Observations - Ties untied by Random

-0.989
 0.204
-0.753
-0.287
 0.46
```

```
-0.372
-1.518
-0.04
 1.518
 0.753
 1.131
 1.3
 0.04
 0.649
-1.3
 0.122
-0.204
-1.849
-1.131
-0.865
 0.989
 0.287
-0.46
-0.122
 1.849
 0.865
-0.552
 0.372
-0.649
 0.552
```

# *class* TableOneWay

Tallies observations into a one-way frequency table.

## Declaration

public class com.imsl.stat.TableOneWay
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Constructor

---

- *TableOneWay*
  `public` **TableOneWay**`( double[] x, int` **nIntervals** `)`
  - **Description**
    Constructor for `TableOneWay`.
  - **Parameters**
    * `x` – A `double` array containing the observations.
    * `nIntervals` – An `int` scalar containing the number of intervals (bins).

### Methods

---

- *getFrequencyTable*
  `public double[]` **getFrequencyTable**`( )`
  - **Description**
    Returns the one-way frequency table. `nIntervals` intervals of equal length are used with the initial interval starting with the minimum value in `x` and the last interval ending with the maximum value in `x`. The initial interval is closed on the left and the right. The remaining intervals are open on the left and the closed on the right. Each interval is of length `(max-min)/nIntervals`, where `max` is the maximum value of x and `min` is the minimum value of `x`.
  - **Returns** – `double` array containing the one-way frequency table.

---

- *getFrequencyTable*
  `public double[]` **getFrequencyTable**`( double` **lower_bound**`, double` **upper_bound** `)`
  - **Description**
    Returns a one-way frequency table using known bounds. The one-way frequency table is computed using two semi-infinite intervals as the initial and last intervals. The initial interval is closed on the right and includes `lower_bound` as its right endpoint. The last interval is open on the left and includes all values greater than `upper_bound`. The remaining `nIntervals - 2` intervals are each of length `(upper_bound - lower_bound)/ (nIntervals - 2)` and are open on the left and closed on the right. `nIntervals` must be greater than or equal to 3.
  - **Parameters**
    * `lower_bound` – `double` specifies the right endpoint.

* upper_bound – double specifies the left endpoint.

– **Returns** – double array containing the one-way frequency table.

---

* *getFrequencyTableUsingClassmarks*
  public double[] **getFrequencyTableUsingClassmarks**( double[] **classmarks** )

  – **Description**
  Returns the one-way frequency table using class marks. Equally spaced class marks in ascending order must be provided in the array classmarks of length nIntervals. The class marks are the midpoints of each of the nIntervals. Each interval is assumed to have length classmarks[1] - classmarks[0]. nIntervals must be greater than or equal to 2.

  – **Parameters**

  * classmarks – double array containing either the cutpoints or the class marks.

  – **Returns** – double array containing the one-way frequency table.

---

* *getFrequencyTableUsingCutpoints*
  public double[] **getFrequencyTableUsingCutpoints**( double[] **cutpoints** )

  – **Description**
  Returns the one-way frequency table using cutpoints. The cutpoints are boundaries that must be provided in the array cutpoints of length nIntervals-1. This option allows unequal interval lengths. The initial interval is closed on the right and includes the initial cutpoint as its right endpoint. The last interval is open on the left and includes all values greater than the last cutpoint. The remaining nIntervals-2 intervals are open on the left and closed on the right. Argument nIntervals must be greater than or equal to 3 for this option.

  – **Parameters**

  * cutpoints – double array containing the cutpoints.

  – **Returns** – double array containing the one-way frequency table.

---

* *getMaximum*
  public double **getMaximum**( )

  – **Description**
  Returns maximum value of x.

  – **Returns** – a double containing the maximum data bound.

---

- *getMinimum*
  `public double` **getMinimum**`( )`

  - **Description**
    Returns the minimum value of `x`.
  - **Returns** – a `double` containing the minimum data bound.

### Example: TableOneWay

The data for this example is from Hinkley (1977) and Belleman and Hoaglin (1981). The measurement (in inches) are for precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

The first test uses the default tally method which may be appropriate when the range of data is unknown. The minimum and maximum data bounds are displayed.

The second test computes the table usings known bounds, where the lower bound is 0.5 and the upper bound is 4.5. The eight interior intervals each have width (4.5 - 0.5)/(10-2) = 0.5. The 10 intervals are $(-\infty, 0.5]$, (0.5,1.0],...,(4.0,4.5], and $(4.5,\infty]$.

In the third test, 10 class marks, 0.25, 0.75, 1.25,...,4.75, are input. This defines the class intervals (0.0,0.5],(0.5,1.0],...,(4.0,4.5],(4.5,5.0]. Note that unlike the previous test, the initial and last intervals are the same length as the remaining intervals.

In the fourth test, cutpoints, 0.5,1.0, 1.5, 2.0, ...,4.5, are input to define the same 10 intervals as in the second test. Here again, the initial and last intervals are semi- infinite intervals.

```
import com.imsl.stat.*;

public class  TableOneWayEx1 {
    public static void main(String args[]) {

        int nIntervals=10;
        double table[];

        double[] x={
            0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
            2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32,
            0.59, 0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96,
            1.89, 0.9, 2.05
        };
        double cutPoints[] = { 0.5, 1.0, 1.5, 2.0, 2.5,
        3.0, 3.5, 4.0, 4.5};
        double classMarks[] = {0.25, 0.75, 1.25, 1.75, 2.25,
```

```
        2.75, 3.25, 3.75, 4.25, 4.75};

        TableOneWay fTbl = new TableOneWay(x, nIntervals);
        //double[] table = new double[nIntervals];

        table = fTbl.getFrequencyTable();

        System.out.println("Example 1 ");
        for (int i=0; i < table.length; i++)
            System.out.println(i+"          "+table[i]);

        System.out.println("-------------------------");
        System.out.println("Lower bounds= "+fTbl.getMinimum());
        System.out.println("Upper bounds= "+fTbl.getMaximum());
        System.out.println("-------------------------");
        /*  getFrequencyTable using a set of known bounds */
        table = fTbl.getFrequencyTable(0.5, 4.5);
        for (int i=0; i < table.length; i++)
            System.out.println(i+"          "+table[i]);

        System.out.println("--------------------");

        table = fTbl.getFrequencyTableUsingClassmarks(classMarks);
        for (int i=0; i < table.length; i++)
            System.out.println(i+"          "+table[i]);

        System.out.println("-------------------");
        table = fTbl.getFrequencyTableUsingCutpoints(cutPoints);
        for (int i=0; i < table.length; i++)
            System.out.println(i+"          "+table[i]);
    }
}
```

## Output

```
Example 1
0          4.0
1          8.0
2          5.0
3          5.0
4          3.0
```

```
5          1.0
6          3.0
7          0.0
8          0.0
9          1.0
-------------------------
Lower bounds= 0.32
Upper bounds= 4.75
-------------------------
0          2.0
1          7.0
2          6.0
3          6.0
4          4.0
5          2.0
6          2.0
7          0.0
8          0.0
9          1.0
--------------------
0          2.0
1          7.0
2          6.0
3          6.0
4          4.0
5          2.0
6          2.0
7          0.0
8          0.0
9          1.0
-------------------
0          2.0
1          7.0
2          6.0
3          6.0
4          4.0
5          2.0
6          2.0
7          0.0
8          0.0
9          1.0
```

## *class* **TableTwoWay**

Tallies observations into a two-way frequency table.

### Declaration

public class com.imsl.stat.TableTwoWay
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Constructor

- *TableTwoWay*
  `public` **TableTwoWay**`( double[] x, int` **xIntervals**`, double[] y, int` **yIntervals** `)`
  - **Description**
    Constructor for `TableTwoWay`.
  - **Parameters**
    * `x` – A `double` array containing the data for the first variable.
    * `xIntervals` – An `int` scalar containing the number of intervals (bins) for variable x.
    * `y` – A `double` array containing the data for the second variable.
    * `yIntervals` – An `int` scalar containing the number of intervals (bins) for variable y.

### Methods

- *getFrequencyTable*
  `public double[][]` **getFrequencyTable**`( )`
  - **Description**
    Returns the two-way frequency table. Intervals of equal length are used. Let `xmin` and `xmax` be the minimum and maximum values in x, respectively, with similiar meanings for `ymin` and `ymax`. Then, the first row of the output table is the tally of observations with the x value less than or equal to `xmin + (xmax - xmin)/xIntervals`, and the y value less than or equal to `ymin + (ymax - ymin)/yIntervals`.

– **Returns** – A two-dimensional `double` array containing the two-way frequency table.

---

- *getFrequencyTable*
`public double[][]` **getFrequencyTable**`( double` **xLowerBound**`, double` **xUpperBound**`, double` **yLowerBound**`, double` **yUpperBound** `)`

  – **Description**
  Compute a two-way frequency table using intervals of equal length and user supplied upper and lower bounds, `xLowerBound, xUpperBound, yLowerBound, yUpperBound`. The first and last intervals for both variables are semi-infinite in length. `xIntervals` and `yIntervals` must be greater than or equal to 3.

  – **Parameters**
    * `xLowerBound` – `double` specifies the right endpoint for `x`.
    * `xUpperBound` – `double` specifies the left endpoint for `x`.
    * `yLowerBound` – `double` specifies the right endpoint for `y`.
    * `yUpperBound` – `double` specifies the left endpoint for `y`.

  – **Returns** – A two dimensional `double` array containing the two-way frequency table.

---

- *getFrequencyTableUsingClassmarks*
`public double[][]` **getFrequencyTableUsingClassmarks**`( double[]` **cx**`, double[]` **cy** `)`

  – **Description**
  Returns the two-way frequency table using either cutpoints or class marks. Cutpoints are boundaries and class marks are the midpoints of `xIntervals` and `yIntervals`. Equally spaced class marks in ascending order must be provided in the arrays `cx` and `cy`. The class marks are the midpoints of each interval. Each interval is taken to have length `cx[1] - cx[0]` in the x direction and `cy[1] - cy[0]` in the y direction. The total number of elements in the output table may be less than the number of observations of input data. Arguments `xIntervals` and `yIntervals` must be greater than or equal to 2 for this option.

  – **Parameters**
    * `cx` – `double` array containing either the cutpoints or the class marks for `x`.
    * `cy` – `double` array containing either the cutpoints or the class marks for `y`.

  – **Returns** – A two dimensional `double` array containing the two-way frequency table.

---

- *getFrequencyTableUsingCutpoints*
`public double[][]` **getFrequencyTableUsingCutpoints**`( double[]` **cx**`, double[]` **cy** `)`

– **Description**
Returns the two-way frequency table using cutpoints. The cutpoints (boundaries) must be provided in the arrays `cx` and `cy`, of length (`xIntervals-1`) and (`yIntervals-1`) respectively. The first row of the output table is the tally of observations for which the x value is less than or equal to `cx[0]`, and the y value is less than or equal to `cy[0]`. This option allows unequal interval lengths. Arguments `cx` and `cy` must be greater than or equal to 2.

– **Parameters**
  * `cx` – `double` array containing either the cutpoints or the class marks for `x`.
  * `cy` – `double` array containing either the cutpoints or the class marks for `y`.

– **Returns** – A two dimensional `double` array containing the two-way frequency table.

---

• *getMaximumX*
`public double` **getMaximumX**`( )`

– **Description**
Returns the maximum value of x.

– **Returns** – a `double` containing the maximum data bound for x.

---

• *getMaximumY*
`public double` **getMaximumY**`( )`

– **Description**
Returns the maximum value of y.

– **Returns** – a `double` containing the maximum data bound for y.

---

• *getMinimumX*
`public double` **getMinimumX**`( )`

– **Description**
Returns the minimum value of x.

– **Returns** – a `double` containing the minimum data bound for x.

---

• *getMinimumY*
`public double` **getMinimumY**`( )`

– **Description**
Returns the minimum value of y.

– **Returns** – a `double` containing the minimum data bound for y.

### Example: TableTwoWay

The data for x in this example is from Hinkley (1977) and Belleman and Hoaglin (1981). The measurement (in inches) are for precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years. The data for y were created by adding small integers to the data in x.

The first test uses the default tally method which may be appropriate when the range of data is unknown. The minimum and maximum data bounds are displayed.

The second test computes the table using known bounds, where the x lower, x upper, y lower, y upper bounds are chosen so that the intervals will be 0 to 1, 1 to 2, and so on for x and 1 to 2, 2 to 3 and so on for y.

In the third test, the class boundaries are input as the same intervals as in the second test. The first element of cmx and cmy specify the first cutpoint between classes.

The fourth test uses the cutpoints tally option with cutpoints such that the intervals are specified as in the previous tests.

```
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class  TableTwoWayEx1 {
    public static void main(String args[]) {

        int nx=5;
        int ny=6;
        double table[][];

        double[] x={
            0.77, 1.74, 0.81, 1.20, 1.95, 1.20, 0.47, 1.43, 3.37,
            2.20, 3.00, 3.09, 1.51, 2.10, 0.52, 1.62, 1.31, 0.32, 0.59,
            0.81, 2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96, 1.89, 0.9,
            2.05
        };
        double y[] = {
            1.77, 3.74, 3.81, 2.20, 3.95, 4.20, 1.47, 3.43, 6.37,
            3.20, 5.00, 6.09, 2.51, 4.10, 3.52, 2.62, 3.31, 3.32, 1.59,
            2.81, 5.81, 2.87, 3.18, 4.35, 5.75, 4.48, 3.96, 2.89, 2.9,
            5.05
        };


        TableTwoWay fTbl = new TableTwoWay(x, nx, y, ny);
```

```
        table = fTbl.getFrequencyTable();

        System.out.println("Example 1 ");
        System.out.println("Use Min and Max for bounds");
        new PrintMatrix("counts").print(table);

        System.out.println("-------------------------");
        System.out.println("Lower xbounds= "+fTbl.getMinimumX());
        System.out.println("Upper xbounds= "+fTbl.getMaximumX());
        System.out.println("Lower ybounds= "+fTbl.getMinimumY());
        System.out.println("Upper ybounds= "+fTbl.getMaximumY());
        System.out.println("-------------------------");


        double xlo = 1.0;
        double xhi = 4.0;
        double ylo = 2.0;
        double yhi = 6.0;
        System.out.println("");
        System.out.println("Use Known bounds");
        table = fTbl.getFrequencyTable(xlo, xhi,ylo, yhi);
        new PrintMatrix("counts").print(table);

        double cmx[] = { 0.5, 1.5, 2.5,3.5, 4.5};
        double cmy[] = {1.5, 2.5, 3.5, 4.5, 5.5, 6.5};
        table = fTbl.getFrequencyTableUsingClassmarks(cmx, cmy);
        System.out.println("");
        System.out.println("Use Class Marks");
        new PrintMatrix("counts").print(table);

        double cpx[] = {1,2,3,4};
        double cpy[] = {2,3,4,5,6};
        table = fTbl.getFrequencyTableUsingCutpoints(cpx, cpy);
        System.out.println("");
        System.out.println("Use Cutpoints");
        new PrintMatrix("counts").print(table);
    }
}
```

### Output

```
Example 1
Use Min and Max for bounds
       counts
   0  1  2  3  4  5
0  4  2  4  2  0  0
1  0  4  3  2  1  0
2  0  0  1  2  0  1
3  0  0  0  0  1  2
4  0  0  0  0  0  1

--------------------------
Lower xbounds= 0.32
Upper xbounds= 4.75
Lower ybounds= 1.47
Upper ybounds= 6.37
--------------------------

Use Known bounds
       counts
   0  1  2  3  4  5
0  3  2  4  0  0  0
1  0  5  5  2  0  0
2  0  0  1  3  2  0
3  0  0  0  0  0  2
4  0  0  0  0  1  0


Use Class Marks
       counts
   0  1  2  3  4  5
0  3  2  4  0  0  0
1  0  5  5  2  0  0
2  0  0  1  3  2  0
3  0  0  0  0  0  2
```

```
4  0  0  0  0  1  0


Use Cutpoints
       counts
   0  1  2  3  4  5
0  3  2  4  0  0  0
1  0  5  5  2  0  0
2  0  0  1  3  2  0
3  0  0  0  0  0  2
4  0  0  0  0  1  0
```

## *class* TableMultiWay

Tallies observations into a multi-way frequency table.

The `TableMultiWay` class determines the distinct values in multivariate data and computes frequencies for the data. This class accepts the data in the matrix `x`, but performs computations only for the variables (columns) in the first `nKeys` columns of `x` or by the variables specified in `indkeys`. In general, the variables for which frequencies should be computed are discrete; they should take on a relatively small number of different values. Variables that are continuous can be grouped first. `TableMultiWay` can be used to group variables and determine the frequencies of groups.

When method `getBalancedTable` is called, the inner class `BalancedTable` fills the vector values with the unique values in the vector of the variables and tallies the number of unique values of each variable table. Each combination of one value from each variable forms a cell in a multi-way table. The frequencies of these cells are entered in a table so that the first variable cycles through its values exactly once, and the last variable cycles through its values most rapidly. Some cells cannot correspond to any observations in the data; in other words, "missing cells" are included in table and have a value of 0. The frequency table is returned by the `BalancedTable` method `getTable`.

When method `getUnbalancedTable` is called, an instance of inner class `UnbalancedTable` is created, the frequency of each cell is entered in the unbalanced table so that the first variable cycles through its values exactly once and the last variable cycles through its values most rapidly. `table` is returned by `UnbalancedTable` method `getTable`. All cells have a frequency of at least 1, i.e., there is no "missing cell." The array `listCells`, returned by method `getListCells` can be considered "parallel" to `table` because row i of `listCells` is the set of `nKeys` values that describes the cell for which row $i$ of `table` contains the corresponding frequency.

## Declaration

public class com.imsl.stat.TableMultiWay
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* TableMultiWay.BalancedTable

Tallies the number of unique values of each variable.

#### Declaration

public class com.imsl.stat.TableMultiWay.BalancedTable
**extends** java.lang.Object

#### Methods

---

- *getNvalues*
  `public int[]` **getNvalues**`( )`
  - **Description**
    Returns an array of length `nKeys` containing in its $i$-th element ($i$=0,1,...nKeys-1), the number of levels or categories of the $i$-th classification variable (column).
  - **Returns** – an `int` array containing the number of levels or for each variable (column) in `x`.

---

- *getTable*
  `public double[]` **getTable**`( )`
  - **Description**
    Returns an array containing the frequencies for each variable. The array is of length `nValues[0] x nValues[1] x ... x nValues[nKeys]` containing the frequencies in the cells of the table to be fit, where `nValues` contains the result from `getNValues`.

Empty cells are included in table, and each element of table is nonnegative. The cells of table are sequenced so that the first variable cycles through its `nValues[0]` categories one time, the second variable cycles through its `nValues[1]` categories `nValues[0]` times, the third variable cycles through its `nValues[2]` categories `nValues[0] * nValues[1]` times, etc., up to the `nKeys`-th variable, which cycles through its `nValues[nKeys - 1]` categories `nValues[0] * nValues[1] * ... * nValues[nKeys - 2]` times.
- **Returns** – a `double` array containing the frequencies for each variable in `x`.

---

- *getValues*
  `public double[]` **getValues**`( )`
  - **Description**
    Returns the values of the classification variables. `getValues` returns an array of length `nValues[0] + nValues[1] + ... + nValues[nKeys - 1]` The first `nValues[0]` elements contain the values for the first classification variable. The next `nValues[1]` contain the values for the second variable. The last `nValues[nKeys - 1]` positions contain the values for the last classification variable, where `nValues` contains the result from `getNValues`.
  - **Returns** – a `double` array containing the values of the classification variables.

## *class* TableMultiWay.UnbalancedTable

Tallies the frequency of each cell in `x`.

### Declaration

public class com.imsl.stat.TableMultiWay.UnbalancedTable
**extends** java.lang.Object

### Methods

---

- *getListCells*
  `public double[]` **getListCells**`( )`
  - **Description**
    Returns for each row, a list of the levels of `nKeys` corresponding classification variables that describe a cell.
  - **Returns** – `double` array containing the list of levels of nKeys corresponding classification variables that describe a cell.

- *getNCells*
  `public int` **getNCells**`( )`
  - **Description**
    Returns the number of non-empty cells.
  - **Returns** – an `int` containing the number of non-empty cells.

- *getTable*
  `public double[]` **getTable**`( )`
  - **Description**
    Returns the frequency for each cell.
  - **Returns** – `double` array containing the frequency for each cell.

## Constructors

- *TableMultiWay*
  `public` **TableMultiWay**`( double[][] x, int` **nKeys** `)`
  - **Description**
    Constructor for `TableMultiWay`.
  - **Parameters**
    - `x` – A `double` matrix containing the observations and variables.
    - `nKeys` – `int` array containing the variables(columns) for which computations are to be performed.

- *TableMultiWay*
  `public` **TableMultiWay**`( double[][] x, int[]` **indkeys** `)`
  - **Description**
    Constructor for `TableMultiWay`.
  - **Parameters**
    - `x` – A `double` matrix containing the observations and variables.
    - `indkeys` – `int` array containing the variables(columns) for which computations are to be performed.

## Methods

- *getBalancedTable*
  `public TableMultiWay.BalancedTable` **getBalancedTable**`( )`
  - **Description**
    Returns an object containing the balanced table.
  - **Returns** – a `TableBalanced object`.

---

- *getGroups*
  `public int[]` **getGroups**`( )`
  - **Description**
    Returns the number of observations (rows) in each group. The number of groups is the length of the returned array. A group contains observations in `x` that are equal with respect to the method of comparison. If `n` contains the returned integer array, then the first `n[0]` rows of the sorted `x` are group number 1. The next `n[1]` rows of the sorted `x` are group number 2, etc. The last `n[n.length - 1]` rows of the sorted `x` are group number `n.length`.
  - **Returns** – an `int` array containing the number of observations (row) in each group.

---

- *getUnbalancedTable*
  `public TableMultiWay.UnbalancedTable` **getUnbalancedTable**`( )`
  - **Description**
    Returns an object containing the unbalanced table.
  - **Returns** – a `TableUnBalanced object`.

---

- *setFrequencies*
  `public void` **setFrequencies**`( double[]` **frequencies** `)`

### Example 1: TableMultiWay

The same data as used in SortEx2 is used in this example. It is a 10 x 3 matrix using Columns 0 and 1 as keys. There are two missing values (NaNs) in the keys. NaN is displayed as a ?. Table MultiWay determines the number of groups of different observations.

```
import com.imsl.stat.*;
import com.imsl.math.*;

public class TableMultiWayEx1 {
    public static void main (String args[]) {
        int nKeys=2;
        double x[][] = {{1.0, 1.0, 1.0},
```

```
                    {2.0, 1.0, 2.0},
                    {1.0, 1.0, 3.0},
                    {1.0, 1.0, 4.0},
                    {2.0, 2.0, 5.0},
                    {1.0, 2.0, 6.0},
                    {1.0, 2.0, 7.0},
                    {1.0, 1.0, 8.0},
                    {2.0, 2.0, 9.0},
                    {1.0, 1.0, 9.0}};

        x[4][1] = Double.NaN;
        x[6][0] = Double.NaN;

        PrintMatrix pm = new PrintMatrix("The Input Array");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        // Print the array
        pm.print(mf, x);
        System.out.println();

        TableMultiWay tbl = new TableMultiWay(x,nKeys);
        int ngroups[] = tbl.getGroups();
        System.out.println(" ngroups");
        for (int i=0; i < ngroups.length; i++)
            System.out.print(ngroups[i] + "  ");
    }
}
```

## Output

```
The Input Array

1  1  1
2  1  2
1  1  3
1  1  4
2  ?  5
1  2  6
?  2  7
1  1  8
```

```
2  2  9
1  1  9


 ngroups
5  1  1  1
```

### Example 2: TableMultiWay

The table of frequencies for a data matrix of size 30 x 2 is output.

```
import com.imsl.stat.*;
import com.imsl.math.*;
import java.text.MessageFormat;

public class TableMultiWayEx2 {

    public static void main(String args[]) {
        int indkeys[]={0,1};
        double x[][] = {
            {0.5, 1.5}, {1.5, 3.5}, {0.5, 3.5}, {1.5, 2.5}, {1.5, 3.5},
            {1.5, 4.5}, {0.5, 1.5}, {1.5, 3.5}, {3.5, 6.5}, {2.5, 3.5},
            {2.5, 4.5}, {3.5, 6.5}, {1.5, 2.5}, {2.5, 4.5}, {0.5, 3.5},
            {1.5, 2.5}, {1.5, 3.5}, {0.5, 3.5}, {0.5, 1.5}, {0.5, 2.5},
            {2.5, 5.5}, {1.5, 2.5}, {1.5, 3.5}, {1.5, 4.5}, {4.5, 5.5},
            {2.5, 4.5}, {0.5, 3.5}, {1.5, 2.5}, {0.5, 2.5}, {2.5, 5.5}
        };

        TableMultiWay tbl = new TableMultiWay(x,indkeys);

        int nvalues[] = tbl.getBalancedTable().getNvalues();

        double values[] = tbl.getBalancedTable().getValues();

        System.out.println("          row values");
        for (int i=0; i< nvalues[0]; i++)
            System.out.print(values[i]+"  ");
        System.out.println("");
        System.out.println("");
        System.out.println("           column values");
        for (int i=0; i< nvalues[1]; i++)
            System.out.print(values[i+nvalues[0]]+"   ");
```

```
        double table[] = tbl.getBalancedTable().getTable();

        System.out.println("");
        System.out.println("");
        System.out.println("        Table");



        System.out.print("      ");
        for (int i=0; i< nvalues[1]; i++)
            System.out.print(values[i+nvalues[0]]+ "   ");
        System.out.println("");
        for (int i=0; i< nvalues[0]; i++) {
            System.out.print(values[i]+ "   ");
            for (int j=0; j<nvalues[1]; j++)
                System.out.print(table[j +(nvalues[1]*i)]+ "   ");

            System.out.println(" ");
        }
    }
}
```

## Output

```

          row values
0.5  1.5  2.5  3.5  4.5

           column values
1.5   2.5   3.5   4.5   5.5   6.5

        Table
      1.5   2.5   3.5   4.5   5.5   6.5
0.5   3.0   2.0   4.0   0.0   0.0   0.0
1.5   0.0   5.0   5.0   2.0   0.0   0.0
2.5   0.0   0.0   1.0   3.0   2.0   0.0
3.5   0.0   0.0   0.0   0.0   0.0   2.0
4.5   0.0   0.0   0.0   0.0   1.0   0.0
```

### Example 3: TableMultiWay

The unbalanced table of frequencies for a data matrix of size 4 x 3 is output.

```
import com.imsl.stat.*;
import com.imsl.math.*;

public class TableMultiWayEx3 {
    public static void main(String args[]) {
        int    indkeys[] = {0,1};
        double x[][] = {
            {2.0, 5.0, 1.0}, {1.0, 5.0, 2.0},
            {1.0, 6.0, 3.0}, {2.0, 6.0, 4.0}
        };
        double frq[] = {1.0, 2.0, 3.0, 4.0};

        TableMultiWay tbl = new TableMultiWay(x,indkeys);
        tbl.setFrequencies(frq);

        int ncells = tbl.getUnbalancedTable().getNCells();
        double listCells[] = tbl.getUnbalancedTable().getListCells();
        double table[] = tbl.getUnbalancedTable().getTable();

        PrintMatrix pm = new PrintMatrix("List Cells");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        // Print the array
        pm.print(mf, listCells);
        System.out.println();

        pm = new PrintMatrix("Unbalanced Table");
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        // Print the array
        pm.print(mf, table);
        System.out.println();
    }
}
```

### Output

```
List Cells

1
5
1
6
2
5
2
6


Unbalanced Table

2
3
1
4
```

# Chapter 13

# Regression

Classes



## *class* LinearRegression

Fits a multiple linear regression model with or without an intercept. If the constructor argument `hasIntercept` is true, the multiple linear regression model is

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_i \quad i = 1, 2, \ldots, n$$

where the observed values of the $y_i$'s constitute the responses or values of the dependent variable, the $x_{i1}$'s, $x_{i2}$'s, $\ldots, x_{ik}$'s are the settings of the independent variables, $\beta_0, \beta_1, \ldots, \beta_k$ are the regression coefficients, and the $e_i$'s are independently distributed normal errors each with mean zero and variance $\sigma^2$. If `hasIntercept` is false, $\beta_0$ is not included in the model.

`LinearRegression` computes estimates of the regression coefficients by minimizing the sum of squares of the deviations of the observed response $y_i$ from the fitted response

$$\hat{y}_i$$

for the observations. This minimum sum of squares (the error sum of squares) is in the ANOVA output and denoted by

$$\text{SSE} = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i)^2$$

In addition, the total sum of squares is output in the ANOVA table. For the case, `hasIntercept` is true; the total sum of squares is the sum of squares of the deviations of $y_i$ from its mean

$$\bar{y}$$

–the so-called *corrected total sum of squares*; it is denoted by

$$\text{SST} = \sum_{i=1}^{n} w_i (y_i - \bar{y})^2$$

For the case `hasIntercept` is false, the total sum of squares is the sum of squares of $y_i$–the so-called *uncorrected total sum of squares*; it is denoted by

$$\text{SST} = \sum_{i=1}^{n} y_i^2$$

See Draper and Smith (1981) for a good general treatment of the multiple linear regression model, its analysis, and many examples.

In order to compute a least-squares solution, `LinearRegression` performs an orthogonal reduction of the matrix of regressors to upper triangular form. Givens rotations are used to reduce the matrix. This method has the advantage that the loss of accuracy resulting from forming the crossproduct matrix used in the normal equations is avoided, while not requiring the storage of the full matrix of regressors. The method is described by Lawson and Hanson, pages 207-212.

### Declaration

public class com.imsl.stat.LinearRegression
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Class

#### *class* LinearRegression.CoefficientTTests

CoefficientTTests contains statistics related to the regression coefficients.

##### Declaration

public class com.imsl.stat.LinearRegression.CoefficientTTests
**extends** java.lang.Object
**implements** java.io.Serializable

##### Methods

---

- *getCoefficient*
  `public double` **getCoefficient**`( int` **i** `)`
  - **Description**
    Returns the estimate for a coefficient.
  - **Parameters**
    - `i` – is the index of the coefficient whose estimate is to be returned.
  - **Returns** – the estimate for the $i$-th coefficient.

- *getPValue*
  `public double` **getPValue**`( int` **i** `)`
  - **Description**
    Returns the $p$-value for the two-sided test.
  - **Parameters**
    - `i` – is the index of the coefficient whose $p$-value is to be returned.
  - **Returns** – the $p$-value for the $i$-th coefficient estimate.

- *getStandardError*
  `public double` **getStandardError**`( int` **i** `)`

– **Description**
Returns the estimated standard error for a coefficient estimate.

– **Parameters**

  * `i` – is the index of the coefficient whose stardard error estimate is to be returned.

– **Returns** – the estimated standard error for the *i*-th coefficient estimate.

---

• *getTStatistic*
`public double` **getTStatistic**`( int` **i** `)`

– **Description**
Returns the t-statistic for the test that the *i*-th coefficient is zero.

– **Parameters**

  * `i` – is the index of the coefficient whose stardard error estimate is to be returned.

– **Returns** – the estimated standard error for the *i*-th coefficient estimate.

## Constructor

---

• *LinearRegression*
`public` **LinearRegression**`( int` **nVariables**`, boolean` **hasIntercept** `)`

– **Description**
Constructs a new linear regression object.

– **Parameters**

  * `nVariables` – `int` number of variables in the regression
  * `hasIntercept` – `int` boolean which indicates whether or not an intercept is in this regression model

## Methods

---

• *getANOVA*
`public synchronized ANOVA` **getANOVA**`( )`

– **Description**
Get an analysis of variance table and related statistics.

– **Returns** – an `ANOVA` table and related statistics

---

- *getCoefficients*
  `public synchronized double[]` **getCoefficients**`( )`
  - **Description**
    Returns the regression coefficients.
  - **Returns** – A `double` array containing the regression coefficients. If hasIntercept is false its length is equal to the number of variables. If hasIntercept is true then its length is the number of variables plus one and the 0-th entry is the value of the intercept.
  - **Throws**
    * `SingularMatrixException` – is thrown when the regression matrix is singular.

---

- *getCoefficientTTests*
  `public LinearRegression.CoefficientTTests` **getCoefficientTTests**`( )`
  - **Description**
    Returns statistics relating to the regression coefficients.

---

- *getR*
  `public synchronized double[][]` **getR**`( )`
  - **Description**
    Returns a copy of the R matrix. $R$ is the upper triangular matrix containing the $R$ matrix from a QR decomposition of the matrix of regressors.
  - **Returns** – a `double` matrix containing a copy of the R matrix

---

- *getRank*
  `public int` **getRank**`( )`
  - **Description**
    Returns the rank of the matrix.
  - **Returns** – the `int` rank of the matrix

---

- *update*
  `public void` **update**`( double[][]` **x**`, double[]` **y** `)`
  - **Description**
    Updates the regression object with a new set of observations.
  - **Parameters**
    * `x` – a `double` matrix containing the independent (explanatory) variables. The number of rows in x must equal the length of y and the number of columns must be equal to the number of variables set in the constructor.
    * `y` – a `double` array containing the dependent (response) variables.

---

- *update*
  `public void` **update**`( double[][] x, double[] y, double[] w )`

  - **Description**
    Updates the regression object with a new set of observations and weights.
  - **Parameters**
    * `x` – a `double` matrix containing the independent (explanatory) variables. The number of rows in x must equal the length of y and the number of columns must be equal to the number of variables set in the constructor.
    * `y` – a `double` array containing the dependent (response) variables.
    * `w` – a `double` array representing the weights

---

- *update*
  `public void` **update**`( double[] x, double y )`

  - **Description**
    Updates the regression object with a new observation.
  - **Parameters**
    * `x` – a `double` array containing the independent (explanatory) variables. Its length must be equal to the number of variables set in the constructor.
    * `y` – a `double` representing the dependent (response) variable

---

- *update*
  `public synchronized void` **update**`( double[] x, double y, double w )`

  - **Description**
    Updates the regression object with a new observation and weight.
  - **Parameters**
    * `x` – a `double` array containing the independent (explanatory) variables. Its length must be equal to the number of variables set in the constructor.
    * `y` – a `double` representing the dependent (response) variable
    * `w` – a `double` representing the weight

### Example: Linear Regression

The coefficients of a simple linear regression model, without an intercept, are computed.

```
import com.imsl.stat.*;

public class LinearRegressionEx1 {
    public static void main(String args[]) {
        // y = 4*x0 + 3*x1
        LinearRegression r = new LinearRegression(2, false);
```

```
        double c[] = {4, 3};
        double x[][] = {{1, 5},{0, 2},{-1, 4}};

        r.update(x[0],  1*c[0]+5*c[1]);
        r.update(x[1],  0*c[0]+2*c[1]);
        r.update(x[2], -1*c[0]+4*c[1]);
        double coef[] = r.getCoefficients();
        System.out.println("The computed regression coefficients are {" +
        coef[0] + ", " + coef[1] + "}");
    }
}
```

### Output

```
The computed regression coefficients are {4.0, 3.0}
```

## *class* **NonlinearRegression**

Fits a multivariate nonlinear regression model using least squares. The nonlinear regression model is

$$y_i = f(x_i; \theta) + \varepsilon_i \qquad i = 1, 2, \ldots, n$$

where the observed values of the $y_i$ constitute the responses or values of the dependent variable, the known $x_i$ are vectors of values of the independent (explanatory) variables, $\theta$ is the vector of $p$ regression parameters, and the $\varepsilon_i$ are independently distributed normal errors each with mean zero and variance $\sigma^2$. For this model, a least squares estimate of $\theta$ is also a maximum likelihood estimate of $\theta$.

The residuals for the model are

$$e_i(\theta) = y_i - f(x_i; \theta) \qquad i = 1, 2, \ldots, n$$

A value of $\theta$ that minimizes

$$\sum_{i=1}^{n} [e_i(\theta)]^2$$

is the least-squares estimate of $\theta$ calculated by this class. `NonlinearRegression` accepts these residuals one at a time as input from a user-supplied function. This allows `NonlinearRegression` to handle cases where $n$ is so large that data cannot reside in an array but must reside in a secondary storage device.

`NonlinearRegression` is based on MINPACK routines `LMDIF` and `LMDER` by More' et al. (1980). `NonlinearRegression` uses a modified Levenberg-Marquardt method to generate a sequence of approximations to the solution. Let $\hat{\theta}_c$ be the current estimate of $\theta$. A new estimate is given by

$$\hat{\theta}_c + s_c$$

where $s_c$ is a solution to

$$(J(\hat{\theta}_c)^T J(\hat{\theta}_c) + \mu_c I)s_c = J(\hat{\theta}_c)^T e(\hat{\theta}_c)$$

Here, $J(\hat{\theta}_c)$ is the Jacobian evaluated at $\hat{\theta}_c$.

The algorithm uses a "trust region" approach with a step bound of $\hat{\delta}_c$. A solution of the equations is first obtained for $\mu_c = 0$. If $||s_c||_2 < \delta_c$, this update is accepted; otherwise, $\mu_c$ is set to a positive value and another solution is obtained. The method is discussed by Levenberg (1944), Marquardt (1963), and Dennis and Schnabel (1983, pages 129 - 147, 218 - 338).

Forward finite differences are used to estimate the Jacobian numerically unless the user supplied function computes the derivatives. In this case the Jacobian is computed analytically via the user-supplied function.

`NonlinearRegression` does not actually store the Jacobian but uses fast Givens transformations to construct an orthogonal reduction of the Jacobian to upper triangular form. The reduction is based on fast Givens transformations (see Golub and Van Loan 1983, pages 156-162, Gentleman 1974). This method has two main advantages: (1) the loss of accuracy resulting from forming the crossproduct matrix used in the equations for $s_c$ is avoided, and (2) the $n$ x $p$ Jacobian need not be stored saving space when $n > p$.

A weighted least squares fit can also be performed. This is appropriate when the variance of $\epsilon_i$ in the nonlinear regression model is not constant but instead is $\sigma^2/w_i$. Here, $w_i$ are weights input via the user supplied function. For the weighted case, `NonlinearRegression` finds the estimate by minimizing a weighted sum of squares error.

## Programming Notes

Nonlinear regression allows users to specify the model's functional form. This added flexibility can cause unexpected convergence problems for users who are unaware of the limitations of the algorithm. Also, in many cases, there are possible remedies that may not be immediately obvious. The following is a list of possible convergence problems and some remedies. There is not a one-to-one correspondence between the problems and the remedies. Remedies for some problems may also be relevant for the other problems.

1. A local minimum is found. Try a different starting value. Good starting values often

can be obtained by fitting simpler models. For example, for a nonlinear function

$$f(x;\theta) = \theta_1 e^{\theta_2 x}$$

good starting values can be obtained from the estimated linear regression coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$ from a simple linear regression of *ln y* on *ln x*. The starting values for the nonlinear regression in this case would be

$$\theta_1 = e^{\hat{\beta}_0} \, and \, \theta_2 = \hat{\beta}_1$$

If an approximate linear model is unclear, then simplify the model by reducing the number of nonlinear regression parameters. For example, some nonlinear parameters for which good starting values are known could be set to these values. This simplifies the approach to computing starting values for the remaining parameters.

2. The estimate of $\theta$ is incorrectly returned as the same or very close to the initial estimate.
   - The scale of the problem may be orders of magnitude smaller than the assumed default of 1 causing premature stopping. For example, if the sums of squares for error is less than approximately $(2.22e^{-16})^2$, the routine stops. See Example 3, which shows how to shut down some of the stopping criteria that may not be relevant for your particular problem and which also shows how to improve the speed of convergence by the input of the scale of the model parameters.
   - The scale of the problem may be orders of magnitude larger than the assumed default causing premature stopping. The information with regard to the input of the scale of the model parameters in Example 3 is also relevant here. In addition, the maximum allowable step size (setMaxStepsize) in Example 3 may need to be increased.
   - The residuals are input with accuracy much less than machine accuracy causing premature stopping because a local minimum is found. Again see Example 3 to see generally how to change some default tolerances. If you cannot improve the precision of the computations of the residual, you need to use method setDigits to indicate the actual number of good digits in the residuals.

3. The model is discontinuous as a function of $\theta$. There may be a mistake in the user-supplied function. Note that the function $f(x;\theta)$ can be a discontinuous function of x.

4. The $R$ matrix returned by `getR` is inaccurate. If only a function is supplied try providing the com.imsl.stat.NonlinearRegression.Derivative. If the derivative is supplied try providing only com.imsl.stat.NonlinearRegression.Function.

5. Overflow occurs during the computations. Make sure the user-supplied functions do not overflow at some value of $\theta$.

6. The estimate of $\theta$ is going to infinity. A parameterization of the problem in terms of reciprocals may help.

7. Some components of $\theta$ are outside known bounds. This can sometimes be handled by making a function that produces artificially large residuals outside of the bounds (even though this introduces a discontinuity in the model function).

Note that the `solve` method must be called prior to calling the "get" member functions, otherwise a `null` is returned.

## Declaration

public class com.imsl.stat.NonlinearRegression
**extends** java.lang.Object

## Inner Classes

### *class* NonlinearRegression.NegativeFreqException

A negative frequency was encountered.

#### Declaration

public static class com.imsl.stat.NonlinearRegression.NegativeFreqException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *NonlinearRegression.NegativeFreqException*
  `public` **NonlinearRegression.NegativeFreqException**`( int` **rowIndex**`, int` **invocation**`, double` **value** `)`
  - **Description**
    Constructs a `NegativeFreqException`.
  - **Parameters**
    * `rowIndex` – An `int` which specifies the row index of X for which the frequency is negative.
    * `invocation` – An `int` which specifies the invocation number at which the error occurred. A 3 would indicate that the error occurred on the third invocation.
    * `value` – An `double` which represents the value of the frequency encountered.

## *class* **NonlinearRegression.NegativeWeightException**

A negative weight was encountered.

### Declaration

public static class com.imsl.stat.NonlinearRegression.NegativeWeightException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *NonlinearRegression.NegativeWeightException*
  `public` **NonlinearRegression.NegativeWeightException**`( int` **rowIndex**`,`
  `int` **invocation**`, double` **value** `)`
  - **Description**
    Constructs a `NegativeWeightException`.
  - **Parameters**
    * `rowIndex` – An `int` which specifies the row index of X for which the weight is negative.
    * `invocation` – An `int` which specifies the invocation number at which the error occurred. A 3 would indicate that the error occurred on the third invocation.
    * `value` – An `double` which represents the value of the weight encountered.

## *class* **NonlinearRegression.TooManyIterationsException**

The number of iterations has exceeded the maximum allowed.

### Declaration

public static class com.imsl.stat.NonlinearRegression.TooManyIterationsException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *NonlinearRegression.TooManyIterationsException*
  `public` **NonlinearRegression.TooManyIterationsException**`( )`

– **Description**
  Constructs a `TooManyIterationsException`.

### *interface* NonlinearRegression.Function

Public interface for the user supplied function for `NonlinearRegression`.

#### Declaration

public static interface com.imsl.stat.NonlinearRegression.Function

#### Method

---

- *f*
  `boolean f( double[] theta, int iobs, double[] frq, double[] wt, double[] e )`
  – **Description**
    Computes the weight, frequency, and residual given the parameter vector `theta` for a single observation.
  – **Parameters**
    * `theta` – An input `double` array containing the parameter values of the model. The length of `theta` corresponds to the number of unknown parameters in the model.
    * `iobs` – An input `int` value indicating the observation index. The function is evaluated at observation `y[iobs]`.
    * `frq` – An output `double` array of length 1 containing the frequency for observation `y[iobs]`.
    * `wt` – An output `double` array of length 1 containing the weight for observation `y[iobs]`. Use `wt` = 1.0 for equal weighting (unweighted least squares).
    * `e` – An output `double` array of length 1 which contains the error (residual) for observation `y[iobs]`.
  – **Returns** – A `boolean` value representing the completion indicator. `true` indicates `iobs` is less than the number of observations. `false` indicates `iobs` is greater than or equal to the number of observations and `wt`, `freq`, and `e` are not output.

## *interface* **NonlinearRegression.Derivative**

Public interface for the user supplied function to compute the derivative for `NonlinearRegression`.

### Declaration

public static interface com.imsl.stat.NonlinearRegression.Derivative
**implements** NonlinearRegression.Function

### Method

---

- *derivative*
  `boolean` **derivative**`(` `double[]` **theta**`,` `int` **iobs**`,` `double[]` **frq**`,` `double[]` **wt**`,` `double[]` **de** `)`
  - **Description**
    Computes the weight, frequency, and partial derivatives of the residual given the parameter vector `theta` for a single observation.
  - **Parameters**
    * `theta` – An input `double` array which contains the parameter values of the regression function. The length of `theta` corresponds to the number of unknown parameters in the regression function.
    * `iobs` – An input `int` value indicating the observation index. The function is evaluated at observation `y[iobs]`.
    * `frq` – An output `double` array of length 1 containing the frequency for observation `y[iobs]`.
    * `wt` – An output `double` array of length 1 containing the weight for the observation `y[iobs]`. Use `wt` = 1.0 for equal weighting (unweighted least squares).
    * `de` – An output `double` array containing the partial derivatives of the error (residual) for observation `y[iobs]`. The length of `de` corresponds to the number of unknown parameters in the regression function.
  - **Returns** – A `boolean` value representing the completion indicator. `true` indicates `iobs` is less than the number of observations. `false` indicates `iobs` is greater than or equal to the number of observations and `wt`, `freq`, and `de` are not output.

### Constructor

---

- *NonlinearRegression*
  `public` **NonlinearRegression**`( int` **nparm** `)`
  - **Description**
    Constructs a new nonlinear regression object.
  - **Parameters**
    * `nparm` – An `int` which specifies the number of unknown parameters in the regression.

### Methods

---

- *getCoefficient*
  `public double` **getCoefficient**`( int` **i** `)`
  - **Description**
    Returns the estimate for a coefficient.
  - **Parameters**
    * `i` – An `int` which specifies the index of a coefficient whose estimate is to be returned.
  - **Returns** – A `double` which contains the estimate for the $i$-th coefficient or `null` if `solve` has not been called.

---

- *getCoefficients*
  `public double[]` **getCoefficients**`( )`
  - **Description**
    Returns the regression coefficients.
  - **Returns** – A `double` array containing the regression coefficients or `null` if `solve` has not been called.

---

- *getDFError*
  `public double` **getDFError**`( )`
  - **Description**
    Returns the degrees of freedom for error.
  - **Returns** – A `double` which specifies the degrees of freedom for error or `null` if `solve` has not been called.

---

- *getErrorStatus*
  `public int` **getErrorStatus**`( )`

– **Description**
Gets information about the performance of `NonlinearRegression`.

– **Returns** – An `int` specifying information about convergence.

| Value | Description |
|---|---|
| 0 | All convergence tests were met. |
| 1 | Scaled step tolerance was satisfied. The current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or `StepTolerance` is too big. |
| 2 | Scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance `RelativeTolerance`. |
| 3 | Iterates appear to be converging to a noncritical point. Incorrect gradient information, a discontinuous function, or stopping tolerances being too tight may be the cause. |
| 4 | Five consecutive steps with the maximum stepsize have been taken. Either the function is unbounded below, or has a finite asymptote in some direction, or the `maxStepsize` is too small. |

---

• *getR*
`public double[][]` **getR**`( )`

– **Description**
Returns a copy of the `R` matrix. `R` is the upper triangular matrix containing the `R` matrix from a QR decomposition of the matrix of regressors.

– **Returns** – A two dimensional `double` array containing a copy of the `R` matrix or `null` if `solve` has not been called.

---

• *getRank*
`public int` **getRank**`( )`

– **Description**
Returns the rank of the matrix.

– **Returns** – An `int` which specifies the rank of the matrix or `null` if `solve` has not been called.

---

- *getSSE*
  `public double` **getSSE**`( )`
  - **Description**
    Returns the sums of squares for error.
  - **Returns** – A `double` which contains the sum of squares for error or `null` if `solve` has not been called.

---

- *setAbsoluteTolerance*
  `public void` **setAbsoluteTolerance**`( double` **absoluteTolerance** `)`
  - **Description**
    Sets the absolute function tolerance.
  - **Parameters**
    * `absoluteTolerance` – A `double` scalar value specifying the absolute function tolerance. The tolerance must be greater than or equal to zero. The default value is 4.93e-32.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `absoluteTolerance` is less than 0

---

- *setDigits*
  `public void` **setDigits**`( int` **nGood** `)`
  - **Description**
    Sets the number of good digits in the residuals.
  - **Parameters**
    * `nGood` – An `int` specifying the number of good digits in the residuals. The number of digits must be greater than zero. The default value is 15.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `ngood` is less than or equal to 0

---

- *setFalseConvergenceTolerance*
  `public void` **setFalseConvergenceTolerance**`( double` **falseConvergenceTolerance** `)`
  - **Description**
    Sets the false convergence tolerance.
  - **Parameters**
    * `falseConvergenceTolerance` – A `double` scalar value specifying the false convergence tolerance. The tolerance must be greater than or equal to zero. The default value is 2.22e-14.

– **Throws**
  - `java.lang.IllegalArgumentException` – is thrown if `falseConvergenceTolerance` is less than 0

---

- *setGradientTolerance*
  `public void` **setGradientTolerance**`( double` **gradientTolerance** `)`

  – **Description**
  Sets the gradient tolerance used to compute the gradient.

  – **Parameters**
  - `gradientTolerance` – A `double` specifying the gradient tolerance used to compute the gradient. The tolerance must be greater than or equal to zero. The default value is 6.055e-6.

  – **Throws**
  - `java.lang.IllegalArgumentException` – is thrown if `gradientTolerance` is less than 0

---

- *setGuess*
  `public void` **setGuess**`( double[]` **thetaGuess** `)`

  – **Description**
  Sets the initial guess of the parameter values

  – **Parameters**
  - `thetaGuess` – A `double` array of initial values for the parameters. The default value is an array of zeroes.

---

- *setInitialTrustRegion*
  `public void` **setInitialTrustRegion**`( double` **initialTrustRegion** `)`

  – **Description**
  Sets the initial trust region radius.

  – **Parameters**
  - `initialTrustRegion` – A `double` scalar value specifying the initial trust region radius. The initial trust radius must be greater than zero. If this member function is not called, a default is set based on the initial scaled Cauchy step.

  – **Throws**
  - `java.lang.IllegalArgumentException` – is thrown if `initialTrustRegion` is less than or equal to 0

---

- *setMaxIterations*
  `public void` **setMaxIterations**`( int` **maxIterations** `)`

– **Description**
Sets the maximum number of iterations allowed during optimization

– **Parameters**

* `maxIterations` – An `int` specifying the maximum number of iterations allowed during optimization. The value must be greater than 0. The default value is 100.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if `maxIterations` is less than or equal to 0

---

- *setMaxStepsize*
`public void` **setMaxStepsize**`( double` **maxStepsize** `)`

– **Description**
Sets the maximum allowable stepsize.

– **Parameters**

* `maxStepsize` – A nonnegative `double` value specifying the maximum allowable stepsize. The maximum allowable stepsize must be greater than zero. If this member function is not called, maximum stepsize is set to a default value based on a scaled `theta`.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if `maxStepsize` is less than or equal to 0

---

- *setRelativeTolerance*
`public void` **setRelativeTolerance**`( double` **relativeTolerance** `)`

– **Description**
Sets the relative function tolerance

– **Parameters**

* `relativeTolerance` – A `double` scalar value specifying the relative function tolerance. The relative function tolerance must be greater than or equal to zero. The default value is 1.0e-20.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if `relativeTolerance` is less than 0

---

- *setScale*
`public void` **setScale**`( double[]` **scale** `)`

– **Description**
Sets the scaling array for `theta`.

– **Parameters**

* `scale` – A `double` array containing the scaling values for the parameters (`theta`). The elements of the scaling array must be greater than zero. `scale` is used mainly in scaling the gradient and the distance between points. If good starting values of `thetaGuess` are known and are nonzero, then a good choice is `scale[i]=1.0/thetaGuess[i]`. Otherwise, if `theta` is known to be in the interval (-10.e5, 10.e5), set `scale[i]=10.e-5`. By default, the elements of the scaling array are set to 1.0.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if any of the elements of `scale` is less than or equal to 0

---

• *setStepTolerance*

`public void` **setStepTolerance**`( double` **stepTolerance** `)`

– **Description**

Sets the step tolerance used to step between two points.

– **Parameters**

* `stepTolerance` – A `double` scalar value specifying the step tolerance used to step between two points. The step tolerance must be greater than or equal to zero. The default value is 3.667e-11.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown if `stepTolerance` is less than 0

---

• *solve*

`public double[]` **solve**`( NonlinearRegression.Function` **F** `) throws com.imsl.stat.NonlinearRegression.TooManyIterationsException, com.imsl.stat.NonlinearRegression.NegativeFreqException, com.imsl.stat.NonlinearRegression.NegativeWeightException`

– **Description**

Solves the least squares problem and returns the regression coefficients.

– **Parameters**

* `F` – A `NonlinearRegression.Function` whose coefficients are to be computed.

– **Returns** – A `double` array containing the regression coefficients.

– **Throws**

* `com.imsl.stat.NonlinearRegression.TooManyIterationsException` – is thrown when the number of allowed iterations is exceeded
* `com.imsl.stat.NonlinearRegression.NegativeFreqException` – is thrown when the specified frequency is negative
* `com.imsl.stat.NonlinearRegression.NegativeWeightException` – is thrown when the weight is negative

### Example 1: Nonlinear Regression using Finite Differences

In this example a nonlinear model is fitted. The derivatives are obtained by finite differences.

```
import com.imsl.stat.*;
import com.imsl.math.*;

public class NonlinearRegressionEx1 {
    public static void main(String args[])
        throws NonlinearRegression.TooManyIterationsException,
        NonlinearRegression.NegativeFreqException,
        NonlinearRegression.NegativeWeightException {
        NonlinearRegression.Function f = new NonlinearRegression.Function() {

            public boolean f(double theta[], int iobs, double frq[],
               double wt[], double e[]){

                double ydata[] = {54.0, 50.0, 45.0, 37.0, 35.0, 25.0, 20.0,
                    16.0, 18.0, 13.0, 8.0, 11.0, 8.0, 4.0, 6.0};
                double xdata[] = {2.0, 5.0, 7.0, 10.0, 14.0, 19.0, 26.0, 31.0,
                    34.0, 38.0, 45.0, 52.0, 53.0, 60.0, 65.0};
                boolean iend;
                int nobs = 15;

                if(iobs < nobs){
                    wt[0] = 1.0;
                    frq[0] = 1.0;
                    iend = true;
                    e[0] = ydata[iobs] - theta[0] * Math.exp(theta[1]
                        * xdata[iobs]);
                } else {
                    iend = false;
                }
                return iend;
            }
        };

        int nparm = 2;
        double theta[] = {60.0, -0.03};
        NonlinearRegression regression = new NonlinearRegression(nparm);
        regression.setGuess(theta);
        double coef[] = regression.solve(f);
```

```
        System.out.println("The computed regression coefficients are {" +
            coef[0] + ", " + coef[1] + "}");
        int rank = regression.getRank();
        System.out.println("The computed rank is "+rank);
        double dfe = regression.getDFError();
        System.out.println("The degrees of freedom for error are "+dfe);
        double sse = regression.getSSE();
        System.out.println("The sums of squares for error is "+sse);
        double r[][] = regression.getR();
        new PrintMatrix("R from the QR decomposition ").print(r);
    }
}
```

## Output

```
The computed regression coefficients are {58.606562944502656, -0.0395864473118334}
The computed rank is 2
The degrees of freedom for error are 13.0
The sums of squares for error is 49.45929986247174
R from the QR decomposition
     0        1
0  1.874  1,139.928
1  0      1,139.798
```

## Example 2: Nonlinear Regression with User-supplied Derivatives

In this example a nonlinear model is fitted. The derivatives are supplied by the user.

```
import com.imsl.stat.*;
import com.imsl.math.*;

public class NonlinearRegressionEx2 {
   public static void main(String args[])
      throws NonlinearRegression.TooManyIterationsException,
      NonlinearRegression.NegativeFreqException,
      NonlinearRegression.NegativeWeightException {

      NonlinearRegression.Derivative deriv =
         new NonlinearRegression.Derivative() {
```

```
    double ydata[] = {54.0, 50.0, 45.0, 37.0, 35.0, 25.0, 20.0, 16.0,
       18.0, 13.0, 8.0, 11.0, 8.0, 4.0, 6.0};
    double xdata[] = {2.0, 5.0, 7.0, 10.0, 14.0, 19.0, 26.0, 31.0, 34.0,
       38.0, 45.0, 52.0, 53.0, 60.0, 65.0};
    boolean iend;
    int nobs = 15;

    public boolean f(double theta[], int iobs, double frq[], double wt[],
       double e[]){

       if(iobs < nobs){
           wt[0] = 1.0;
           frq[0] = 1.0;
           iend = true;
           e[0] = ydata[iobs] - theta[0] * Math.exp(theta[1]
              * xdata[iobs]);
       } else {
           iend = false;
       }
       return iend;
    }

    public boolean derivative(double theta[], int iobs, double frq[],
       double wt[], double de[]){
       if(iobs < nobs){
           wt[0] = 1.0;
           frq[0] = 1.0;
           iend = true;
           de[0] = -Math.exp(theta[1]*xdata[iobs]);
           de[1] = -theta[0] * xdata[iobs] * Math.exp(theta[1]
              * xdata[iobs]);
       } else {
           iend = false;
       }
       return iend;
    }
};

   int nparm = 2;
   double theta[] = {60.0, -0.03};
   NonlinearRegression regression = new NonlinearRegression(nparm);
```

```
        regression.setGuess(theta);
        double coef[] = regression.solve(deriv);
        System.out.println("The computed regression coefficients are {" +
        coef[0] + ", " + coef[1] + "}");
        int rank = regression.getRank();
        System.out.println("The computed rank is "+rank);
        double dfe = regression.getDFError();
        System.out.println("The degrees of freedom for error are "+dfe);
        double sse = regression.getSSE();
        System.out.println("The sums of squares for error is "+sse);
        double r[][] = regression.getR();
        new PrintMatrix("R from the QR decomposition ").print(r);
    }
}
```

## Output

```
The computed regression coefficients are {58.60656292541919, -0.039586447277524736}
The computed rank is 2
The degrees of freedom for error are 13.0
The sums of squares for error is 49.45929986247219
R from the QR decomposition
     0       1
0  1.874  1,139.928
1  0      1,139.798
```

## Example 3: Nonlinear Regression using Set Methods

In this example some nondefault tolerances and scales are used to fit a nonlinear model. The data is 1.e-10 times the data of example 1. In order to fit this model without rescaling the data we first set the absolute function tolerance to 0.0. The default value would have caused the program to terminate after one iteration because the residual sum of squares is roughly 1.e-19. We also set the relative function tolerance to 0.0. The gradient tolerance is properly scaled for this problem so we leave it at \ its default value. Finally, we set the elements of scale to be the absolute value of the recipricol of the starting value.The derivatives are obtained by finite differences.

```
import com.imsl.stat.*;
```

```
public class NonlinearRegressionEx3 {
   public static void main(String args[])
      throws NonlinearRegression.TooManyIterationsException,
      NonlinearRegression.NegativeFreqException,
      NonlinearRegression.NegativeWeightException {

      NonlinearRegression.Function f = new NonlinearRegression.Function() {

         public boolean f(double theta[], int iobs, double frq[], double wt[],
            double e[]){

            double ydata[] = {54.e-10, 50.e-10, 45.e-10, 37.e-10, 35.e-10,
               25.e-10, 20.e-10, 16.e-10, 18.e-10, 13.e-10, 8.e-10, 11.e-10,
               8.e-10, 4.e-10, 6.e-10};
            double xdata[] = {2.0, 5.0, 7.0, 10.0, 14.0, 19.0, 26.0, 31.0,
               34.0, 38.0, 45.0, 52.0, 53.0, 60.0, 65.0};
            boolean iend;
            int nobs = 15;
            if(iobs < nobs){
                wt[0] = 1.0;
                frq[0] = 1.0;
                iend = true;
                e[0] = ydata[iobs] - theta[0] * Math.exp(theta[1]
                   * xdata[iobs]);
            } else {
                iend = false;
            }
            return iend;
         }
      };
      int nparm = 2;
      double theta[] = {6.e-9, -0.03};
      double scale[] = new double[nparm];
      double r[][] = new double[nparm][nparm];
      NonlinearRegression regression = new NonlinearRegression(nparm);
      regression.setGuess(theta);
      regression.setAbsoluteTolerance(0.0);
      regression.setRelativeTolerance(0.0);
      scale[0] = 1.0/Math.abs(theta[0]);
      scale[1] = 1.0/Math.abs(theta[1]);
      regression.setScale(scale);
      double coef[] = regression.solve(f);
```

```
        System.out.println("The computed regression coefficients are {" +
            coef[0] + ", " + coef[1] + "}");
        int rank = regression.getRank();
        System.out.println("The computed rank is "+rank);
        double dfe = regression.getDFError();
        System.out.println("The degrees of freedom for error are "+dfe);
        double sse = regression.getSSE();
        System.out.println("The sums of squares for error is "+sse);
        r = regression.getR();
        System.out.println("R from the QR decomposition is "
            + r[0][0] + " " + r[0][1]);
        System.out.println("                                "
            + r[1][0] + " " + r[1][1]);
    }
}
```

### Output

```
The computed regression coefficients are {5.7837836210879824E-9, -0.0396252538296399}
The computed rank is 2
The degrees of freedom for error are 13.0
The sums of squares for error is 5.166376610434158E-19
R from the QR decomposition is 1.873105632124423 5.7473458654105505E-9
                               0.0 5.837139910539398E-11
```

## *class* **UserBasisRegression**

Generates summary statistics using user supplied functions in a nonlinear regression model

### Declaration

public class com.imsl.stat.UserBasisRegression
**extends** java.lang.Object

### Constructor

- *UserBasisRegression*
  `public` **UserBasisRegression**`( RegressionBasis` **basis**`, int` **nBasis**`, boolean` **hasIntercept** `)`

  – **Description**
    Constructs a `UserBasisRegression` object
  – **Parameters**
    * `basis` – a `RegressionBasis` basis function supplied by the user
    * `nBasis` – an `int` which specifies the number of basis functions
    * `hasIntercept` – a `boolean` which specifies whether or not the model has an intercept

## Methods

---

- *getANOVA*
  `public ANOVA` **getANOVA**`( )`

  – **Description**
    Get an analysis of variance table and related statistics.
  – **Returns** – an `ANOVA` table and related statistics

---

- *getCoefficients*
  `public double[]` **getCoefficients**`( )`

  – **Description**
    Returns the regression coefficients.
  – **Returns** – A `double` array containing the regression coefficients. If hasIntercept is false its length is equal to the number of variables. If hasIntercept is true then its length is the number of variables plus one and the 0-th entry is the value of the intercept.
  – **Throws**
    * `SingularMatrixException` – is thrown when the regression matrix is singular.

---

- *update*
  `public void` **update**`( double` **x**`, double` **y**`, double` **w** `)`

  – **Description**
    Adds a new observation and associated weight to the `RegressionBasis` object.
  – **Parameters**
    * `x` – a `double` containing the independent (explanatory) variable.
    * `y` – a `double` containing the dependent (response) variable.
    * `w` – a `double` representing the weight

### Example: Regression with User-supplied Basis Functions

In this example, we fit the function 1 + sin(x) + 7 * sin(3x) with no error introduced. The function is evaluated at 90 equally spaced points on the interval [0, 6]. Four basis functions are used, sin(kx) for k = 1,...,4 with no intercept.

```
import com.imsl.stat.*;
import com.imsl.math.*;

public class UserBasisRegressionEx1 {
    public static void main(String args[]) {
        class Basis1 implements RegressionBasis {
            public double basis(int index, double x) {
                return Math.sin((index+1)*x);
            }
        }

        double coef[] = new double[4];
        UserBasisRegression ubr =
        new UserBasisRegression(new Basis1(), 4, false);

        for (int k = 0;  k < 90;  k++) {
            double x = 6.0*k/89.0;
            double y = 1.0 + Math.sin(x) + 7.0*Math.sin(3.0*x);
            ubr.update(x, y, 1.0);
        }
        coef = ubr.getCoefficients();
        new PrintMatrix("The regression coefficients are:").print(coef);
    }
}
```

### Output

```
The regression coefficients are:
     0
0  1.01
1  0.02
2  7.029
3  0.037
```

# *interface* RegressionBasis

Public interface for user supplied function to `UserBasisRegression` object.

## Declaration

public interface com.imsl.stat.RegressionBasis

## Method

---

- *basis*
  `double` **`basis`**`( int` **`index`**`, double` **`x`** `)`
  - **Description**
    Public interface for the nonlinear least-squares function.
  - **Parameters**
    * `index` – an `int` which specifies the index of the basis function to be evaluated at x
    * `x` – a `double`, the point at which the function is to be evaluated
  - **Returns** – a `double`, the returned value of the function at x

# *class* SelectionRegression

Selects the best multiple linear regression models.

Class `SelectionRegression` finds the best subset regressions for a regression problem with three or more independent variables. Typically, the intercept is forced into all models and is not a candidate variable. In this case, a sum of squares and crossproducts matrix for the independent and dependent variables corrected for the mean is computed internally. Optionally, `SelectionRegression` supports user-calculated sum-of-squares and crossproducts matrices; see the description of the `compute` method.

"Best" is defined by using one of the following three criteria:

- $R^2$ (in percent)

$$R^2 = 100(1 - \frac{\text{SSE}_p}{\text{SST}})$$

- $R_a^2$ (adjusted $R^2$)

$$R_a^2 = 100[1 - (\frac{n-1}{n-p})\frac{\text{SSE}_p}{\text{SST}}]$$

  Note that maximizing the $R_a^2$ is equivalent to minimizing the residual mean squared error:

$$\frac{\text{SSE}_p}{(n-p)}$$

- Mallow's $C_p$ statistic

$$C_p = \frac{\text{SSE}_p}{s^2_{\text{k}}} + 2p - n$$

Here, $n$ is equal to the sum of the frequencies (or the number of rows in `x` if frequencies are not specified in the `compute` method), and SST is the total sum of squares. $k$ is the number of candidate or independent variables, represented as the `nCandidate` argument in the `SelectionRegression` constructor. $\text{SSE}_p$ is the error sum of squares in a model containing $p$ regression parameters including $\beta_0$ (or $p$ - 1 of the $k$ candidate variables). Variable

$$S^2_{\text{k}}$$

is the error mean square from the model with all $k$ variables in the model. Hocking (1972) and Draper and Smith (1981, pp. 296-302) discuss these criteria.

Class `SelectionRegression` is based on the algorithm of Furnival and Wilson (1974). This algorithm finds the maximum number of good saved candidate regressions for each possible subset size. For more details, see method setMaximumGoodSaved. These regressions are used to identify a set of best regressions. In large problems, many regressions are not computed. They may be rejected without computation based on results for other subsets; this yields an efficient technique for considering all possible regressions.

There are cases when the user may want to input the variance-covariance matrix rather than allow it to be calculated. This can be accomplished using the appropriate `compute` method. Three situations in which the user may want to do this are as follows:

1. The intercept is not in the model. A raw (uncorrected) sum of squares and crossproducts matrix for the independent and dependent variables is required. Argument `nObservations` must be set to 1 greater than the number of observations. Form $A^TA$, where $A = [A,\ Y]$, to compute the raw sum of squares and crossproducts matrix.
2. An intercept is a candidate variable. A raw (uncorrected) sum of squares and crossproducts matrix for the constant regressor (= 1.0), independent, and dependent variables is required for `cov` . In this case, `cov` contains one additional row and column corresponding to the constant regressor. This row and column contain the sum of squares and crossproducts of the constant regressor with the independent

and dependent variables. The remaining elements in `cov` are the same as in the previous case. Argument `nObservations` must be set to 1 greater than the number of observations.

3. There are $m$ variables that must be forced into the models. A sum of squares and crossproducts matrix adjusted for the $m$ variables is required (calculated by regressing the candidate variables on the variables to be forced into the model). Argument `nObservations` must be set to $m$ less than the number of observations.

## Programming Notes

`SelectionRegression` can save considerable CPU time over explicitly computing all possible regressions. However, the function has some limitations that can cause unexpected results for users who are unaware of the limitations of the software.

1. For $\mathrm{k} + 1 > -\log_2(\epsilon)$, where $\epsilon$ is the largest relative spacing for double precision, some results can be incorrect. This limitation arises because the possible models indicated (the model numbers 1, 2, ..., $2k$) are stored as floating-point values; for sufficiently large $k$, the model numbers cannot be stored exactly. On many computers, this means `SelectionRegression` (for $\mathrm{k} > 49$) can produce incorrect results.
2. `SelectionRegression` eliminates some subsets of candidate variables by obtaining lower bounds on the error sum of squares from fitting larger models. First, the full model containing all independent variables is fit sequentially using a forward stepwise procedure in which one variable enters the model at a time, and criterion values and model numbers for all the candidate variables that can enter at each step are stored. If linearly dependent variables are removed from the full model, a warning "SelectionRegression.VariablesDeleted") is issued. In this case, some submodels that contain variables removed from the full model because of linear dependency can be overlooked if they have not already been identified during the initial forward stepwise procedure. If this warning is issued and you want the variables that were removed from the full model to be considered in smaller models, you can rerun the program with a set of linearly independent variables.

## Declaration

public class com.imsl.stat.SelectionRegression
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* SelectionRegression.NoVariablesException

No Variables can enter the model.

#### Declaration

public static class com.imsl.stat.SelectionRegression.NoVariablesException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *SelectionRegression.NoVariablesException*
  `public` **SelectionRegression.NoVariablesException**`( )`
  - **Description**
    Constructs a `NoVariablesException`.

### *class* SelectionRegression.Statistics

`Statistics` contains statistics related to the regression coefficients.

#### Declaration

public class com.imsl.stat.SelectionRegression.Statistics
**extends** java.lang.Object
**implements** java.io.Serializable

#### Methods

---

- *getCoefficientStatistics*
  `public double[][]` **getCoefficientStatistics**`( int` **regressionIndex** `)`
  - **Description**
    Returns the coefficients statistics for each of the best regressions found for each subset considered.

The value set by method setMaximumBestFound determines the total number of best regressions to find. The number of best regression is equal to (`maxSubset` x `maxFound`), if criterion `R_SQUARED_CRITERION` is specified or it is equal to `maxFound` if either `MALLOWS_CP_CRITERION` or `ADJUSTED_R_SQUARED_CRITERION` is specified.
Each row contains statistics related to the regression coefficients of the best models. The regressions are ordered so that the better regressions appear first. The statistic in the columns are as follows (inferences are conditional on the selected model):

| Column | Description |
|---|---|
| 0 | variable number |
| 1 | coefficient estimate |
| 2 | estimated standard error of the estimate |
| 3 | $t$-statistic for the test that the coefficient is 0 |
| 4 | $p$-value for the two-sided $t$ test |

– **Parameters**
  * `regressionIndex` – An `int` which specifies the index of the best regression statistics to return. There will be 0 to (`maxSubset` x `maxFound - 1`) best regressions if `R_SQUARED_CRITERION` is specified or 0 to (`maxFound - 1`) if either `MALLOWS_CP_CRITERION` or `ADJUSTED_R_SQUARED_CRITERION` is specified.

– **Returns** – A two-dimensional `double` array containing the regression statistics.

---

• *getCriterionValues*
`public double[]` **getCriterionValues**`( int` **numVariables** `)`

– **Description**
Returns an array containing the values of the best criterion for the number of variables considered.

– **Parameters**
  * `numVariables` – An `int` which specifies the number of variables considered.

– **Returns** – A `double` array with `maxSubset` rows and `nCandidate` columns containing the criterion values.

---

• *getIndependentVariables*
`public int[][]` **getIndependentVariables**`( int` **numVariables** `)`

– **Description**
Returns the identification numbers for the independent variables for the number of variables considered and in the same order as the criteria returned by getCriterionValues.

– **Parameters**
  * `numVariables` – An `int` which specifies the number of variables considered.
– **Returns** – An `int` matrix containing the identification numbers for the independent variables considered.

## Fields

---

- public static final `int` **R_SQUARED_CRITERION**
  – Indicates $R^2$ criterion regression.
- public static final `int` **ADJUSTED_R_SQUARED_CRITERION**
  – Indicates $R_a^2$ (adjusted $R^2$) criterion regression.
- public static final `int` **MALLOWS_CP_CRITERION**
  – Indicates Mallow's $C_p$ criterion regression.

## Constructor

---

- *SelectionRegression*
  `public` **SelectionRegression**`( int` **nCandidate** `)`
  – **Description**
  Constructs a new `SelectionRegression` object.
  – **Parameters**
    * `nCandidate` – An `int` containing the number of candidate variables (independent variables). `nCandidate` must be greater than 2.

## Methods

---

- *compute*
  `public void` **compute**`( double[][]` **x**`, double[]` **y** `) throws com.imsl.stat.SelectionRegression.NoVariablesException, com.imsl.stat.Covariances.TooManyObsDeletedException, com.imsl.stat.Covariances.MoreObsDelThanEnteredException, com.imsl.stat.Covariances.DiffObsDeletedException`

– **Description**
Computes the best multiple linear regression models.

– **Parameters**
  * `x` – A `double` matrix containing the observations of the candidate (independent) variables. The number of columns in `x` must be equal to the number of variables set in the constructor.
  * `y` – A `double` array containing the observations of the dependent variable.

– **Throws**
  * `com.imsl.stat.SelectionRegression.NoVariablesException` – if no variables can enter any model
  * `com.imsl.stat.Covariances.TooManyObsDeletedException` – more observations have been deleted than were originally entered
  * `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – more observations are being deleted from the output covariance matrix than were originally entered
  * `com.imsl.stat.Covariances.DiffObsDeletedException` – different observations are being deleted from return matrix than were originally entered

---

• *compute*

`public void` **compute**`( double[][]` **x**`, double[]` **y**`, double[]` **weights** `)`
`throws com.imsl.stat.SelectionRegression.NoVariablesException,`
`com.imsl.stat.Covariances.NonnegativeWeightException,`
`com.imsl.stat.Covariances.TooManyObsDeletedException,`
`com.imsl.stat.Covariances.MoreObsDelThanEnteredException,`
`com.imsl.stat.Covariances.DiffObsDeletedException`

– **Description**
Computes the best weighted multiple linear regression models.

– **Parameters**
  * `x` – A `double` matrix containing the observations of the candidate (independent) variables. The number of columns in `x` must be equal to the number of variables set in the constructor.
  * `y` – A `double` array containing the observations of the dependent variable.
  * `weights` – A `double` array containing the weight for each of the observations.

– **Throws**
  * `com.imsl.stat.SelectionRegression.NoVariablesException` – if no variables can enter any model
  * `com.imsl.stat.Covariances.NonnegativeWeightException` – weights must be nonnegative
  * `com.imsl.stat.Covariances.TooManyObsDeletedException` – more observations have been deleted than were originally entered

* `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – more observations are being deleted from the output covariance matrix than were originally entered
* `com.imsl.stat.Covariances.DiffObsDeletedException` – different observations are being deleted from return matrix than were originally entered

---

- *compute*

  public void **compute**( double[][] **x**, double[] **y**, double[] **weights**, double[] **frequencies** ) throws com.imsl.stat.SelectionRegression.NoVariablesException, com.imsl.stat.Covariances.NonnegativeFreqException, com.imsl.stat.Covariances.NonnegativeWeightException, com.imsl.stat.Covariances.TooManyObsDeletedException, com.imsl.stat.Covariances.MoreObsDelThanEnteredException, com.imsl.stat.Covariances.DiffObsDeletedException

  - **Description**

    Computes the best weighted multiple linear regression models using frequencies for each observation.
  - **Parameters**
    * `x` – A `double` matrix containing the observations of the candidate (independent) variables. The number of columns in `x` must be equal to the number of variables set in the constructor.
    * `y` – A `double` array containing the observations of the dependent variable.
    * `weights` – A `double` array containing the weight for each of the observations.
    * `frequencies` – A `double` array containing the frequency for each of the observations of `x`.
  - **Throws**
    * `com.imsl.stat.SelectionRegression.NoVariablesException` – if no variables can enter any model
    * `com.imsl.stat.Covariances.NonnegativeFreqException` – frequencies must be nonnegative
    * `com.imsl.stat.Covariances.NonnegativeWeightException` – weights must be nonnegative
    * `com.imsl.stat.Covariances.TooManyObsDeletedException` – more observations have been deleted than were originally entered
    * `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – more observations are being deleted from the output covariance matrix than were originally entered
    * `com.imsl.stat.Covariances.DiffObsDeletedException` – different observations are being deleted from return matrix than were originally entered

- *compute*
  `public void` **compute**`( double[][]` **cov**`, int` **nObservations** `) throws com.imsl.stat.SelectionRegression.NoVariablesException`

  – **Description**
    Computes the best multiple linear regression models using a user-supplied covariance matrix.
  – **Parameters**
    * `cov` – A `double` matrix containing a variance-covariance or sum of squares and crossproducts matrix, in which the last column must correspond to the dependent variable. `cov` can be computed using the Covariances class.
    * `nObservations` – An `int` containing the number of observations used to compute `cov`.
  – **Throws**
    * `com.imsl.stat.SelectionRegression.NoVariablesException` – if no variables can enter any model

- *getCriterionOption*
  `public int` **getCriterionOption**`( )`

  – **Description**
    Returns the criterion option used to calculate the regression estimates.
  – **Returns** – An `int` containing the criterion option.

- *getStatistics*
  `public SelectionRegression.Statistics` **getStatistics**`( )`

  – **Description**
    Returns a new `Statistics` object.
  – **Returns** – A `Statistics` object containing the Coefficient statistics.

- *setCriterionOption*
  `public void` **setCriterionOption**`( int` **criterionOption** `)`

  – **Description**
    Sets the Criterion to be used. By default for all criteria, subset size 1,2, ..., $k =$ `nCandidate` are considered. However, for $R^2$ the maximum number of subsets can be restricted to `maxSubset` in the setMaximumSubsetSize method.

| Criterion Option | Description |
|---|---|
| R_SQUARED_CRITERION | For $R^2$, subset sizes 1, 2, ..., `maxSubset` are examined. This is the default with `maxSubset` = `nCandidate`. |
| ADJUSTED_R_SQUARED_CRITERION | For Adjusted $R^2$, subset sizes 1, 2, ..., `nCandidate` are examined. |
| MALLOWS_CP_CRITERION | For Mallow's $C_p$ Subset sizes 1, 2, ..., `nCandidate` are examined. |

- **Parameters**
  - `criterionOption` – An `int` containing the criterion option used for the best subset regression selection.

---

- *setMaximumBestFound*

  `public void` **setMaximumBestFound**`( int` **maxFound** `)`

  - **Description**

    Sets the maximum number of best regressions to be found.
    If the $R^2$ criterion option is selected, the `maxFound` best regressions for each subset size examined are reported. If the adjusted $R^2$ or Mallow's $C_p$ criteria are selected, the `maxFound` among all possible regressions are found.
  - **Parameters**
    - `maxFound` – An `int` containing the maximum number of best regressions to be reported. Default: `maxFound` = 1.

---

- *setMaximumGoodSaved*

  `public void` **setMaximumGoodSaved**`( int` **maxSaved** `)`

  - **Description**

    Sets the maximum number of good regressions for each subset size saved. Argument `maxSaved` must be greater than or equal to `maxFound`. Normally, `maxSaved` should be less than or equal to 10. It should never need be larger than `maxSubset`, the maximum number of subsets for any subset size. Computing time required is inversely related to `maxSaved`.
  - **Parameters**
    - `maxSaved` – An `int` containing the maximum number of good regressions saved for each subset size. Default: `maxSaved` = maximum(10,`maxSubset`).

---

- *setMaximumSubsetSize*

  `public void` **setMaximumSubsetSize**`( int` **maxSubset** `)`

  - **Description**

    Sets the maximum subset size if $R^2$ criterion is used.
  - **Parameters**

* `maxSubset` – An `int` containing the maximum subset size when $R^2$ criterion is used. Default: `maxSubset` = `nCandidate`.

### Example 1: SelectionRegression

This example uses a data set from Draper and Smith (1981, pp. 629*630). Class `SelectionRegression` is invoked to find the best regression for each subset size using the $R^2$ criterion.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class SelectionRegressionEx1 {

    public static void main(String[] args)  throws Exception {
        double x[][] = { {7., 26., 6., 60.}, {1., 29., 15., 52.},
            {11., 56., 8., 20.}, {11., 31., 8., 47.}, {7., 52., 6., 33.},
            {11., 55., 9., 22.}, {3., 71., 17., 6.}, {1., 31., 22., 44.},
            {2., 54., 18., 22.}, {21., 47., 4., 26}, {1., 40., 23., 34.},
            {11., 66., 9., 12.}, {10.0, 68., 8., 12.}};

        double y[] = { 78.5, 74.3, 104.3, 87.6, 95.9, 109.2, 102.7, 72.5, 93.1,
            115.9, 83.8, 113.3, 109.4};

        String criterionOption;
        MessageFormat critMsg =
           new MessageFormat("Regressions with {0} variable(s) ({1})");
        MessageFormat critLabel =
           new MessageFormat("   Criterion               Variables");
        MessageFormat coefMsg =
           new MessageFormat("Best Regressions with {0} variable(s) ({1})");
        MessageFormat coefLabel = new MessageFormat("Variable   Coefficient" +
           "   Standard Error  t-statistic   p-value");
        MessageFormat critData = new MessageFormat("{0}   {1}   {2}   {3}" +
           "   {4}   {5}");

            SelectionRegression sr = new SelectionRegression(4);
            sr.compute(x, y);
            SelectionRegression.Statistics stats =
                sr.getStatistics();
```

```
            criterionOption = new String("R-squared");

        for (int i=1; i <= 4 ; i++) {
            double[] tmpCrit = stats.getCriterionValues(i);
            int[][] indvar = stats.getIndependentVariables(i);

            Object p[] = {new Integer(i), criterionOption};
            System.out.println(critMsg.format(p));
            Object p1[] = {null};
            System.out.println(critLabel.format(p1));

            for (int j=0; j< tmpCrit.length; j++) {
                System.out.print("     "+tmpCrit[j]+"        ");
                for (int k = 0; k < indvar[j].length ; k++) {
                    System.out.print(indvar[j][k]+"   ");
                }
                System.out.println("");
            }
            System.out.println("");
        }

        for (int i=0; i < 4; i++) {
            System.out.println("");
            Object p[] = {new Integer(i+1), criterionOption};
            System.out.println(coefMsg.format(p));
            Object p2[] = {null};
            System.out.println(coefLabel.format(p2));

            double[][] tmpCoef= stats.getCoefficientStatistics(i);
            PrintMatrix pm = new PrintMatrix();
            pm.setColumnSpacing(10);
            PrintMatrixFormat tst = new PrintMatrixFormat();
            tst.setNoColumnLabels();
            tst.setNoRowLabels();
            pm.print(tst, tmpCoef);
            System.out.println("");
            System.out.println("");
        }
    }

}
```

## Output

```
Regressions with 1 variable(s) (R-squared)
   Criterion               Variables
     67.45419641316093        4
     66.6268257633294        2
     53.39480238350336        1
     28.587273122981173        3

Regressions with 2 variable(s) (R-squared)
   Criterion               Variables
     97.86783745356321        1   2
     97.24710477169315        1   4
     93.52896406158075        3   4
     68.00604079500503        2   4
     54.81667488448235        1   3

Regressions with 3 variable(s) (R-squared)
   Criterion               Variables
     98.23354512004268        1   2   4
     98.22846792190867        1   2   3
     98.12810925873437        1   3   4
     97.28199593862732        2   3   4

Regressions with 4 variable(s) (R-squared)
   Criterion               Variables
     98.23756204076803        1   2   3   4


Best Regressions with 1 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

4          -0.738          0.155          -4.775          0.001




Best Regressions with 2 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

1          1.468          0.121          12.105          0
```

```
2          0.662          0.046          14.442          0
```

```
Best Regressions with 3 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

1           1.452          0.117          12.41           0
2           0.416          0.186           2.242          0.052
4          -0.237          0.173          -1.365          0.205
```

```
Best Regressions with 4 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

1           1.551          0.745           2.083          0.071
2           0.51           0.724           0.705          0.501
3           0.102          0.755           0.135          0.896
4          -0.144          0.709          -0.203          0.844
```

### Example 2: SelectionRegression

This example uses the same data set as the first example, but Mallow's $C_p$ statistic is used as the criterion rather than $R^2$. Note that when Mallow's $C_p$ statistic (or adjusted $R^2$) is specified, the method `setMaximumBestFound` is used to indicate the total number of "best" regressions (rather than indicating the number of best regressions per subset size, as in the case of the $R^2$ criterion). In this example, the three best regressions are found to be (1, 2), (1, 2, 4), and (1, 2, 3).

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class SelectionRegressionEx2 {
```

```
public static void main(String[] args)  throws Exception {
    double x[][] = {
        {7., 26., 6., 60.},
        {1., 29., 15., 52.},
        {11., 56., 8., 20.},
        {11., 31., 8., 47.},
        {7., 52., 6., 33.},
        {11., 55., 9., 22.},
        {3., 71., 17., 6.},
        {1., 31., 22., 44.},
        {2., 54., 18., 22.},
        {21., 47., 4., 26},
        {1., 40., 23., 34.},
        {11., 66., 9., 12.},
        {10.0, 68., 8., 12.}};

    double y[] = {
        78.5,
        74.3,
        104.3,
        87.6,
        95.9,
        109.2,
        102.7,
        72.5,
        93.1,
        115.9,
        83.8,
        113.3,
        109.4};

    String criterionOption;
    MessageFormat critMsg =
       new MessageFormat("Regressions with {0} variable(s) ({1})");
    MessageFormat critLabel =
       new MessageFormat("   Criterion               Variables");
    MessageFormat coefMsg = new MessageFormat("Best Regressions with" +
       " {0} variable(s) ({1})");
    MessageFormat coefLabel = new MessageFormat("Variable   Coefficient" +
       "   Standard Error  t-statistic   p-value");
    MessageFormat critData = new MessageFormat("{0}   {1}   {2}   {3}" +
       "   {4}   {5}");
```

```
SelectionRegression sr = new SelectionRegression(4);
sr.setCriterionOption(sr.MALLOWS_CP_CRITERION);
sr.setMaximumBestFound(3);
sr.compute(x, y);
SelectionRegression.Statistics stats = sr.getStatistics();

criterionOption = new String("R-squared");

for (int i=1; i <= 4; i++) {
    double[] tmpCrit = stats.getCriterionValues(i);
    int[][] indvar = stats.getIndependentVariables(i);

    Object p[] = {new Integer(i), criterionOption};
    System.out.println(critMsg.format(p));
    Object p1[] = {null};
    System.out.println(critLabel.format(p1));

    for (int j=0; j< tmpCrit.length; j++) {
        System.out.print("     "+tmpCrit[j]+"        ");
        for (int k = 0; k < indvar[j].length ; k++) {
            System.out.print(indvar[j][k]+"   ");
        }
        System.out.println("");
    }
    System.out.println("");
}

String tmp;
for (int i=0; i < 3; i++) {
    System.out.println("");

    double[][] tmpCoef= stats.getCoefficientStatistics(i);

    Object p[] = {new Integer(tmpCoef.length), criterionOption};
        System.out.println(coefMsg.format(p));
    Object p2[] = {null};
    System.out.println(coefLabel.format(p2));

    PrintMatrix pm = new PrintMatrix();
    pm.setColumnSpacing(10);
    NumberFormat nf = NumberFormat.getInstance();
```

```
                nf.setMinimumFractionDigits(4);
                PrintMatrixFormat tst = new PrintMatrixFormat();
                tst.setNoColumnLabels();
                tst.setNoRowLabels();
                tst.setNumberFormat(nf);
                pm.print(tst, tmpCoef);
                System.out.println("");
                System.out.println("");
            }
    }
}
```

### Output

```
Regressions with 1 variable(s) (R-squared)
   Criterion               Variables
     138.73083349167865        4
     142.48640693696262        2
     202.54876912345225        1
     315.15428414008386        3

Regressions with 2 variable(s) (R-squared)
   Criterion               Variables
     2.6782415983184293        1   2
     5.4958508247586515        1   4
     22.373111964697628        3   4
     138.2259197546432        2   4
     198.09465256959135        1   3

Regressions with 3 variable(s) (R-squared)
   Criterion               Variables
     3.0182334734873457        1   2   4
     3.041279723064166        1   2   3
     3.4968244423484762        1   3   4
     7.337473995655984        2   3   4

Regressions with 4 variable(s) (R-squared)
   Criterion               Variables
     5.0        1   2   3   4
```

```
Best Regressions with 2 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

1.0000          1.4683          0.1213          12.1047          0.0000
2.0000          0.6623          0.0459          14.4424          0.0000
```

```
Best Regressions with 3 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

1.0000           1.4519          0.1170          12.4100          0.0000
2.0000           0.4161          0.1856           2.2418          0.0517
4.0000          -0.2365          0.1733          -1.3650          0.2054
```

```
Best Regressions with 3 variable(s) (R-squared)
Variable   Coefficient   Standard Error  t-statistic   p-value

1.0000          1.6959          0.2046           8.2895          0.0000
2.0000          0.6569          0.0442          14.8508          0.0000
3.0000          0.2500          0.1847           1.3536          0.2089
```

## *class* **StepwiseRegression**

Builds multiple linear regression models using forward selection, backward selection, or stepwise selection.

Class `StepwiseRegression` builds a multiple linear regression model using forward selection, backward selection, or forward stepwise (with a backward glance) selection.

Levels of priority can be assigned to the candidate independent variables using the setLevels method. All variables with a priority level of 1 must enter the model before variables with a priority level of 2. Similarly, variables with a level of 2 must enter before variables with a level of 3, etc. Variables also can be forced into the model (setForce).

Note that specifying "force" without also specifying the levels will result in all variables being forced into the model.

Typically, the intercept is forced into all models and is not a candidate variable. In this case, a sum-of-squares and crossproducts matrix for the independent and dependent variables corrected for the mean is required. Other possibilities are as follows:

1. The intercept is not in the model. A raw (uncorrected) sum-of-squares and crossproducts matrix for the independent and dependent variables is required as input in `cov`. Argument `nObservations` must be set to one greater than the number of observations.
2. An intercept is a candidate variable. A raw (uncorrected) sum-of-squares and crossproducts matrix for the constant regressor (=1), independent and dependent variables are required for `cov`. In this case, `cov` contains one additional row and column corresponding to the constant regressor. This row/column contains the sum-of-squares and crossproducts of the constant regressor with the independent and dependent variables. The remaining elements in `cov` are the same as in the previous case. Argument `nObservations` must be set to one greater than the number of observations.

The stepwise regression algorithm is due to Efroymson (1960). `StepwiseRegression` uses sweeps of the covariance matrix (input in `cov`, if the covariance matrix is specified, or generated internally) to move variables in and out of the model (Hemmerle 1967, Chapter 3). The SWEEP operator discussed in Goodnight (1979) is used. A description of the stepwise algorithm is also given by Kennedy and Gentle (1980, pp. 335-340). The advantage of stepwise model building over all possible regression (com.imsl.stat.SelectionRegression) is that it is less demanding computationally when the number of candidate independent variables is very large. However, there is no guarantee that the model selected will be the best model (highest $R^2$) for any subset size of independent variables.

## Declaration

public class com.imsl.stat.StepwiseRegression
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* StepwiseRegression.CyclingIsOccurringException

Cycling is occurring.

### Declaration

public static class com.imsl.stat.StepwiseRegression.CyclingIsOccurringException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *StepwiseRegression.CyclingIsOccurringException*
  `public` **StepwiseRegression.CyclingIsOccurringException**`( int nStep )`
  - **Description**
    Constructs a `CyclingIsOccurringException`.
  - **Parameters**
    * `nStep` – An `int` which specifies the number of steps taken.

## *class* StepwiseRegression.NoVariablesEnteredException

No Variables can enter the model.

### Declaration

public static class com.imsl.stat.StepwiseRegression.NoVariablesEnteredException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *StepwiseRegression.NoVariablesEnteredException*
  `public` **StepwiseRegression.NoVariablesEnteredException**`( )`
  - **Description**
    Constructs a `NoVariablesEnteredException`.

## *class* StepwiseRegression.CoefficientTTests

`CoefficientTTests` contains statistics related to the student-$t$ test, for each regression coefficient.

### Declaration

public class com.imsl.stat.StepwiseRegression.CoefficientTTests
**extends** java.lang.Object
**implements** java.io.Serializable

### Methods

---

- *getCoefficient*
  `public double` **getCoefficient**`( int` **index** `)`
  - **Description**
    Returns the estimate for a coefficient of the independent variable.
  - **Parameters**
    * `index` – An `int` which specifies the index of the coefficient whose estimate is to be returned. `index` must be between 1 and the number of independent variables.
  - **Returns** – A `double` which contains the estimate for the coefficient.

---

- *getPValue*
  `public double` **getPValue**`( int` **index** `)`
  - **Description**
    Returns the $p$-value for the two-sided test $H_0 : \beta = 0$ vs. $H_1 : \beta \neq 0$.
  - **Parameters**
    * `index` – An `int` which specifies the index of the coefficient whose $p$-value is to be returned. `index` must be between 1 and the number of independent variables.
  - **Returns** – A `double` which contains the estimated $p$-value for the coefficient.

---

- *getStandardError*
  `public double` **getStandardError**`( int` **index** `)`
  - **Description**
    Returns the estimated standard error for a coefficient estimate.
  - **Parameters**
    * `index` – An `int` which specifies the index of the coefficient whose standard error estimate is to be returned. `index` must be between 1 and the number of independent variables.
  - **Returns** – A `double` which contains the estimated standard error for the coefficient.

- *getTStatistic*
  `public double` **getTStatistic**`( int` **index** `)`

  – **Description**
    Returns the student-$t$ test statistic for testing the $i$-th coefficient equal to zero ($\beta_{index} = 0$).
  – **Parameters**
    * `index` – An `int` which specifies the index of the coefficient whose $t$-test statistic is to be returned. `index` must be between 1 and the number of independent variables.
  – **Returns** – A `double` which contains the estimated $t$-test statistic for the coefficient.

## Fields

- public static final `int` **FORWARD_REGRESSION**
  – Indicates forward regression. An attempt is made to add a variable to the model. A variable is added if its $p$-value is less than `pValueIn`. During intitialization, only forced variables enter the model.
- public static final `int` **BACKWARD_REGRESSION**
  – Indicates backward regression. An attempt is made to remove a variable from the model. A variable is removed if its $p$-value exceeds `pValueOut`. During initialization, all candidate independent variables enter the model.
- public static final `int` **STEPWISE_REGRESSION**
  – Indicates stepwise regression. A backward step is attempted. After the backward step, a forward step is attempted. This is a stepwise step. Any forced variables enter the model during initialization.

## Constructors

- *StepwiseRegression*
  `public` **StepwiseRegression**`( double[][]` **x**`, double[]` **y** `) throws com.imsl.stat.Covariances.TooManyObsDeletedException, com.imsl.stat.Covariances.MoreObsDelThanEnteredException, com.imsl.stat.Covariances.DiffObsDeletedException`

– **Description**
Creates a new instance of `StepwiseRegression`.

– **Parameters**
  * `x` – A `double` matrix of *nObs* by *nVars*, where *nObs* is the number of observations and *nVars* is the number of independent variables.
  * `y` – A `double` array containing the observations of the dependent variable.

– **Throws**
  * `com.imsl.stat.Covariances.TooManyObsDeletedException` – is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative
  * `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – is thrown if more observations are being deleted from "variance-covariance" matrix than were originally entered. The corresponding row,column of the incidence matrix is less than zero.
  * `com.imsl.stat.Covariances.DiffObsDeletedException` – is thrown if different observations are being deleted than were originally entered

---

• *StepwiseRegression*
`public` **StepwiseRegression**`( double[][] x, double[] y, double[]` **weights** `) throws com.imsl.stat.Covariances.NonnegativeWeightException, com.imsl.stat.Covariances.TooManyObsDeletedException, com.imsl.stat.Covariances.MoreObsDelThanEnteredException, com.imsl.stat.Covariances.DiffObsDeletedException`

– **Description**
Creates a new instance of weighted `StepwiseRegression`.

– **Parameters**
  * `x` – A `double` matrix of *nObs* by *nVars*, where *nObs* is the number of observations and *nVars* is the number of independent variables.
  * `y` – A `double` array containing the observations of the dependent variable.
  * `weights` – A `double` array containing the weight for each observation of `x`.

– **Throws**
  * `com.imsl.stat.Covariances.NonnegativeWeightException` – is thrown if the weights are negative
  * `com.imsl.stat.Covariances.TooManyObsDeletedException` – is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative
  * `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – is thrown if more observations are being deleted from "variance-covariance" matrix than were originally entered. The corresponding row,column of the incidence matrix is less than zero.
  * `com.imsl.stat.Covariances.DiffObsDeletedException` – is thrown if different observations are being deleted than were originally entered

- *StepwiseRegression*
  `public` **StepwiseRegression**`(` `double[][]` **x**, `double[]` **y**, `double[]` **weights**, `double[]` **frequencies** `) throws com.imsl.stat.Covariances.NonnegativeFreqException, com.imsl.stat.Covariances.NonnegativeWeightException, com.imsl.stat.Covariances.TooManyObsDeletedException, com.imsl.stat.Covariances.MoreObsDelThanEnteredException, com.imsl.stat.Covariances.DiffObsDeletedException`

  - **Description**
    Creates a new instance of weighted `StepwiseRegression` using observation frequencies.
  - **Parameters**
    * `x` – A `double` matrix of *nObs* by *nVars*, where *nObs* is the number of observations and *nVars* is the number of independent variables.
    * `y` – A `double` array containing the observations of the dependent variable.
    * `weights` – A `double` array containing the weight for each observation of `x`.
    * `frequencies` – A `double` array containing the frequency for each row of `x`.
  - **Throws**
    * `com.imsl.stat.Covariances.NonnegativeFreqException` – is thrown if the frequencies are negative
    * `com.imsl.stat.Covariances.NonnegativeWeightException` – is thrown if the weights are negative
    * `com.imsl.stat.Covariances.TooManyObsDeletedException` – is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative
    * `com.imsl.stat.Covariances.MoreObsDelThanEnteredException` – is thrown if more observations are being deleted from "variance-covariance" matrix than were originally entered. The corresponding row,column of the incidence matrix is less than zero.
    * `com.imsl.stat.Covariances.DiffObsDeletedException` – is thrown if different observations are being deleted than were originally entered

- *StepwiseRegression*
  `public` **StepwiseRegression**`(` `double[][]` **cov**, `int` **nObservations** `)`

  - **Description**
    Creates a new instance of `StepwiseRegression` from a user-supplied variance-covariance matrix.
  - **Parameters**
    * `cov` – A `double` matrix containing a variance-covariance or sum of squares and crossproducts matrix, in which the last column must correspond to the

dependent variable. `cov` can be computed using the com.imsl.stat.Covariances class.

  * `nObservations` – An `int` containing the number of observations associated with `cov`.

## Methods

---

- *compute*

  `public void` **compute**`( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`

  - **Description**

    Builds the multiple linear regression models using forward selection, backward selection, or stepwise selection.

  - **Throws**

    * `com.imsl.stat.StepwiseRegression.NoVariablesEnteredException` – is thrown if no variables entered the model. All elements of com.imsl.stat.ANOVA table are set to `NaN`
    * `com.imsl.stat.StepwiseRegression.CyclingIsOccurringException` – is thrown if cycling occurs

---

- *getANOVA*

  `public synchronized ANOVA` **getANOVA**`( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`

  - **Description**

    Get an analysis of variance table and related statistics.

  - **Returns** – An com.imsl.stat.ANOVA table and related statistics.

---

- *getCoefficientTTests*

  `public StepwiseRegression.CoefficientTTests` **getCoefficientTTests**`( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`

  - **Description**

    Returns the student-$t$ test statistics for the regression coefficients.

  - **Returns** – A com.imsl.stat.StepwiseRegression.CoefficientTTests object containing statistics relating to the regression coefficients.

---

- *getCoefficientVIF*
  `public double[]` **getCoefficientVIF**`( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`
  - **Description**
    Returns the variance inflation factors for the final model in this invocation. The elements are in the same order as the independent variables in `x` (or, if the covariance matrix is specified, the elements are in the same order as the variables in `cov`). Each element corresponding to a variable not in the model contains statistics for a model which includes the variables of the final model and the variables corresponding to the element in question.
    The square of the multiple correlation coefficient for the $i$-th regressor after all others can be obtained from the $i$-th element for the returned array by the following formula:
    $$1.0 - \frac{1.0}{VIF}$$
  - **Returns** – A `double` array containing the variance inflation factors for the final model in this invocation.

---

- *getCovariancesSwept*
  `public double[][]` **getCovariancesSwept**`( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`
  - **Description**
    Returns the results after `cov` has been swept for the columns corresponding to the variables in the model.
  - **Returns** – A `double` matrix containing the results after `cov` has been swept on the columns corresponding to the variables in the model. The estimated variance-covariance matrix of the estimated regression coefficients in the final model can be obtained by extracting the rows and columns corresponding to the independent variables in the final model and multiplying the elements of this matrix by the error mean square.

---

- *getHistory*
  `public double[]` **getHistory**`( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`
  - **Description**
    Returns the stepwise regression history for the independent variables.

– **Returns** – A `double` array containing the recent history of the independent variables. The last element corresponds to the dependent variable.

| history[$i$] | Status of $i$-th Variable |
|---|---|
| 0.0 | This variable has never been added to the model. |
| 0.5 | This variable was added into the model during initialization. |
| $k>$ 0.0 | This variable was added to the model during the $k$-th step. |
| $k<$ 0.0 | This variable was deleted from model during the $k$-th step |

---

- *getSwept*
`public double[] getSwept( ) throws com.imsl.stat.StepwiseRegression.NoVariablesEnteredException, com.imsl.stat.StepwiseRegression.CyclingIsOccurringException`

  – **Description**
  Returns an array containing information indicating whether or not a particular variable is in the model.

  – **Returns** – A `double` array with information to indicate the independent variables in the model. The last element corresponds to the dependent variable. A +1 in the $i$-th position indicates that the variable is in the selected model. A -1 indicates that the variable is not in the selected model.

---

- *setForce*
`public void setForce( int force )`

  – **Description**
  Forces independent variables into the model based on their level assigned from `setlevels`.

  – **Parameters**
    * `force` – An `int` specifying the upper bound on the variables forced into the model. Variables with levels 1, 2, ..., `force` are forced into the model as independent variables.

---

- *setLevels*
`public void setLevels( int[] levels )`

  – **Description**
  Sets the levels of priority for variables entering and leaving the regression. Each variable is assigned a positive value which indicates its level of entry into the model. A variable can enter the model only after all variables with smaller

nonzero levels of entry have entered. Similarly, a variable can only leave the model after all variables with higher levels of entry have left. Variables with the same level of entry compete for entry (deletion) at each step. Argument `levels[i]=0` means the $i$-th variable never enters the model. Argument `levels[i]=-1` means the $i$-th variable is the dependent variable. The last element in `levels` must correspond to the dependent variable, except when the variance-covariance or sum of squares and crossproducts matrix is supplied.

– **Parameters**
  * `levels` – An `int` array containing the levels of entry into the model for each variable. Default: 1, 1, ..., 1, -1 where -1 corresponds to the dependent variable.

---

• *setMethod*
`public void` **setMethod**`( int` **method** `)`

– **Description**
Specifies the stepwise selection method, forward, backward, or stepwise Regression.

– **Parameters**
  * `method` – An `int` value between -1 and 1 specifying the stepwise selection method. Fields `FORWARD_REGRESSION`, `BACKWARD_REGRESSION`, and `STEPWISE_REGRESSION` should be used. Default: `STEPWISE_REGRESSION`.

---

• *setPValueIn*
`public void` **setPValueIn**`( double` **pValueIn** `)`

– **Description**
Defines the largest $p$-value for variables entering the model. Variables with $p$-value less than `pValueIn` may enter the model. Backward regression does not use this value.

– **Parameters**
  * `pValueIn` – A `double` containing the largest $p$-value for variables entering the model. Default: `pValueIn` = 0.05.

---

• *setPValueOut*
`public void` **setPValueOut**`( double` **pValueOut** `)`

– **Description**
Defines the smallest $p$-value for removing variables. Variables with $p$-values greater than `pValueOut` may leave the model. `pValueOut` must be greater than or equal to `pValueIn`. A common choice for `pValueOut` is 2*`pValueIn`. Forward regression does not use this value.

– **Parameters**

* pValueOut – A double containing the smallest $p$-value for removing variables from the model. Default: pValueOut = 0.10.

---

- *setTolerance*
  public void **setTolerance**( double **tolerance** )
  - **Description**
    The tolerance used to detect linear dependence among the independent variables.
  - **Parameters**
    * tolerance – A double containing the tolerance used for detecting linear dependence. Default: tolerance = 2.2204460492503e-16.

## Example 1: StepwiseRegression

This example uses a data set from Draper and Smith (1981, pp. 629-630). Method compute is invoked to find the best regression for each subset size using the $R^2$ criterion. By default, stepwise regression is performed.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.IMSLException.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class StepwiseRegressionEx1 {

   public static void main(String[] args)  throws Exception {
      double x[][] = {
         {7., 26., 6., 60.},
         {1., 29., 15., 52.},
         {11., 56., 8., 20.},
         {11., 31., 8., 47.},
         {7., 52., 6., 33.},
         {11., 55., 9., 22.},
         {3., 71., 17., 6.},
         {1., 31., 22., 44.},
         {2., 54., 18., 22.},
         {21., 47., 4., 26},
         {1., 40., 23., 34.},
         {11., 66., 9., 12.},
         {10.0, 68., 8., 12.}};
```

```
      double y[] = {
         78.5, 74.3, 104.3, 87.6, 95.9, 109.2, 102.7,
         72.5, 93.1, 115.9, 83.8, 113.3, 109.4};

         StepwiseRegression sr = new StepwiseRegression(x,y);
         sr.compute();

         PrintMatrix pm = new PrintMatrix();
         pm.setTitle("*** ANOVA *** "); pm.print(sr.getANOVA().getArray());

         StepwiseRegression.CoefficientTTests coefT =
            sr.getCoefficientTTests();
         double coef[][] = new double[4][4];
         for (int i=0; i<4; i++) {
            coef[i][0] = coefT.getCoefficient(i);
            coef[i][1] = coefT.getStandardError(i);
            coef[i][2] = coefT.getTStatistic(i);
            coef[i][3] = coefT.getPValue(i);
         }
         pm.setTitle("*** Coef *** "); pm.print(coef);
         pm.setTitle("*** Swept *** "); pm.print(sr.getSwept());
         pm.setTitle("*** History *** "); pm.print(sr.getHistory());
         pm.setTitle("*** VIF *** "); pm.print(sr.getCoefficientVIF());
         pm.setTitle("*** CovS *** "); pm.print(sr.getCovariancesSwept());
   }
}
```

## Output

```
*** ANOVA ***
        0
 0      2
 1     10
 2     12
 3  2,641.001
 4     74.762
 5  2,715.763
 6  1,320.5
 7      7.476
 8    176.627
 9      0
```

```
10     97.247
11     96.697
12      2.734
13      ?
14      ?

          *** Coef ***
     0       1       2       3
0   1.44   0.138   10.403  0
1   0.416  0.186    2.242  0.052
2  -0.41   0.199   -2.058  0.07
3  -0.614  0.049  -12.621  0

*** Swept ***
   0
0   1
1  -1
2  -1
3   1
4  -1

*** History ***
   0
0  2
1  0
2  0
3  1
4  0

*** VIF ***
     0
0   1.064
1  18.78
2   3.46
3   1.064

                 *** CovS ***
     0        1         2        3        4
0   0.003    -0.029    -0.946   0         1.44
1  -0.029   154.72   -142.8     0.907   64.381
2  -0.946  -142.8     142.302   0.07   -58.35
3   0          0.907     0.07    0       -0.614
```

```
4   1.44     64.381   -58.35   -0.614   74.762
```

# Chapter 14

# Analysis of Variance

## Classes



## Usage Notes

The classes described in this chapter are for commonly-used experimental designs. Typically, responses are stored in the input vector $y$ in a pattern that takes advantage of the balanced design structure. Consequently, the full set of model subscripts is not needed to identify each response. The classes assume the usual pattern, which requires that the last model subscript change most rapidly, followed by the model subscript next in line, and so forth, with the first subscript changing at the slowest rate. This pattern is referred to as *lexicographical ordering.*

`ANOVA` class allows missing responses if confidence interval information is not requested. Double.NaN (Not a Number) is the missing value code used by these classes. Any element of $y$ that is missing must be set to NaN. Other classes described in this chapter do not allow missing responses because the classes generally deal with balanced designs.

As a diagnostic tool for determination of the validity of a model, classes in this chapter typically perform a test for lack of fit when $n(n > 1)$ responses are available in each cell of the experimental design.

# *class* **ANOVA**

Analysis of Variance table and related statistics.

## Declaration

public class com.imsl.stat.ANOVA
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructors

---

- *ANOVA*
  `public` **ANOVA**`(` `double[][]` **y** `)`
  - **Description**
    Analyzes a one-way classification model.
  - **Parameters**
    * `y` – is a two-dimension `double` array containing the responses. The rows in `y` correspond to observation groups. Each row of `y` can contain a different number of observations.

---

- *ANOVA*
  `public` **ANOVA**`(` `double` **dfr**, `double` **ssr**, `double` **dfe**, `double` **sse**, `double` **gmean** `)`
  - **Description**
    Construct an analysis of variance table and related statistics. Intended for use by the LinearRegression class.
  - **Parameters**
    * `dfr` – a `double` scalar value representing the degrees of freedom for model
    * `ssr` – a `double` scalar value representing the sum of squares for model
    * `dfe` – a `double` scalar value representing the degrees of freedom for error
    * `sse` – a `double` scalar value representing the sum of squares for error
    * `gmean` – a `double` scalar value representing the grand mean. If the grand mean is not known it may be set to not-a-number.

## Methods

---

- *getAdjustedRSquared*
  `public double` **getAdjustedRSquared**`( )`

  – **Description**
    Returns the adjusted R-squared (in percent).
  – **Returns** – a `double` scalar value representing the adjusted R-squared (in percent)

---

- *getArray*
  `public double[]` **getArray**`( )`

  – **Description**
    Returns the ANOVA values as an array.
  – **Returns** – a `double`[15] array containing the following values:

| *index* | *Value* |
|---|---|
| 0 | Degrees of freedom for model |
| 1 | Degrees of freedom for error |
| 2 | Total degrees of freedom |
| 3 | Sum of squares for model |
| 4 | Sum of squares for error |
| 5 | Total sum of squares |
| 6 | Model mean square |
| 7 | Error mean square |
| 8 | F statistic |
| 9 | p-value |
| 10 | R-squared (in percent) |
| 11 | Adjusted R-squared (in percent) |
| 12 | Estimated standard deviation of the model error |
| 13 | Mean of the response (dependent variable) |
| 14 | Coefficient of variation (in percent) |

---

- *getCoefficientOfVariation*
  `public double` **getCoefficientOfVariation**`( )`

  – **Description**
    Returns the coefficient of variation (in percent).
  – **Returns** – a `double` scalar value representing the coefficient of variation (in percent)

---

- *getDegreesOfFreedomForError*
  `public double` **getDegreesOfFreedomForError**`( )`

– **Description**
Returns the degrees of freedom for error.

– **Returns** – a `double` scalar value representing the degrees of freedom for error

---

- *getDegreesOfFreedomForModel*
`public double` **getDegreesOfFreedomForModel**`( )`

– **Description**
Returns the degrees of freedom for model.

– **Returns** – a `double` scalar value representing the degrees of freedom for model

---

- *getDunnSidak*
`public double` **getDunnSidak**`( int` **i**`, int` **j** `)`

– **Description**
Computes the confidence interval of $i$-th mean - $j$-th mean, using the Dunn-Sidak method.

– **Parameters**
  * `i` – is a `int` indicating the $i$-th member of the pair, $\mu_i$
  * `j` – is a `int` indicating the $j$-th member of the pair, $\mu_j$

– **Returns** – the confidence intervals of $i$-th mean - $j$-th mean using the Dunn-Sidak method

---

- *getErrorMeanSquare*
`public double` **getErrorMeanSquare**`( )`

– **Description**
Returns the error mean square.

– **Returns** – a `double` scalar value representing the error mean square

---

- *getF*
`public double` **getF**`( )`

– **Description**
Returns the F statistic.

– **Returns** – a `double` scalar value representing the F statistic

---

- *getGroupInformation*
`public double[][]` **getGroupInformation**`( )`

– **Description**
Returns information concerning the groups.

– **Returns** – a two-dimension `double` array containing information concerning the groups. Row $i$ contains information pertaining to the $i$-th group. The information in the columns is as follows:

| *Column* | *Information* |
|---|---|
| 0 | Group Number |
| 1 | Number of nonmissing observations |
| 2 | Group Mean |
| 3 | Group Standard Deviation |

---

- *getMeanOfY*
`public double` **getMeanOfY**`( )`
  – **Description**
  Returns the mean of the response (dependent variable).
  – **Returns** – a `double` scalar value representing the mean of the response (dependent variable)

---

- *getModelErrorStdev*
`public double` **getModelErrorStdev**`( )`
  – **Description**
  Returns the estimated standard deviation of the model error.
  – **Returns** – a `double` scalar value representing the estimated standard deviation of the model error

---

- *getModelMeanSquare*
`public double` **getModelMeanSquare**`( )`
  – **Description**
  Returns the model mean square.
  – **Returns** – a `double` scalar value representing the model mean square

---

- *getP*
`public double` **getP**`( )`
  – **Description**
  Returns the p-value.
  – **Returns** – a `double` scalar value representing the $p$-value

---

- *getRSquared*
`public double` **getRSquared**`( )`
  – **Description**
  Returns the R-squared (in percent).

– **Returns** – a `double` scalar value representing the $R$-squared (in percent)

---

- *getSumOfSquaresForError*
  `public double` **getSumOfSquaresForError**`( )`
  – **Description**
  Returns the sum of squares for error.
  – **Returns** – a `double` scalar value representing the sum of squares for error

---

- *getSumOfSquaresForModel*
  `public double` **getSumOfSquaresForModel**`( )`
  – **Description**
  Returns the sum of squares for model.
  – **Returns** – a `double` scalar value representing the sum of squares for model

---

- *getTotalDegreesOfFreedom*
  `public double` **getTotalDegreesOfFreedom**`( )`
  – **Description**
  Returns the total degrees of freedom.
  – **Returns** – a `double` scalar value representing the total degrees of freedom

---

- *getTotalMissing*
  `public int` **getTotalMissing**`( )`
  – **Description**
  Returns the total number of missing values.
  – **Returns** – an `int` scalar value representing the total number of missing values (NaN) in input Y. Elements of Y containing NaN (not a number) are omitted from the computations.

---

- *getTotalSumOfSquares*
  `public double` **getTotalSumOfSquares**`( )`
  – **Description**
  Returns the total sum of squares.
  – **Returns** – a `double` scalar value representing the total sum of squares

### Example: ANOVA

This example computes a one-way analysis of variance for data discussed by Searle (1971, Table 5.1, pages 165-179). The responses are plant weights for 6 plants of 3 different types - 3 normal, 2 off-types, and 1 aberrant. The 3 normal plant weights are 101, 105, and 94. The 2 off-type plant weights are 84 and 88. The 1 aberrant plant weight is 32. Note in the results that for the group with only one response, the standard deviation is undefined and is set to NaN (not a number).

```
import com.imsl.stat.*;
import com.imsl.math.*;


public class ANOVAEx1 {
    public static void main(String args[]) {
        double y[][] = {
            {101, 105, 94},
            {84, 88},
            {32}
        };
        ANOVA anova = new ANOVA(y);
        double aov[] = anova.getArray();

        System.out.println("Degrees Of Freedom For Model = "+ aov[0]);
        System.out.println("Degrees Of Freedom For Error = "+ aov[1]);
        System.out.println("Total (Corrected) Degrees Of Freedom = "+ aov[2]);
        System.out.println("Sum Of Squares For Model = "+ aov[3]);
        System.out.println("Sum Of Squares For Error = "+ aov[4]);
        System.out.println("Total (Corrected) Sum Of Squares = "+ aov[5]);
        System.out.println("Model Mean Square = "+ aov[6]);
        System.out.println("Error Mean Square = "+ aov[7]);
        System.out.println("F statistic = "+ aov[8]);
        System.out.println("P value= "+ aov[9]);
        System.out.println("R Squared (in percent) = "+ aov[10]);
        System.out.println("Adjusted R Squared (in percent) = "+ aov[11]);
        System.out.println("Model Error Standard deviation = "+ aov[12]);
        System.out.println("Mean Of Y = "+ aov[13]);
        System.out.println("Coefficient Of Variation (in percent) = "+ aov[14]);
        System.out.println("Total number of missing values = " +
        anova.getTotalMissing());

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        String labels[] = { "Group", "N", "Mean", "Std. Deviation"};
```

```
        pmf.setColumnLabels(labels);
        pmf.setNumberFormat(null);
        new PrintMatrix("Group Information").print(pmf,
        anova.getGroupInformation());
    }
}
```

### Output

```
Degrees Of Freedom For Model = 2.0
Degrees Of Freedom For Error = 3.0
Total (Corrected) Degrees Of Freedom = 5.0
Sum Of Squares For Model = 3480.0
Sum Of Squares For Error = 70.0
Total (Corrected) Sum Of Squares = 3550.0
Model Mean Square = 1740.0
Error Mean Square = 23.333333333333332
F statistic = 74.57142857142857
P value= 0.0027688825253497917
R Squared (in percent) = 98.02816901408451
Adjusted R Squared (in percent) = 96.71361502347418
Model Error Standard deviation = 4.83045891539648
Mean Of Y = 84.0
Coefficient Of Variation (in percent) = 5.750546327852952
Total number of missing values = 0
           Group Information
   Group   N   Mean     Std. Deviation
0   0.0   3.0  100.0  5.5677643628300215
1   1.0   2.0   86.0  2.8284271247461903
2   2.0   1.0   32.0  NaN
```

## *class* **ANOVAFactorial**

Analyzes a balanced factorial design with fixed effects.

Class `ANOVAFactorial` performs an analysis for an $n$-way classification design with balanced data. For balanced data, there must be an equal number of responses in each cell of the $n$-way layout. The effects are assumed to be fixed effects. The model is an extension of the two-way model to include $n$ factors. The interactions (two-way, three-way, up to $n$-way) can be included in the model, or some of the higher-way interactions can be pooled into error. `setModelOrder` specifies the number of factors to be included in the highest-way interaction. For example, if three-way and higher-way interactions are to be pooled into error, specify `modelOrder = 2`. (By default, `modelOrder = nSubscripts - 1` with the last subscript being the error subscript.) `PURE_ERROR` indicates there are repeated responses within the $n$-way cell; `POOL_INTERACTIONS` indicates otherwise.

Class `ANOVAFactorial` requires the responses as input into a single vector `y` in lexicographical order, so that the response subscript associated with the first factor varies least rapidly, followed by the subscript associated with the second factor, and so forth. Hemmerle (1967, Chapter 5) discusses the computational method.

## Declaration

```
public class com.imsl.stat.ANOVAFactorial
extends java.lang.Object
implements java.io.Serializable, java.lang.Cloneable
```

## Fields

- public static final `int` **PURE_ERROR**
  - Indicates factor `nSubscripts` is error.
- public static final `int` **POOL_INTERACTIONS**
  - Indicates factor `nSubscripts` is not error.

## Constructor

- *ANOVAFactorial*
  `public` **ANOVAFactorial**`( int` **nSubscripts**`, int[]` **nLevels**`, double[]` **y** `)`
  - **Description**
    Constructor for `ANOVAFactorial`.
  - **Parameters**
    * `nSubscripts` – an `int` scalar containing the number of subscripts. Number of factors in the model + 1 (for the error term).

* nLevels – an int array of length nSubscripts containing the number of levels for each of the factors for the first nSubscripts-1 elements. nLevels[nSubscripts-1] is the number of observations per cell.
* y – a double array of length nLevels[0] * nLevels[1] * ... * nLevels[nSubscripts-1] containing the responses. y must not contain NaN for any of its elements, i.e., missing values are not allowed.

– **Throws**
  * java.lang.IllegalArgumentException – is thrown if nLevels.length, and y.length are not consistent

## Methods

---

- *compute*
  public final double **compute**( )
  – **Description**
  Analyzes a balanced factorial design with fixed effects.
  – **Returns** – a double scalar containing the $p$-value for the overall $F$ test

---

- *getANOVATable*
  public double[] **getANOVATable**( )
  – **Description**
  Returns the analysis of variance table.
  – **Returns** – a double array containing the analysis of variance table. The analysis of variance statistics are given as follows:

| **Element** | **Analysis of Variance Statistics** |
|---|---|
| 0 | degrees of freedom for the model |
| 1 | degrees of freedom for error |
| 2 | total (corrected) degrees of freedom |
| 3 | sum of squares for the model |
| 4 | sum of squares for error |
| 5 | total (corrected) sum of squares |
| 6 | model mean square |
| 7 | error mean square |
| 8 | overall $F$-statistic |
| 9 | $p$-value |
| 10 | $R^2$ (in percent) |
| 11 | adjusted $R^2$ (in percent) |
| 12 | estimate of the standard deviation |
| 13 | overall mean of y |
| 14 | coefficient of variation (in percent) |

- *getMeans*
`public double[]` **getMeans**`( )`
  - **Description**
    Returns the subgroup means.
  - **Returns** – a `double` array containing the subgroup means

- *getTestEffects*
`public double[][]` **getTestEffects**`( )`
  - **Description**
    Returns statistics relating to the sums of squares for the effects in the model.
  - **Returns** – a `double` matrix containing statistics relating to the sums of squares for the effects in the model. Here,

$$\mathrm{NEF} = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{\min(n, |\mathrm{model\_order}|)}$$

where `n` is given by `nSubscripts` if `ANOVAFactorial.POOL_INTERACTIONS` is specified; otherwise, `nSubscripts - 1`. Suppose the factors are A, B, C, and error. With `modelOrder = 3`, rows 0 through NEF-1 would correspond to A, B, C, AB, AC, BC, and ABC, respectively.
The columns of the output matrix are as follows:

| **Column** | **Description** |
|---|---|
| 0 | degrees of freedom |
| 1 | sum of squares |
| 2 | $F$-statistic |
| 3 | $p$-value |

- *setErrorIncludeType*
`public void` **setErrorIncludeType**`( int` **type** `)`
  - **Description**
    Sets error included type.
  - **Parameters**
    * `type` – an `int` scalar. `ANOVAFactorial.PURE_ERROR`, the default option, indicates factor `nSubscripts` is error. Its main effect and all its interaction effects are pooled into the error with the other (`modelOrder + 1`)-way and higher-way interactions. `ANOVAFactorial.POOL_INTERACTIONS` indicates factor `nSubscripts` is not error. Only (`modelOrder + 1`)-way and higher-way interactions are included in the error.

- *setModelOrder*
  public void **setModelOrder**( int **modelOrder** )
  - **Description**
    Sets the number of factors to be included in the highest-way interaction in the model.
  - **Parameters**
    * `modelOrder` – an `int` scalar containing the number of factors to be included in the highest-way interaction in the model. `modelOrder` must be in the interval [1, `nSubscripts - 1` ]. For example, a `modelOrder` of 1 indicates that a main effect model will be analyzed, and a `modelOrder` of 2 indicates that two-way interactions will be included in the model. Default: `modelOrder = nSubscripts - 1`

### Example 1: Two-way Analysis of Variance

A two-way analysis of variance is performed with balanced data discussed by Snedecor and Cochran (1967, Table 12.5.1, p. 347). The responses are the weight gains (in grams) of rats that were fed diets varying in the source (A) and level (B) of protein. The model is

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \varepsilon_{ijk} \quad i = 1,\ 2;\ j = 1,\ 2,\ 3;\ k = 1,\ 2,\ \ldots,\ 10$$

where

$$\sum_{i=1}^{2} \alpha_i = 0; \sum_{j=1}^{3} \beta_j = 0; \sum_{i=1}^{2} \gamma_{ij} = 0 \quad \text{for } j = 1,\ 2,\ 3;$$

and

$$\sum_{j=1}^{3} \gamma_{ij} = 0 \quad \text{for } j = 1,\ 2$$

The first responses in each cell in the two-way layout are given in the following table:

| | **Protein Source (A)** | | |
|---|---|---|---|
| **Protein Level (B)** | **Beef** | **Cereal** | **Pork** |
| High | 73, 102, 118, 104, 81, 107, 100, 87, 117, 111 | 98, 74, 56, 111, 95, 88, 82, 77, 86, 92 | 94, 79, 96, 98, 102, 102, 108, 91, 120, 105 |
| Low | 90, 76, 90, 64, 86, 51, 72, 90, 95, 78 | 107, 95, 97, 80, 98, 74, 74, 67, 89, 58 | 49, 82, 73, 86, 81, 97, 106, 70, 61, 82 |

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class ANOVAFactorialEx1 {
    public static void main(String args[]) {
        int nSubscripts = 3;
        int[] nLevels = {3, 2, 10};
        double[] y = {
            73.0, 102.0, 118.0, 104.0, 81.0, 107.0, 100.0, 87.0, 117.0, 111.0,
            90.0, 76.0, 90.0, 64.0, 86.0, 51.0, 72.0, 90.0, 95.0, 78.0,
            98.0, 74.0, 56.0, 111.0, 95.0, 88.0, 82.0, 77.0, 86.0, 92.0,
            107.0, 95.0, 97.0, 80.0, 98.0, 74.0, 74.0, 67.0, 89.0, 58.0,
            94.0, 79.0, 96.0, 98.0, 102.0, 102.0, 108.0, 91.0, 120.0, 105.0,
            49.0, 82.0, 73.0, 86.0, 81.0, 97.0, 106.0, 70.0, 61.0, 82.0
        };
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(6);
        ANOVAFactorial af = new ANOVAFactorial(nSubscripts, nLevels, y);

        System.out.println("P-value = " + nf.format(af.compute()));
    }
}
```

### Output

```
P-value = 0.002299
```

### Example 2: Two-way Analysis of Variance

In this example, the same model and data is fit as in the example 1, but additional information is printed.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class ANOVAFactorialEx2 {
```

```
public static void main(String args[]) {
    int nSubscripts = 3, i;
    int[] nLevels = {3, 2, 10};
    double[] y = {
        73.0, 102.0, 118.0, 104.0, 81.0, 107.0, 100.0, 87.0, 117.0, 111.0,
        90.0, 76.0, 90.0, 64.0, 86.0, 51.0, 72.0, 90.0, 95.0, 78.0,
        98.0, 74.0, 56.0, 111.0, 95.0, 88.0, 82.0, 77.0, 86.0, 92.0,
        107.0, 95.0, 97.0, 80.0, 98.0, 74.0, 74.0, 67.0, 89.0, 58.0,
        94.0, 79.0, 96.0, 98.0, 102.0, 102.0, 108.0, 91.0, 120.0, 105.0,
        49.0, 82.0, 73.0, 86.0, 81.0, 97.0, 106.0, 70.0, 61.0, 82.0
    };
    String[] labels = {
        "degrees of freedom for the model                 ",
        "degrees of freedom for error                    ",
        "total (corrected) degrees of freedom            ",
        "sum of squares for the model                  ",
        "sum of squares for error                     ",
        "total (corrected) sum of squares             ",
        "model mean square                              ",
        "error mean square                              ",
        "F-statistic                                      ",
        "p-value                                          ",
        "R-squared (in percent)                          ",
        "Adjusted R-squared (in percent)                 ",
        "est. standard deviation of the model error      ",
        "overall mean of y                               ",
        "coefficient of variation (in percent)           "
    };
    String[] rlabels = {"A", "B", "A*B"};
    String[] mlabels = {
        "grand mean     ", "A1             ", "A2             ",
        "A3             ", "B1             ", "B2             ",
        "A1*B1         ",  "A1*B2          ", "A2*B1          ",
        "A2*B2          ", "A3*B1          ", "A3*B2          "
    };
    NumberFormat nf = NumberFormat.getInstance();
    ANOVAFactorial af = new ANOVAFactorial(nSubscripts, nLevels, y);

    nf.setMinimumFractionDigits(6);
    System.out.println("P-value = " + nf.format(af.compute()));

    nf.setMaximumFractionDigits(4);
```

```
        System.out.println("\n          * * * Analysis of Variance * * *");
        double[] anova = af.getANOVATable();
        for (i = 0; i < anova.length; i++) {
            System.out.println(labels[i] + " " + nf.format(anova[i]));
        }

        System.out.println("\n          * * * Variation Due to the " +
        "Model * * *");
        System.out.println("Source\tDF\tSum of Squares\tMean Square" +
        "\tProb. of Larger F");
        double[][] te = af.getTestEffects();
        for (i = 0; i < te.length; i++) {
            System.out.println(rlabels[i] + "\t" + nf.format(te[i][0]) + "\t" +
            nf.format(te[i][1]) + "\t" + nf.format(te[i][2]) + "\t\t" +
            nf.format(te[i][3]));
        }

        System.out.println("\n* * * Subgroup Means * * *");
        double[] means = af.getMeans();
        for (i = 0; i < means.length; i++) {
            System.out.println(mlabels[i] + " " + nf.format(means[i]));
        }
    }
}
```

## Output

```
P-value = 0.002299

          * * * Analysis of Variance * * *
degrees of freedom for the model                5.0000
degrees of freedom for error                   54.0000
total (corrected) degrees of freedom           59.0000
sum of squares for the model                4,612.9333
sum of squares for error                   11,586.0000
total (corrected) sum of squares           16,198.9333
model mean square                             922.5867
error mean square                             214.5556
F-statistic                                     4.3000
p-value                                         0.0023
```

```
R-squared (in percent)                          28.4768
Adjusted R-squared (in percent)                 21.8543
est. standard deviation of the model error      14.6477
overall mean of y                               87.8667
coefficient of variation (in percent)           16.6704

          * * * Variation Due to the Model * * *
Source DF Sum of Squares Mean Square Prob. of Larger F
A 2.0000 266.5333 0.6211 0.5411
B 1.0000 3,168.2667 14.7666 0.0003
A*B 2.0000 1,178.1333 2.7455 0.0732

* * * Subgroup Means * * *
grand mean      87.8667
A1              89.6000
A2              84.9000
A3              89.1000
B1              95.1333
B2              80.6000
A1*B1          100.0000
A1*B2           79.2000
A2*B1           85.9000
A2*B2           83.9000
A3*B1           99.5000
A3*B2           78.7000
```

**Example 3: Three-way Analysis of Variance**

This example performs a three-way analysis of variance using data discussed by John (1971, pp. 91 92). The responses are weights (in grams) of roots of carrots grown with varying amounts of applied nitrogen (A), potassium (B), and phosphorus (C). Each cell of the three-way layout has one response. Note that the ABC interactions sum of squares, which is 186, is given incorrectly by John (1971, Table 5.2.) The three-way layout is given in the following table:

| | $A_0$ | | |
|---|---|---|---|
| | $B_0$ | $B_1$ | $B_2$ |
| $C_0$ | 88.76 | 91.41 | 97.85 |
| $C_1$ | 87.45 | 98.27 | 95.85 |
| $C_2$ | 86.01 | 104.20 | 90.09 |

| | $A_1$ | | |
|---|---|---|---|
| | $B_0$ | $B_1$ | $B_2$ |
| $C_0$ | 94.83 | 100.49 | 99.75 |
| $C_1$ | 84.57 | 97.20 | 112.30 |
| $C_2$ | 81.06 | 120.80 | 108.77 |

| | $A_2$ | | |
|---|---|---|---|
| | $B_0$ | $B_1$ | $B_2$ |
| $C_0$ | 99.90 | 100.23 | 104.50 |
| $C_1$ | 92.98 | 107.77 | 110.94 |
| $C_2$ | 94.72 | 118.39 | 102.87 |

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class ANOVAFactorialEx3 {
    public static void main(String args[]) {
        int nSubscripts = 3, i;
        int[] nLevels = {3, 3, 3};
        double[] y = {88.76, 87.45, 86.01, 91.41, 98.27, 104.2, 97.85, 95.85,
        90.09, 94.83, 84.57, 81.06, 100.49, 97.2, 120.8, 99.75, 112.3, 108.77,
        99.9, 92.98, 94.72, 100.23, 107.77, 118.39, 104.51, 110.94, 102.87};
        String[] labels = {
            "degrees of freedom for the model                ",
            "degrees of freedom for error                     ",
            "total (corrected) degrees of freedom            ",
            "sum of squares for the model                ",
            "sum of squares for error                      ",
            "total (corrected) sum of squares            ",
            "model mean square                              ",
            "error mean square                               ",
            "F-statistic                                      ",
            "p-value                                          ",
            "R-squared (in percent)                          ",
            "Adjusted R-squared (in percent)                 ",
            "est. standard deviation of the model error       ",
            "overall mean of y                               ",
```

```
    "coefficient of variation (in percent)             "
};
String[] rlabels = {"A", "B", "C", "A*B", "A*C", "B*C"};
NumberFormat nf = NumberFormat.getInstance();
ANOVAFactorial af = new ANOVAFactorial(nSubscripts, nLevels, y);

af.setErrorIncludeType(ANOVAFactorial.POOL_INTERACTIONS);
nf.setMinimumFractionDigits(6);
System.out.println("P-value = " + nf.format(af.compute()));

nf.setMaximumFractionDigits(4);

System.out.println("\n          * * * Analysis of Variance * * *");
double[] anova = af.getANOVATable();
for (i = 0; i < anova.length; i++) {
    System.out.println(labels[i] + " " + nf.format(anova[i]));
}

System.out.println("\n          * * * Variation Due to the " +
"Model * * *");
System.out.println("Source\tDF\tSum of Squares\tMean Square" +
"\tProb. of Larger F");
double[][] te = af.getTestEffects();
for (i = 0; i < te.length; i++) {
    StringBuffer sb = new StringBuffer(rlabels[i]);

    int len = sb.length();
    for(int j = 0; j < (8-len); j++) sb.append(' ');
    sb.append(nf.format(te[i][0]));

    len = sb.length();
    for(int j = 0; j < (16-len); j++) sb.append(' ');
    sb.append(nf.format(te[i][1]));

    len = sb.length();
    for(int j = 0; j < (32-len); j++) sb.append(' ');
    sb.append(nf.format(te[i][2]));

    len = sb.length();
    for(int j = 0; j < (48-len); j++) sb.append(' ');
    sb.append(nf.format(te[i][3]));
```

```
            System.out.println(sb.toString());
        }
    }
}
```

**Output**

```
P-value = 0.008299

          * * * Analysis of Variance * * *
degrees of freedom for the model                 18.0000
degrees of freedom for error                      8.0000
total (corrected) degrees of freedom             26.0000
sum of squares for the model                  2,395.7290
sum of squares for error                        185.7763
total (corrected) sum of squares              2,581.5052
model mean square                               133.0961
error mean square                                23.2220
F-statistic                                       5.7315
p-value                                           0.0083
R-squared (in percent)                           92.8036
Adjusted R-squared (in percent)                  76.6116
est. standard deviation of the model error        4.8189
overall mean of y                                98.9619
coefficient of variation (in percent)             4.8695

          * * * Variation Due to the Model * * *
Source DF Sum of Squares Mean Square Prob. of Larger F
A       2.0000  488.3675        10.5152        0.0058
B       2.0000  1,090.6564      23.4832        0.0004
C       2.0000  49.1485         1.0582         0.3911
A*B     4.0000  142.5853        1.5350         0.2804
A*C     4.0000  32.3474         0.3482         0.8383
B*C     4.0000  592.6238        6.3800         0.0131
```

# *class* MultipleComparisons

Performs Student-Newman-Keuls multiple comparisons test.

Class `MultipleComparisons` performs a multiple comparison analysis of means using the

Student-Newman-Keuls method. The null hypothesis is equality of all possible ordered subsets of a set of means. This null hypothesis is tested using the Studentized range of each of the corresponding subsets of sample means. The method is discussed in many elementary statistics texts, e.g., Kirk (1982, pp. 123-125).

## Declaration

public class com.imsl.stat.MultipleComparisons
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *MultipleComparisons*
  `public` **MultipleComparisons**`(` `double[]` **means**`,` `int` **df**`,` `double` **stdError** `)`
  - **Description**
    Constructor for `MultipleComparisons`.
  - **Parameters**
    * `means` – A `double` array containing the means.
    * `df` – An `int` scalar containing the degrees of freedom associated with `stdError`.
    * `stdError` – A `double` scalar containing the effective estimated standard error of a mean. In fixed effects models, `stdError` equals the estimated standard error of a mean. For example, in a one-way model stdError $= \sqrt{s^2/n}$ where $s^2$ is the estimate of $\sigma^2$ and n is the number of responses in a sample mean. In models with random components, use stdError $=$ sedif$/\sqrt{2}$ where `sedif` is the estimated standard error of the difference of two means.

## Methods

- *compute*
  `public final int[]` **compute**`( )`
  - **Description**
    Performs Student-Newman-Keuls multiple comparisons test.

– **Returns** – An `int` array , call it `equalMeans` indicating the size of the groups of means declared to be equal. Value `equalMeans[I] = J` indicates the *I*-th smallest mean and the next *J*-1 larger means are declared equal. Value `equalMeans[I] = 0` indicates no group of means starts with the *I*-th smallest mean.

---

- *setAlpha*
  `public void setAlpha( double alpha )`

  – **Description**
    Sets the significance level of the test

  – **Parameters**
    * `alpha` – A `double` scalar containing the significance level of test. `alpha` must be in the interval [0.01, 0.10]. Default: alpha = 0.01

### Example: Multiple Comparisons Test

A multiple-comparisons analysis is performed using data discussed by Kirk (1982, pp. 123-125). The results show that there are three groups of means with three separate sets of values: (36.7, 40.3, 43.4), (40.3, 43.4, 47.2), and (43.4, 47.2, 48.7).

```
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class MultipleComparisonsEx1 {
    public static void main(String args[]) {
        double[] means = {36.7, 48.7, 43.4, 47.2, 40.3};

        /* Perform multiple comparisons tests */
        MultipleComparisons mc = new MultipleComparisons(means, 45, 1.6970563);

        new PrintMatrix("Size of Groups of Means").print(mc.compute());
    }
}
```

### Output

```
Size of Groups of Means
   0
0  3
1  3
2  3
```

3 0

# Chapter 15

# Categorical and Discrete Data Analysis

**Classes**



## Usage Notes

The `ContingencyTable` class computes many statistics of interest in a two-way table. Statistics computed by this routine include the usual chi-squared statistics, measures of association, Kappa, and many others.

The `CategoricalGenLinModel` class is concerned with generalized linear models in discrete data. This routine may be used to compute estimates and associated statistics in probit, logistic, minimum extreme value, Poisson, negative binomial (with known number of successes), and logarithmic models. Classification variables as well as weights, frequencies, and additive constants may be used so that quite general linear models can be fit. Residuals, a measure of influence, the coefficient estimates, and other statistics are returned for each model fit. When infinite parameter estimates are required, extended maximum likelihood estimation may be used. Log-linear models may be fit through the use of Poisson regression models. Results from Poisson regression models involving structural and sampling zeros will be identical to the results obtained from the log-linear model but will be fit by a quasi-Newton algorithm rather than through iterative proportional fitting.

## *class* ContingencyTable

Performs a chi-squared analysis of a two-way contingency table.

Class `ContingencyTable` computes statistics associated with an $r \times c$ contingency table. The function computes the chi-squared test of independence, expected values, contributions to chi-squared, row and column marginal totals, some measures of association, correlation, prediction, uncertainty, the McNemar test for symmetry, a test for linear trend, the odds and the log odds ratio, and the kappa statistic (if the appropriate optional arguments are selected).

**Notation**

Let $x_{ij}$ denote the observed cell frequency in the $ij$ cell of the table and $n$ denote the total count in the table. Let $p_{ij} = p_{i\bullet} p_{j\bullet}$ denote the predicted cell probabilities under the null hypothesis of independence, where $p_{i\bullet}$ and $p_{j\bullet}$ are the row and column marginal relative frequencies. Next, compute the expected cell counts as $e_{ij} = np_{ij}$.

Also required in the following are $a_{uv}$ and $b_{uv}$ for $u, v = 1, \ldots, n$. Let $(r_s, c_s)$ denote the row and column response of observation $s$. Then, $a_{uv} = 1, 0$, or -1, depending on whether $r_u < r_v, r_u = r_v$, or $r_u > r_v$, respectively. The $b_{uv}$ are similarly defined in terms of the $c_s$ variables.

**Chi-squared Statistic**

For each cell in the table, the contribution to $\chi^2$ is given as $(x_{ij} - e_{ij})^2/e_{ij}$. The Pearson chi-squared statistic (denoted $\chi^2$) is computed as the sum of the cell contributions to chi-squared. It has *(r - 1) (c - 1)* degrees of freedom and tests the null hypothesis of independence, i.e., $H_0 : p_{ij} = p_{i\bullet} p_{j\bullet}$. The null hypothesis is rejected if the computed value of $\chi^2$ is too large.

The maximum likelihood equivalent of $\chi^2, G^2$ is computed as follows:

$$G^2 = -2 \sum_{i,j} x_{ij} \ln \left( x_{ij}/np_{ij} \right)$$

$G^2$ is asymptotically equivalent to $\chi^2$ and tests the same hypothesis with the same degrees of freedom.

**Measures Related to Chi-squared (Phi, Contingency Coefficient, and Cramer's V)**

There are three measures related to chi-squared that do not depend on sample size:

$$\text{phi, } \phi = \sqrt{\chi^2/n}$$

$$\text{contingency coefficient, } P = \sqrt{\chi^2/\left(n + \chi^2\right)}$$

$$\text{Cramer's } V, \ V = \sqrt{\chi^2/\left(n \min\left(r, c\right)\right)}$$

Since these statistics do not depend on sample size and are large when the hypothesis of independence is rejected, they can be thought of as measures of association and can be compared across tables with different sized samples. While both $P$ and $V$ have a range between 0.0 and 1.0, the upper bound of $P$ is actually somewhat less than 1.0 for any given table (see Kendall and Stuart 1979, p. 587). The significance of all three statistics is the same as that of the $\chi^2$ statistic, return value from the `getChiSquared` method.

The distribution of the $\chi^2$ statistic in finite samples approximates a chi-squared distribution. To compute the exact mean and standard deviation of the $\chi^2$ statistic, Haldane (1939) uses the multinomial distribution with fixed table marginals. The exact mean and standard deviation generally differ little from the mean and standard deviation of the associated chi-squared distribution.

**Standard Errors and p-values for Some Measures of Association**

In Columns 1 through 4 of statistics, estimated standard errors and asymptotic $p$-values are reported. Estimates of the standard errors are computed in two ways. The first estimate, in Column 1 of the return matrix from the `getStatistics` method, is asymptotically valid for any value of the statistic. The second estimate, in Column 2 of the array, is only correct under the null hypothesis of no association. The $z$-scores in Column 3 of statistics are computed using this second estimate of the standard errors. The $p$-values in Column 4 are computed from this $z$-score. See Brown and Benedetti (1977) for a discussion and formulas for the standard errors in Column 2.

**Measures of Association for Ranked Rows and Columns**

The measures of association, $\phi$, $P$, and $V$, do not require any ordering of the row and column categories. Class `ContingencyTable` also computes several measures of association for tables in which the rows and column categories correspond to ranked observations. Two of these tests, the product-moment correlation and the Spearman correlation, are correlation coefficients computed using assigned scores for the row and column categories. The cell indices are used for the product-moment correlation, while the average of the tied ranks of the row and column marginals is used for the Spearman rank correlation. Other scores are possible.

Gamma, Kendall's $\tau_b$, Stuart's $\tau_c$, and Somers' $D$ are measures of association that are computed like a correlation coefficient in the numerator. In all these measures, the numerator is computed as the "covariance" between the $a_{uv}$ variables and $b_{uv}$ variables defined above, i.e., as follows:

$$\sum_u \sum_v a_{uv} b_{uv}$$

Recall that $a_{uv}$ and $b_{uv}$ can take values -1, 0, or 1. Since the product $a_{uv}b_{uv} = 1$ only if $a_{uv}$ and $b_{uv}$ are both 1 or are both -1, it is easy to show that this "covariance" is twice the total number of agreements minus the number of disagreements, where a disagreement occurs when $a_{uv}b_{uv} = -1$.

Kendall's $\tau_b$ is computed as the correlation between the $a_{uv}$ variables and the $b_{uv}$ variables (see Kendall and Stuart 1979, p. 593). In a rectangular table ($r \neq c$), Kendall's $\tau_b$ cannot be 1.0 (if all marginal totals are positive). For this reason, Stuart suggested a modification to the denominator of $\tau$ in which the denominator becomes the largest possible value of the "covariance." This maximizing value is approximately $n^2m/(m-1)$, where $m = min$ *(r, c)*. Stuart's $\tau_c$ uses this approximate value in its denominator. For large $n, \tau_c \approx m\tau_b/(m-1)$.

Gamma can be motivated in a slightly different manner. Because the "covariance" of the $a_{uv}$ variables and the $b_{uv}$ variables can be thought of as twice the number of agreements minus the disagreements, *2(A - D)*, where $A$ is the number of agreements and $D$ is the number of disagreements, Gamma is motivated as the probability of agreement minus the probability of disagreement, given that either agreement or disagreement occurred. This is shown as $\gamma = (A - D)/(A + D)$.

Two definitions of Somers' $D$ are possible, one for rows and a second for columns. Somers' $D$ for rows can be thought of as the regression coefficient for predicting $a_{uv}$ from $b_{uv}$. Moreover, Somer's $D$ for rows is the probability of agreement minus the probability of disagreement, given that the column variable, $b_{uv}$, is not 0. Somers' $D$ for columns is defined in a similar manner.

A discussion of all of the measures of association in this section can be found in Kendall and Stuart (1979, p. 592).

**Measures of Prediction and Uncertainty**

**Optimal Prediction Coefficients:** The measures in this section do not require any ordering of the row or column variables. They are based entirely upon probabilities. Most are discussed in Bishop et al. (1975, p. 385).

Consider predicting (or classifying) the column for a given row in the table. Under the null hypothesis of independence, choose the column with the highest column marginal probability for all rows. In this case, the probability of misclassification for any row is 1 minus this marginal probability. If independence is not assumed within each row, choose the column with the highest row conditional probability. The probability of misclassification for the row becomes 1 minus this conditional probability.

Define the optimal prediction coefficient $\lambda_{c|r}$ for predicting columns from rows as the proportion of the probability of misclassification that is eliminated because the random variables are not independent. It is estimated by

$$\lambda_{c\,|\,r} = \frac{(1 - p_{\bullet m}) - (1 - \sum_i p_{im})}{1 - p_{\bullet m}}$$

where $m$ is the index of the maximum estimated probability in the row ($p_{im}$) or row margin ($p_{\bullet m}$). A similar coefficient is defined for predicting the rows from the columns. The symmetric version of the optimal prediction $\lambda$ is obtained by summing the numerators and denominators of $\lambda_{r|c}$ and $\lambda_{c|r}$ then dividing. Standard errors for these coefficients are given in Bishop et al. (1975, p. 388).

A problem with the optimal prediction coefficients $\lambda$ is that they vary with the marginal probabilities. One way to correct this is to use row conditional probabilities. The optimal prediction $\lambda*$ coefficients are defined as the corresponding $\lambda$ coefficients in which first the row (or column) marginals are adjusted to the same number of observations. This yields

$$\lambda^*_{c\,|\,r} = \frac{\sum_i \max_j p_{j\,|\,i} - \max_j (\sum_i p_{j\,|\,i})}{R - \max_j (\sum_i p_{j\,|\,i})}$$

where $i$ indexes the rows, $j$ indexes the columns, and $p_{j|i}$ is the (estimated) probability of column $j$ given row $i$.

$$\lambda^*_{r\,|\,c}$$

is similarly defined.

**Goodman and Kruskal $\tau$:** A second kind of prediction measure attempts to explain the proportion of the explained variation of the row (column) measure given the column (row) measure. Define the total variation in the rows as follows:

$$n/2 - (\sum_i x^2_{i\bullet}) / \, (2n)$$

Note that this is *1/(2n)* times the sums of squares of the $a_{uv}$ variables.

With this definition of variation, the Goodman and Kruskal $\tau$ coefficient for rows is computed as the reduction of the total variation for rows accounted for by the columns, divided by the total variation for the rows. To compute the reduction in the total variation of the rows accounted for by the columns, note that the total variation for the rows within column j is defined as follows:

$$q_j = x_{\bullet j}/2 - (\sum_i x_{ij}^2)/(2x_{i\bullet})$$

The total variation for rows within columns is the sum of the $q_j$ variables. Consistent with the usual methods in the analysis of variance, the reduction in the total variation is given as the difference between the total variation for rows and the total variation for rows within the columns.

Goodman and Kruskal's $\tau$ for columns is similarly defined. See Bishop et al. (1975, p. 391) for the standard errors.

**Uncertainty Coefficients:** The uncertainty coefficient for rows is the increase in the log-likelihood that is achieved by the most general model over the independence model, divided by the marginal log-likelihood for the rows. This is given by the following equation:

$$U_{r|c} = \frac{\sum_{i,j} x_{ij} \log\left(x_{i\bullet} x_{\bullet j}/n x_{ij}\right)}{\sum_i x_{i\bullet} \log\left(x_{i\bullet}/n\right)}$$

The uncertainty coefficient for columns is similarly defined. The symmetric uncertainty coefficient contains the same numerator as $U_{r|c}$ and $U_{c|r}$but averages the denominators of these two statistics. Standard errors for $U$ are given in Brown (1983).

**Kruskal-Wallis:** The Kruskal-Wallis statistic for rows is a one-way analysis-of-variance-type test that assumes the column variable is monotonically ordered. It tests the null hypothesis that no row populations are identical, using average ranks for the column variable. The Kruskal-Wallis statistic for columns is similarly defined. Conover (1980) discusses the Kruskal-Wallis test.

**Test for Linear Trend:** When there are two rows, it is possible to test for a linear trend in the row probabilities if it is assumed that the column variable is monotonically ordered. In this test, the probabilities for row 1 are predicted by the column index using weighted simple linear regression. This slope is given by

$$\hat{\beta} = \frac{\sum_j x_{\bullet j}\left(x_{1j}/x_{\bullet j} - x_{1\bullet}/n\right)(j - \bar{j})}{\sum_j x_{\bullet j}(j - \bar{j})^2}$$

where

$$\bar{j} = \sum_j x_{\bullet j} j / n$$

is the average column index. An asymptotic test that the slope is 0 may then be obtained (in large samples) as the usual regression test of zero slope.

In two-column data, a similar test for a linear trend in the column probabilities is computed. This test assumes that the rows are monotonically ordered.

**Kappa:** Kappa is a measure of agreement computed on square tables only. In the kappa statistic, the rows and columns correspond to the responses of two judges. The judges agree along the diagonal and disagree off the diagonal. Let

$$p_0 = \sum_i x_{ii} / n$$

denote the probability that the two judges agree, and let

$$p_c = \sum_i e_{ii} / n$$

denote the expected probability of agreement under the independence model. Kappa is then given by $(p_0 - p_c)/(1 - p_c)$.

**McNemar Tests:** The McNemar test is a test of symmetry in a square contingency table. In other words, it is a test of the null hypothesis $H_0 : \theta_{ij} = \theta_{ji}$. The multiple degrees-of-freedom version of the McNemar test with *r (r - 1)/2* degrees of freedom is computed as follows:

$$\sum_{i<j} \frac{(x_{ij} - x_{ji})^2}{(x_{ij} + x_{ji})}$$

The single degree-of-freedom test assumes that the differences, $x_{ij} - x_{ji}$, are all in one direction. The single degree-of-freedom test will be more powerful than the multiple degrees-of-freedom test when this is the case. The test statistic is given as follows:

$$\frac{\left( \sum_{i<j} (x_{ij} - x_{ji}) \right)^2}{\sum_{i<j} (x_{ij} + x_{ji})}$$

The exact probability can be computed by the binomial distribution.

## Declaration

public class com.imsl.stat.ContingencyTable
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

---

- *ContingencyTable*
  `public` **ContingencyTable**`( double[][]` **table** `)`
  - **Description**
    Constructs and performs a chi-squared analysis of a two-way contingency table.
  - **Parameters**
    * `table` – A `double` matrix containing the observed counts in the contingency table.

## Methods

---

- *getChiSquared*
  `public double` **getChiSquared**`( )`
  - **Description**
    Returns the Pearson chi-squared test statistic.
  - **Returns** – A `double` scalar containing the Pearson chi-squared test statistic.

---

- *getContingencyCoef*
  `public double` **getContingencyCoef**`( )`
  - **Description**
    Returns contingency coefficient.
  - **Returns** – A `double` scalar containing the contingency coefficient based on Pearson chi-squared statistic.

---

- *getContributions*
  `public double[][]` **getContributions**`( )`
  - **Description**
    Returns the contributions to chi-squared for each cell in the table.

- **Returns** – A `double` matrix of size (`table.length`+1) * (`table[0].length`+1) containing the contributions to chi-squared for each cell in the table. The last row and column contain the total contribution to chi-squared for that row or column.

---

- *getCramersV*
  `public double` **getCramersV**`( )`
  - **Description**
    Returns Cramer's V.
  - **Returns** – A `double` scalar containing the Cramer's V based on Pearson chi-squared statistic.

---

- *getDegreesOfFreedom*
  `public int` **getDegreesOfFreedom**`( )`
  - **Description**
    Returns the degrees of freedom for the chi-squared tests associated with the table.
  - **Returns** – An `int` scalar containing the degrees of freedom for the chi-squared tests associated with the table.

---

- *getExactMean*
  `public double` **getExactMean**`( )`
  - **Description**
    Returns exact mean.
  - **Returns** – A `double` scalar containing the exact mean based on Pearson's chi-square statistic.

---

- *getExactStdev*
  `public double` **getExactStdev**`( )`
  - **Description**
    Returns exact standard deviation.
  - **Returns** – A `double` scalar containing the exact standard deviation based on Pearson's chi-square statistic.

---

- *getExpectedValues*
  `public double[][]` **getExpectedValues**`( )`
  - **Description**
    Returns the expected values of each cell in the table.

– **Returns** – A `double` matrix of size (`table.length`+1) * (`table[0].length`+1) containing the expected values of each cell in the table, under the null hypothesis. The marginal totals are in the last row and column.

---

- *getGSquared*
  `public double` **getGSquared**`( )`

  – **Description**
  Returns the likelihood ratio $G^2$ (chi-squared).

  – **Returns** – A `double` scalar containing the likelihood ratio $G^2$ (chi-squared).

---

- *getGSquaredP*
  `public double` **getGSquaredP**`( )`

  – **Description**
  Returns the probability of a larger $G^2$ (chi-squared).

  – **Returns** – A `double` scalar containing the probability of a larger $G^2$ (chi-squared).

---

- *getP*
  `public double` **getP**`( )`

  – **Description**
  Returns the Pearson chi-squared $p$-value for independence of rows and columns.

  – **Returns** – A `double` scalar containing the Pearson chi-squared $p$-value for independence of rows and columns.

---

- *getPhi*
  `public double` **getPhi**`( )`

  – **Description**
  Returns phi.

  – **Returns** – A `double` scalar containing the phi based on Pearson chi-squared statistic.

---

- *getStatistics*
  `public double[][]` **getStatistics**`( )`

  – **Description**
  Returns the statistics associated with this table.

  – **Returns** – A `double` matrix of size 23 * 5 containing statistics associated with this table. Each row corresponds to a statistic.

| Row | Statistics |
|---|---|
| 0 | gamma |
| 1 | Kendall's $\tau_b$ |
| 2 | Stuart's $\tau_c$ |
| 3 | Somers' D for rows (given columns) |
| 4 | Somers' D for columns (given rows) |
| 5 | product moment correlation |
| 6 | Spearman rank correlation |
| 7 | Goodman and Kruskal $\tau$ for rows (given columns) |
| 8 | Goodman and Kruskal $\tau$ for columns (given rows) |
| 9 | uncertainty coefficient $U$ (symmetric) |
| 10 | uncertainty $U_{r\|c}$ (rows) |
| 11 | uncertainty $U_{c\|r}$ (columns) |
| 12 | optimal prediction $\lambda$ (symmetric) |
| 13 | optimal prediction $\lambda_{r\|c}$ (rows) |
| 14 | optimal prediction $\lambda_{c\|r}$ (columns) |
| 15 | optimal prediction $\lambda^*_{r\|c}$ (rows) |
| 16 | optimal prediction $\lambda^*_{c\|r}$ (columns) |
| 17 | test for linear trend in row probabilities if `table.length` = 2. If `table.length` is not 2, a test for linear trend in column probabilities if table[0].length = 2. |
| 18 | Kruskal-Wallis test for no row effect |
| 19 | Kruskal-Wallis test for no column effect |
| 20 | kappa (square tables only) |
| 21 | McNemar test of symmetry (square tables only) |
| 22 | McNemar one degree of freedom test of symmetry (square tables only) |

If a statistic cannot be computed, or if some value is not relevant for the computed statistic, the entry is NaN (Not a Number).

The columns are as follows:

| Column | Value |
|---|---|
| 0 | estimated statistic |
| 1 | standard error for any parameter value |
| 2 | standard error under the null hypothesis |
| 3 | $t$ value for testing the null hypothesis |
| 4 | $p$-value of the test in column 3 |

In the McNemar tests, column 0 contains the statistic, column 1 contains the chi-squared degrees of freedom, column 3 contains the exact $p$-value (1 degree of freedom only), and column 4 contains the chi-squared asymptotic $p$-value. The Kruskal-Wallis test is the same except no exact $p$-value is computed.

### Example 1: Contingency Table

The following example is taken from Kendall and Stuart (1979) and involves the distance vision in the right and left eyes.

```
import com.imsl.stat.*;

public class ContingencyTableEx1 {
    public static void main(String args[]) {
        double[][] table = {
            {821, 112, 85, 35},
            {116, 494, 145, 27},
            {72, 151, 583, 87},
            {43, 34, 106, 331}
        };
        ContingencyTable ct = new ContingencyTable(table);
        System.out.println("P-value = " + ct.getP());
    }
}
```

### Output

```
P-value = 0.0
```

### Example 2: Contingency Table

The following example, which illustrates the use of Kappa and McNemar tests, uses the same distance vision data as in Example 1.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.*;

public class ContingencyTableEx2 {
    public static void main(String args[]) {
        double[][] table = {
            {821.0, 112.0, 85.0, 35.0},
            {116.0, 494.0, 145.0, 27.0},
            {72.0, 151.0, 583.0, 87.0},
            {43.0, 34.0, 106.0, 331.0}
        };
        String[] rlabels = {"Gamma", "Tau B", "Tau C", "D-Row", "D-Column",
        "Correlation", "Spearman", "GK tau rows", "GK tau cols.", "U - sym.",
        "U - rows", "U - cols.", "Lambda-sym.", "Lambda-row", "Lambda-col.",
        "l-star-rows", "l-star-col.", "Lin. trend", "Kruskal row",
        "Kruskal col.", "Kappa", "McNemar", "McNemar df=1"};
        ContingencyTable ct = new ContingencyTable(table);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(4);

        System.out.println("Pearson chi-squared statistic = " +
        nf.format(ct.getChiSquared()));
        System.out.println("p-value for Pearson chi-squared = " +
        nf.format(ct.getP()));
        System.out.println("degrees of freedom = " + ct.getDegreesOfFreedom());
        System.out.println("G-squared statistic = " +
        nf.format(ct.getGSquared()));
        System.out.println("p-value for G-squared = " +
        nf.format(ct.getGSquaredP()));
        System.out.println("degrees of freedom = " + ct.getDegreesOfFreedom());

        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);
        PrintMatrix pm = new PrintMatrix("\n* * * Table Values * * *");
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
```

```
pm.print(pmf, table);

pm.setTitle("* * * Expected Values * * *");
pm.print(pmf, ct.getExpectedValues());

nf.setMinimumFractionDigits(4);
pmf.setNumberFormat(nf);
pm.setTitle("* * * Contributions to Chi-squared* * *");
pm.print(pmf, ct.getContributions());

nf.setMinimumFractionDigits(4);
System.out.println("* * * Chi-square Statistics * * *");
System.out.println("Exact mean = " + nf.format(ct.getExactMean()));
System.out.println("Exact standard deviation = " +
nf.format(ct.getExactStdev()));
System.out.println("Phi = " + nf.format(ct.getPhi()));
System.out.println("P = " + nf.format(ct.getContingencyCoef()));
System.out.println("Cramer's V = " + nf.format(ct.getCramersV()));

System.out.println("\n             stat.      std. err.   " +
"std. err.(Ho) t-value(Ho)  p-value");
double[][] stat = ct.getStatistics();
for (int i = 0; i < stat.length; i++) {
    StringBuffer sb = new StringBuffer(rlabels[i]);

    int len = sb.length();
    for(int j = 0; j < (13-len); j++) sb.append(' ');
    sb.append(nf.format(stat[i][0]));

    len = sb.length();
    for(int j = 0; j < (24-len); j++) sb.append(' ');
    sb.append(nf.format(stat[i][1]));

    len = sb.length();
    for(int j = 0; j < (36-len); j++) sb.append(' ');
    sb.append(nf.format(stat[i][2]));

    len = sb.length();
    for(int j = 0; j < (50-len); j++) sb.append(' ');
    sb.append(nf.format(stat[i][3]));

    len = sb.length();
```

```
            for(int j = 0; j < (63-len); j++) sb.append(' ');
            sb.append(nf.format(stat[i][4]));

            System.out.println(sb.toString());
        }
    }
}
```

### Output

```
Pearson chi-squared statistic = 3,304.3684
p-value for Pearson chi-squared = 0.0000
degrees of freedom = 9
G-squared statistic = 2,781.0190
p-value for G-squared = 0.0000
degrees of freedom = 9

* * * Table Values * * *
     0       1       2       3
0  821.00  112.00   85.00   35.00
1  116.00  494.00  145.00   27.00
2   72.00  151.00  583.00   87.00
3   43.00   34.00  106.00  331.00

          * * * Expected Values * * *
      0        1       2       3        4
0    341.69  256.92  298.49  155.90  1,053.00
1    253.75  190.80  221.67  115.78    782.00
2    289.77  217.88  253.14  132.21    893.00
3    166.79  125.41  145.70   76.10    514.00
4  1,052.00  791.00  919.00  480.00  3,242.00

          * * * Contributions to Chi-squared* * *
       0          1         2          3           4
0    672.3626   81.7416  152.6959     93.7612  1,000.5613
1     74.7802  481.8351   26.5189     68.0768    651.2109
2    163.6605   20.5287  429.8489     15.4625    629.5006
3     91.8743   66.6263   10.8183    853.7768  1,023.0957
4  1,002.6776  650.7317  619.8819  1,031.0772  3,304.3684

* * * Chi-square Statistics * * *
```

```
Exact mean = 9.0028
Exact standard deviation = 4.2402
Phi = 1.0096
P = 0.7105
Cramer's V = 0.5829

             stat.       std. err.   std. err.(Ho) t-value(Ho)  p-value
Gamma        0.7757      0.0123      0.0149        52.1897      0.0000
Tau B        0.6429      0.0122      0.0123        52.1897      0.0000
Tau C        0.6293      0.0121      ?             52.1897      0.0000
D-Row        0.6418      0.0122      0.0123        52.1897      0.0000
D-Column     0.6439      0.0122      0.0123        52.1897      0.0000
Correlation  0.6926      0.0128      0.0172        40.2669      0.0000
Spearman     0.6939      0.0127      0.0127        54.6614      0.0000
GK tau rows  0.3420      0.0123      ?             ?            ?
GK tau cols. 0.3430      0.0122      ?             ?            ?
U - sym.     0.3171      0.0110      ?             ?            ?
U - rows     0.3178      0.0110      ?             ?            ?
U - cols.    0.3164      0.0110      ?             ?            ?
Lambda-sym.  0.5373      0.0124      ?             ?            ?
Lambda-row   0.5374      0.0126      ?             ?            ?
Lambda-col.  0.5372      0.0126      ?             ?            ?
l-star-rows  0.5506      0.0136      ?             ?            ?
l-star-col.  0.5636      0.0127      ?             ?            ?
Lin. trend   ?           ?           ?             ?            ?
Kruskal row  1,561.4859 3.0000      ?             ?            0.0000
Kruskal col. 1,563.0303 3.0000      ?             ?            0.0000
Kappa        0.5744      0.0111      0.0106        54.3583      0.0000
McNemar      4.7625      6.0000      ?             ?            0.5746
McNemar df=1 0.9487      1.0000      ?             0.3459       0.3301
```

## *class* **CategoricalGenLinModel**

Analyzes categorical data using logistic, probit, Poisson, and other linear models.

Reweighted least squares is used to compute (extended) maximum likelihood estimates in some generalized linear models involving categorized data. One of several models, including probit, logistic, Poisson, logarithmic, and negative binomial models, may be fit for input point or interval observations. (In the usual case, only point observations are observed.)

Let

$$\gamma_i = w_i + x_i^T \beta = w_i + \eta_i$$

be the linear response where $x_i$ is a design column vector obtained from a row of $x, \beta$ is the column vector of coefficients to be estimated, and $w_i$ is a fixed parameter that may be input in `x`. When some of the $\gamma_i$ are infinite at the supremum of the likelihood, then extended *maximum likelihood estimates* are computed. Extended maximum likelihood are computed as the finite (but nonunique) estimates $\hat{\beta}$ that optimize the likelihood containing only the observations with finite $\hat{\gamma}_i$. These estimates, when combined with the set of indices of the observations such that $\hat{\gamma}_i$ is infinite at the supremum of the likelihood, are called extended maximum estimates. When none of the optimal $\hat{\gamma}_i$ are infinite, extended maximum likelihood estimates are identical to maximum likelihood estimates. Extended maximum likelihood estimation is discussed in more detail by Clarkson and Jennrich (1991). In `CategoricalGenLinModel`, observations with potentially infinite

$$\hat{\eta}_i = x_i^T \hat{\beta}$$

are detected and removed from the likelihood if `infin` = 0. See below.

The models available in CategoricalGenLinModel are:

| **Model Name** | **Parameterization** | **Response PDF** |
|---|---|---|
| MODEL0 (Poisson) | $\lambda = N \times e^{w+\eta}$ | $f(y) = \lambda^y e^{-\lambda}/y!$ |
| MODEL1 (Negative Binomial) | $\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$ | $f(y) = \begin{pmatrix} S+y-1 \\ y-1 \end{pmatrix} \theta^S (1-\theta)^y$ |
| MODEL2 (Logarithmic) | $\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$ | $f(y) = (1-\theta)^y/(y \ln \theta)$ |
| MODEL3 (Logistic) | $\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$ | $f(y) = \begin{pmatrix} N \\ y \end{pmatrix} \theta^y (1-\theta)^{N-y}$ |
| MODEL4 (Probit) | $\theta = \Phi(w+\eta)$ | $f(y) = \begin{pmatrix} N \\ y \end{pmatrix} \theta^y (1-\theta)^{N-y}$ |
| MODEL5 (Log-log) | $\theta = 1 - e^{-e^{w+\eta}}$ | $f(y) = \begin{pmatrix} N \\ y \end{pmatrix} \theta^y (1-\theta)^{N-y}$ |

Here $\Phi$ denotes the cumulative normal distribution, $N$ and $S$ are known parameters specified for each observation via column `ipar` of `x`, and $w$ is an optional fixed parameter specified for each observation via column `ifix` of `x`. (By default $N$ is taken to be 1 for `model` = 0, 3, 4 and 5 and $S$ is taken to be 1 for `model` = 1. By default $w$ is taken to be 0.) Since the log-log model (`model` = 5) probabilities are not symmetric with respect to 0.5, quantitatively, as well as qualitatively, different models result when the definitions of "success" and "failure" are interchanged in this distribution. In this model and all other models involving $\theta$, $\theta$ is taken to be the probability of a "success."

Note that each row vector in the data matrix can represent a single observation; or, through the use of column `ifrq` of the matrix `x`, each vector can represent several observations. Also note that classification variables and their products are easily incorporated into the models via the usual regression-type specifications.

## Computational Details

For interval observations, the probability of the observation is computed by summing the probability distribution function over the range of values in the observation interval. For right-interval observations, $\Pr(Y \geq y)$ is computed as a sum based upon the equality $\Pr(Y \geq y) = 1 - \Pr(Y < y)$. Derivatives are computed similarly. `CategoricalGenLinModel` allows three types of interval observations. In full interval observations, both the lower and the upper endpoints of the interval must be specified. For right-interval observations, only the lower endpoint need be given while for left-interval observations, only the upper endpoint is given.

The computations proceed as follows:

1. The input parameters are checked for consistency and validity.
2. Estimates of the means of the "independent" or design variables are computed. The frequency of the observation in all but binomial distribution model is taken from column `ifrq` of the data matrix `x`. In binomial distribution models, the frequency is taken as the product of $n =$ `x[i][ipar]` and `x[i][ifrq]`. In all cases these values default to 1. Means are computed as

$$\bar{x} = \frac{\Sigma_i f_i x_i}{\Sigma_i f_i}$$

3. If `init` $= 0$, initial estimates of the coefficients are obtained (based upon the observation intervals) as multiple regression estimates relating transformed observation probabilities to the observation design vector. For example, in the binomial distribution models, $\theta$ for point observations may be estimated as

$$\hat{\theta} = x[i][irt]/x[i][ipar]$$

   and, when `model` $= 3$, the linear relationship is given by

$$\left(\ln(\hat{\theta}/(1-\hat{\theta})) \approx x\beta\right)$$

   while if `model` $= 4$,

$$\left(\Phi^{-1}(\hat{\theta}) = x\beta\right)$$

   For bounded interval observations, the midpoint of the interval is used for `x[i][irt]`. Right-interval observations are not used in obtaining initial estimates when the distribution has unbounded support (since the midpoint of the interval is not defined). When computing initial estimates, standard modifications are made to prevent illegal operations such as division by zero.

Regression estimates are obtained at this point, as well as later, by use of linear regression.

4. Newton-Raphson iteration for the maximum likelihood estimates is implemented via iteratively reweighted least squares. Let

$$\Psi(x_i^T\beta)$$

denote the log of the probability of the $i$-th observation for coefficients $\beta$. In the least-squares model, the weight of the $i$-th observation is taken as the absolute value of the second derivative of

$$\Psi(x_i^T\beta)$$

with respect to

$$\gamma_i = x_i^T\beta$$

(times the frequency of the observation), and the dependent variable is taken as the first derivative $\Psi$ with respect to $\gamma_i$, divided by the square root of the weight times the frequency. The Newton step is given by

$$\Delta\beta = \left(\sum_i |\Psi^{''}(\gamma_i)| x_i x_i^T\right)^{-1} \sum_i \Psi^{'}(\gamma_i) x_i$$

where all derivatives are evaluated at the current estimate of $\gamma$, and $\beta_{n+1} = \beta_n - \Delta\beta$. This step is computed as the estimated regression coefficients in the least-squares model. Step halving is used when necessary to ensure a decrease in the criterion.

5. Convergence is assumed when the maximum relative change in any coefficient update from one iteration to the next is less than `eps` or when the relative change in the log-likelihood from one iteration to the next is less than `eps`/100. Convergence is also assumed after `maxIterations` or when step halving leads to a step size of less than .0001 with no increase in the log-likelihood.

6. For interval observations, the contribution to the log-likelihood is the log of the sum of the probabilities of each possible outcome in the interval. Because the distributions are discrete, the sum may involve many terms. The user should be aware that data with wide intervals can lead to expensive (in terms of computer time) computations.

7. If `setInfiniteEstimateMethod` set to 0, then the methods of Clarkson and Jennrich (1991) are used to check for the existence of infinite estimates in

$$\eta_i = x_i^T\beta$$

As an example of a situation in which infinite estimates can occur, suppose that observation $j$ is right censored with $t_j > 15$ in a logistic model. If design matrix `x` is is such that $x_{jm} = 1$ and $x_{im} = 0$ for all $i \neq j$, then the optimal estimate of $\beta_m$ occurs at

$$\hat{\beta_m} = \infty$$

leading to an infinite estimate of both $\beta_m$ and $\eta_j$. In `CategoricalGenLinModel`, such estimates may be "computed."

In all models fit by `CategoricalGenLinModel` , infinite estimates can only occur when the optimal estimated probability associated with the left- or right-censored observation is 1. If `setInfiniteEstimateMethod` set to 0, left- or right- censored observations that have estimated probability greater than 0.995 at some point during the iterations are excluded from the log-likelihood, and the iterations proceed with a log-likelihood based upon the remaining observations. This allows convergence of the algorithm when the maximum relative change in the estimated coefficients is small and also allows for the determination of observations with infinite

$$\eta_i = x_i^T \beta$$

At convergence, linear programming is used to ensure that the eliminated observations have infinite $\eta_i$. If some (or all) of the removed observations should not have been removed (because their estimated $\eta_{i's}$ must be finite), then the iterations are restarted with a log-likelihood based upon the finite $\eta_i$ observations. See Clarkson and Jennrich (1991) for more details.

When `setInfiniteEstimateMethod` is set to 1, no observations are eliminated during the iterations. In this case, when infinite estimates occur, some (or all) of the coefficient estimates $\hat{\beta}$ will become large, and it is likely that the Hessian will become (numerically) singular prior to convergence.

When infinite estimates for the $\hat{\eta_i}$ are detected, linear regression (see Chapter 2, Regression;) is used at the convergence of the algorithm to obtain unique estimates $\hat{\beta}$. This is accomplished by regressing the optimal $\hat{\eta_i}$ or the observations with finite $\eta$ against $x\beta$, yielding a unique $\hat{\beta}$ (by setting coefficients $\hat{\beta}$ that are linearly related to previous coefficients in the model to zero). All of the final statistics relating to $\hat{\beta}$ are based upon these estimates.

8. Residuals are computed according to methods discussed by Pregibon (1981). Let $\ell_i(\gamma_i)$ denote the log-likelihood of the $i$-th observation evaluated at $\gamma_i$. Then, the standardized residual is computed as

$$r_i = \frac{\ell_i'(\hat{\gamma_i})}{\sqrt{\ell_i''(\hat{\gamma_i})}}$$

where $\hat{\gamma_i}$ is the value of $\gamma_i$ when evaluated at the optimal $\hat{\beta}$ and the derivatives here (and only here) are with respect to $\gamma$ rather than with respect to $\beta$. The denominator of this expression is used as the "standard error of the residual" while the numerator is the "raw" residual.

Following Cook and Weisberg (1982), we take the influence of the $i$-th observation to be

$$\ell_i'(\hat{\gamma}_i)^T \ell''(\hat{\gamma})^{-1} \ell'(\hat{\gamma}_i)$$

This quantity is a one-step approximation to the change in the estimates when the $i$-th observation is deleted. Here, the partial derivatives are with respect to $\beta$.

## Programming Notes

1. Classification variables are specified via `setClassificationVariableColumn` . Indicator or dummy variables are created for the classification variables.
2. To enhance precision "centering" of covariates is performed if `setModelIntercept` is set to 1 and (number of observations) - (number of rows in `x` missing one or more values) >1. In doing so, the sample means of the design variables are subtracted from each observation prior to its inclusion in the model. On convergence the intercept, its variance and its covariance with the remaining estimates are transformed to the uncentered estimate values.
3. Two methods for specifying a binomial distribution model are possible. In the first method, `x[i][ifrq]` contains the frequency of the observation while `x[i][irt]` is 0 or 1 depending upon whether the observation is a success or failure. In this case, $N$ = `x[i][ipar]` is always 1. The model is treated as repeated Bernoulli trials, and interval observations are not possible.

A second method for specifying binomial models is to use `x[i][irt]` to represent the number of successes in the `x[i][ipar]` trials. In this case, `x[i][ifrq]` will usually be 1, but it may be greater than 1, in which case interval observations are possible.

Note that the `solve` method must be called prior to calling the "get" member functions, otherwise a `null` is returned.

## Declaration

```
public class com.imsl.stat.CategoricalGenLinModel
extends java.lang.Object
```

## Inner Classes

### *class* CategoricalGenLinModel.ClassificationVariableException

The ClassificationVariable vector has not been initialized.

### Declaration

public static class com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *CategoricalGenLinModel.ClassificationVariableException*
  `public` **CategoricalGenLinModel.ClassificationVariableException**`( )`
  - **Description**
    Constructs a `ClassificationVariableException`.

## *class* CategoricalGenLinModel.ClassificationVariableLimitException

The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

### Declaration

public static class com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException
**extends** com.imsl.IMSLException (page 1240)

### Constructor

---

- *CategoricalGenLinModel.ClassificationVariableLimitException*
  `public` **CategoricalGenLinModel.ClassificationVariableLimitException**`(`
  `int` **maxcl** `)`
  - **Description**
    Constructs a `ClassificationVariableLimitException`.
  - **Parameters**
    * `maxcl` – An `int` which specifies the upper bound.

## *class* CategoricalGenLinModel.ClassificationVariableValueException

The number of distinct values for each Classification Variable must be greater than 1.

#### Declaration

public static class com.imsl.stat.CategoricalGenLinModel.ClassificationVariableValueException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *CategoricalGenLinModel.ClassificationVariableValueException*
  `public` **CategoricalGenLinModel.ClassificationVariableValueException**`(` `int` **index**`,` `int` **value** `)`
  - **Description**
    Constructs a `ClassificationVariableValueException`.
  - **Parameters**
    * `index` – An `int` which specifies the index of a classification variable.
    * `value` – An `int` which specifies the number of distinct values that can be taken by this classification variable.

### *class* CategoricalGenLinModel.DeleteObservationsException

The number of observations to be deleted (set by `setObservationMax`) has grown too large.

#### Declaration

public static class com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *CategoricalGenLinModel.DeleteObservationsException*
  `public` **CategoricalGenLinModel.DeleteObservationsException**`(` `int` **nmax** `)`
  - **Description**
    Constructs a `DeleteObservationsException`.
  - **Parameters**
    * `nmax` – An `int` which specifies the maximum number of observations that can be handled in the linear programming as set by `setObservationMax`.

## Fields

- public static final `int` **MODEL0**
  - Indicates an exponential function is used to model the distribution parameter. The distribution of the response variable is Poisson. The lower bound of the response variable is 0.
- public static final `int` **MODEL1**
  - Indicates a logistic function is used to model the distribution parameter. The distribution of the response variable is negative Binomial. The lower bound of the response variable is 0.
- public static final `int` **MODEL2**
  - Indicates a logistic function is used to model the distribution parameter. The distribution of the response variable is Logarithmic. The lower bound of the response variable is 1.
- public static final `int` **MODEL3**
  - Indicates a logistic function is used to model the distribution parameter. The distribution of the response variable is Binomial. The lower bound of the response variable is 0.
- public static final `int` **MODEL4**
  - Indicates a probit function is used to model the distribution parameter. The distribution of the response variable is Binomial. The lower bound of the response variable is 0.
- public static final `int` **MODEL5**
  - Indicates a log-log function is used to model the distribution parameter. The distribution of the response variable is Binomial. The lower bound of the response variable is 0.

## Constructor

- *CategoricalGenLinModel*
  `public` **CategoricalGenLinModel**`( double[][] x, int` **model** `)`
  - **Description**
    Constructs a new `CategoricalGenLinModel`.

– **Parameters**

 * `x` – A `double` input matrix containing the data where the number of rows in the matrix is equal to the number of observations.
 * `model` – An `int` scalar which specifies the distribution of the response variable and the function used to model the distribution parameter. Use one of the class members from the following table. The lower bound given in the table is the minimum possible value of the response variable:

| **Model** | **Distribution** | **Function** | **Lower-bound** |
|---|---|---|---|
| 0 | Poisson | Exponential | 0 |
| 1 | Negative Binomial | Logistic | 0 |
| 2 | Logarithmic | Logistic | 1 |
| 3 | Binomial | Logistic | 0 |
| 4 | Binomial | Probit | 0 |
| 5 | Binomial | Log-log | 0 |

 Let $\gamma$ be the dot product of a row in the design matrix with the parameters (plus the fixed parameter, if used). Then, the functions used to model the distribution parameter are given by:

| **Name** | **Function** |
|---|---|
| Exponential | $e^{\gamma}$ |
| Logistic | $e^{\gamma}/(1+e^{\gamma})$ |
| Probit | $\Phi(\gamma)$ (where $\Phi$ is the normal cdf) |
| Log-log | $1-e^{-\gamma}$ |

## Methods

- *getCaseAnalysis*
 `public double[][]` **getCaseAnalysis**`( )`

 – **Description**
 Returns the case analysis.

 – **Returns** – A `double` matrix containing the case analysis or `null` if `solve` has not been called. The matrix is $nobs \times 5$ where $nobs$ is the number of observations. The matrix contains:

| Column | Statistic |
|---|---|
| 0 | Prediction. |
| 1 | The residual. |
| 2 | The estimated standard error of the residual. |
| 3 | The estimated influence of the observation. |
| 4 | The standardized residual. |

Case studies are computed for all observations except where missing values prevent their computation. The prediction in column 0 depends upon the model used as follows:

| Model | Prediction |
|---|---|
| 0 | The predicted mean for the observation. |
| 1-4 | The probability of a success on a single trial. |

---

- *getClassificationVariableCounts*
  `public int[]` **getClassificationVariableCounts**`( ) throws com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException`
  - **Description**
    Returns the number of values taken by each classification variable.
  - **Returns** – An `int` array of length *nclvar* containing the number of values taken by each classification variable where *nclvar* is the number of classification variables or `null` if `solve` has not been called.
  - **Throws**
    - `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException` – is thrown when the number of values taken by each classification variable has been set by the user to be less than or equal to 1

---

- *getClassificationVariableValues*
  `public double[]` **getClassificationVariableValues**`( ) throws com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException`
  - **Description**
    Returns the distinct values of the classification variables in ascending order.
  - **Returns** – A `double` array of length $\sum_{k=0}^{\mathrm{nclvar}} \mathrm{nclval[k]}$ containing the distinct values of the classification variables in ascending order where *nclvar* is the number of classification variables and *nclval[i]* is the number of values taken by the *i*-th classification variable. A `null` is returned if `solve` has not been called prior to calling this method.

– **Throws**

  *
  `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException`
  – is thrown when the number of values taken by each classification variable has been set by the user to be less than or equal to 1

---

- *getCovarianceMatrix*
  `public double[][]` **getCovarianceMatrix**`( )`

  – **Description**
  Returns the estimated asymptotic covariance matrix of the coefficients.

  – **Returns** – A `double` matrix containing the estimated asymptotic covariance matrix of the coefficients or `null` if `solve` has not been called. The covariance matrix is *nCoef* by *nCoef* where *nCoef* is the number of coefficients in the model.

---

- *getDesignVariableMeans*
  `public double[]` **getDesignVariableMeans**`( )`

  – **Description**
  Returns the means of the design variables.

  – **Returns** – A `double` array of length *nCoef* containing the means of the design variables where *nCoef* is the number of coefficients in the model or `null` if `solve` has not been called.

---

- *getExtendedLikelihoodObservations*
  `public int[]` **getExtendedLikelihoodObservations**`( )`

  – **Description**
  Returns a vector indicating which observations are included in the extended likelihood.

  – **Returns** – An `int` array of length *nobs* indicating which observations are included in the extended likelihood where *nobs* is the number of observations. The values within the array are interpreted as:

| **Value** | **Status of observation** |
|---|---|
| 0 | Observation $i$ is in the likelihood. |
| 1 | Observation $i$ cannot be in the likelihood because it contains at least one missing value in `x`. |
| 2 | Observation $i$ is not in the likelihood. Its estimated parameter is infinite. |

  A `null` is returned if `solve` has not been called prior to calling this method.

---

- *getHessian*
  `public double[][]` **getHessian**`( ) throws com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException, com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException, com.imsl.stat.CategoricalGenLinModel.ClassificationVariableValueException, com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException`

  – **Description**
    Returns the Hessian computed at the initial parameter estimates.
  – **Returns** – A `double` matrix containing the Hessian computed at the input parameter estimates. The Hessian matrix is *nCoef* by *nCoef* where *nCoef* is the number of coefficients in the model. This member function will call `solve` to get the Hessian if the Hessian has not already been computed.
  – **Throws**
    * `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException` – is thrown when the number of values taken by each classification variable has been set by the user to be less than or equal to 1
    * `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException` – is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, `maxcl`
    * `com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException` – is thrown if the number of observations to be deleted has grown too large

- *getLastParameterUpdates*
  `public double[]` **getLastParameterUpdates**`( )`

  – **Description**
    Returns the last parameter updates (excluding step halvings).
  – **Returns** – A `double` array of length *nCoef* containing the last parameter updates (excluding step halvings) or `null` if `solve` has not been called.

- *getNRowsMissing*
  `public int` **getNRowsMissing**`( )`

  – **Description**
    Returns the number of rows of data in `x` that contain missing values in one or more specific columns of `x`.
  – **Returns** – An `int` scalar representing the number of rows of data in `x` that contain missing values in one or more specific columns of `x` or `null` if `solve` has not been called. The columns of `x` included in the count are the columns

containing the upper or lower endpoints of full interval, left interval, or right interval observations. Also included are the columns containing the frequency responses, fixed parameters, optional distribution parameters, and interval type for each observation. Columns containing classification variables and columns associated with each effect in the model are also included.

---

- *getOptimizedCriterion*
`public double` **getOptimizedCriterion**`( )`
  - **Description**
    Returns the optimized criterion.
  - **Returns** – A `double` scalar representing the optimized criterion or `null` if `solve` has not been called. The criterion to be maximized is a constant plus the log-likelihood.

---

- *getParameters*
`public double[][]` **getParameters**`( )`
  - **Description**
    Returns the parameter estimates and associated statistics.
  - **Returns** – An `nCoef` row by 4 column `double` matrix containing the parameter estimates and associated statistics or `null` if `solve` has not been called. Here, `nCoef` is the number of coefficients in the model. The statistics returned are as follows:

| **Column** | **Statistic** |
|---|---|
| 0 | Coefficient estimate. |
| 1 | Estimated standard deviation of the estimated coefficient. |
| 2 | Asymptotic normal score for testing that the coefficient is zero. |
| 3 | $\rho$ - value associated with the normal score in column 2. |

---

- *getProduct*
`public double[]` **getProduct**`( ) throws`
`com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException,`
`com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException,`
`com.imsl.stat.CategoricalGenLinModel.ClassificationVariableValueException,`
`com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException`
  - **Description**
    Returns the inverse of the Hessian times the gradient vector computed at the input parameter estimates.

- **Returns** – A `double` array of length *nCoef* containing the inverse of the Hessian times the gradient vector computed at the input parameter estimates. *nCoef* is the number of coefficients in the model. This member function will call `solve` to get the product if the product has not already been computed.
- **Throws**
  - `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException` – is thrown when the number of values taken by each classification variable has been set by the user to be less than or equal to 1
  - `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException` – is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, `maxcl`
  - `com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException` – is thrown if the number of observations to be deleted has grown too large

---

- *setCensorColumn*

`public void` **`setCensorColumn`**`( int` **`icen`** `)`

- **Description**
  Sets the column number in `x` which contains the interval type for each observation.
- **Parameters**
  - `icen` – An `int` scalar which indicates the column number `x` which contains the interval type code for each observation. The valid codes are interpreted as:

| `x[i][icen]` | **Censoring** |
|---|---|
| 0 | Point observation. The response is unique and is given by `x[i][irt]`. |
| 1 | Right interval. The response is greater than or equal to `x[i][irt]` and less than or equal to the upper bound, if any, of the distribution. |
| 2 | Left interval. The response is less than or equal to `x[i][ilt]` and greater than or equal to the lower bound of the distribution. |
| 3 | Full interval. The response is greater than or equal to `x[i][irt]` but less than or equal to `x[i][ilt]`. |

  If this member function is not called a censoring code of 0 is assumed.
- **Throws**

* `java.lang.IllegalArgumentException` – is thrown when `icen` is less than 0 or greater than or equal to the number of columns of `x`

---

- *setClassificationVariableColumn*
  `public void` **setClassificationVariableColumn**`( int[]` **indcl** `)`

  – **Description**
  Initializes an index vector to contain the column numbers in `x` that are classification variables.

  – **Parameters**
  * `indcl` – An `int` vector which contains the column numbers in `x` that are classification variables. By default this vector is not referenced.

  – **Throws**
  * `java.lang.IllegalArgumentException` – is thrown when an element of `indcl` is less than 0 or greater than or equal to the number of columns of `x`

---

- *setConvergenceTolerance*
  `public void` **setConvergenceTolerance**`( double` **eps** `)`

  – **Description**
  Set the convergence criterion.

  – **Parameters**
  * `eps` – A `double` scalar specifying the convergence criterion. Convergence is assumed when the maximum relative change in any coefficient estimate is less than `eps` from one iteration to the next or when the relative change in the log-likelihood, `getOptimizedCriterion`, from one iteration to the next is less than `eps`/100. `eps` must be greater than 0. If this member function is not called, `eps` = .001 is assumed.

  – **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if `eps` is or equal to 0

---

- *setEffects*
  `public void` **setEffects**`( int[]` **indef**`, int[]` **nvef** `)`

  – **Description**
  Initializes an index vector to contain the column numbers in `x` associated with each effect.

  – **Parameters**
  * `indef` – An `int` vector of length $\sum_{k=0}^{\mathrm{nef}-1} \mathrm{nvef}[k]$ where `nef` is the number of effects in the model. `indef` contains the column numbers in `x` that are associated with each effect. Member function `setEffects(int [], nvef [])` sets the number of variables associated with each effect in the model. The first `nvef[0]` elements of `indef` give the column numbers of the variables in

the first effect. The next `nvef[0]` elements give the column numbers of the variables in the second effect, etc. By default this vector is not referenced.
    * `nvef` – An `int` vector of length *nef* where *nef* is the number of effects in the model. `nvef` contains the number of variables associated with each effect in the model. By default this vector is not referenced.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when an element of `indef` is less than 0 or greater than or equal to the number of columns of `x` or if an element of `nvef` is less than or equal to 0

---

- *setExtendedLikelihoodObservations*
  `public void` **setExtendedLikelihoodObservations**`( int[]` **iadds** `)`
  - **Description**
    Initializes a vector indicating which observations are to be included in the extended likelihood.
  - **Parameters**
    * `iadds` – An `int` array of length *nobs* indicating which observations are included in the extended likelihood where *nobs* is the number of observations. The values within the array are interpreted as:

| **Value** | **Status of observation** |
|---|---|
| 0 | Observation $i$ is in the likelihood. |
| 1 | Observation $i$ cannot be in the likelihood because it contains at least one missing value in `x` . |
| 2 | Observation $i$ is not in the likelihood. Its estimated parameter is infinite. |

If this member function is not called, `iadds` is set to all zeroes.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown when an element of `iadds` is not in the range [0,2]

---

- *setFixedParameterColumn*
  `public void` **setFixedParameterColumn**`( int` **ifix** `)`
  - **Description**
    Sets the column number in `x` that contains a fixed parameter for each observation that is added to the linear response prior to computing the model parameter.
  - **Parameters**
    * `ifix` – An `int` scalar which indicates the column number in `x` that contains a fixed parameter for each observation that is added to the linear response

prior to computing the model parameter. The "fixed" parameter allows one to test hypothesis about the parameters via the log-likelihoods. By default the fixed parameter is assumed to be zero.

– **Throws**

  * `java.lang.IllegalArgumentException` – is thrown when `ifix` is less than 0 or greater than or equal to the number of columns of `x`

---

- *setFrequencyColumn*

  `public void` **setFrequencyColumn**`( int` **ifrq** `)`

  – **Description**

    Sets the column number in `x` that contains the frequency of response for each observation.

  – **Parameters**

    * `ifrq` – An `int` scalar which indicates the column number in `x` that contains the frequency of response for each observation. By default a frequency of 1 for each observation is assumed.

  – **Throws**

    * `java.lang.IllegalArgumentException` – is thrown when `ifrq` is less than 0 or greater than or equal to the number of columns of `x`

---

- *setInfiniteEstimateMethod*

  `public void` **setInfiniteEstimateMethod**`( int` **infin** `)`

  – **Description**

    Sets the method to be used for handling infinite estimates.

  – **Parameters**

    * `infin` – An `int` scalar which indicates the method to be used for handling infinite estimates. The method value is interpreted as follows:

| infin | **Method** |
|---|---|
| 0 | Remove a right or left-censored observation from the log-likelihood whenever the probability of the observation exceeds 0.995. At convergence, use linear programming to check that all removed observations actually have an estimated linear response that is infinite. Set `iadds[i]` for observation $i$ to 2 if the linear response is infinite. If not all removed observations have infinite linear response, recompute the estimates based upon the observations with estimated linear response that is finite. This option is valid only for censoring codes 1 and 2. |
| 1 | Iterate without checking for infinite estimates. |

By default `infin` = 1.

- **Throws**
  * `java.lang.IllegalArgumentException` – is thrown when `infin` is less than 0 or greater than 1

---

- *setInitialEstimates*

`public void setInitialEstimates( int init, double[] estimates )`

- **Description**

  Sets the initial parameter estimates option.
- **Parameters**
  * `init` – An input `int` indicating the desired initialization method for the initial estimates of the parameters. If this method is not called, `init` is set to 0.

| init | **Action** |
|---|---|
| 0 | Unweighted linear regression is used to obtain initial estimates. |
| 1 | The *nCoef*, number of coefficients, elements of `estimates` contain initial estimates of the parameters. Use of this option requires that the user know *nCoef* beforehand. |

  * `estimates` – An input `double` array of length *nCoef* containing the initial estimates of the parameters where *nCoef* is the number of estimated

coefficients in the model. (Used if `init` = 1.) If this member function is not called, unweighted linear regression is used to obtain the initial estimates.

- **Throws**
  - `java.lang.IllegalArgumentException` – is thrown when `init` is not in the range [0,1]

---

- *setLowerEndpointColumn*
  `public void` **setLowerEndpointColumn**`( int` **irt** `)`
  - **Description**
    Sets the column number in `x` that contains the lower endpoint of the observation interval for full interval and right interval observations.
  - **Parameters**
    - `irt` – An `int` scalar which indicates the column number in `x` that contains the lower endpoint of the observation interval for full interval and right interval observations. By default all observations are treated as "point" observations and `x[i][irt]` contains the observation point. If this member function is not called, the last column of `x` is assumed to contain the "point" observations.
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown when `irt` is less than 0 or greater than or equal to the number of columns of `x`

---

- *setMaxIterations*
  `public void` **setMaxIterations**`( int` **maxIterations** `)`
  - **Description**
    Set the maximum number of iterations allowed.
  - **Parameters**
    - `maxIterations` – An `int` specifying the maximum number of iterations allowed. `maxIterations` must be greater than 0. If this member function is not called, the maximum number of iterations is set to 30.
  - **Throws**
    - `java.lang.IllegalArgumentException` – is thrown if `maxIterations` is less than or equal to 0

---

- *setModelIntercept*
  `public void` **setModelIntercept**`( int` **intcep** `)`
  - **Description**
    Sets the intercept option.
  - **Parameters**

* `intcep` – An `int` scalar which indicates whether or not the model has an intercept. Input `intcep` is interpreted as follows:

| Value | Action |
|---|---|
| 0 | No intercept is in the model (unless otherwise provided for by the user). |
| 1 | Intercept is automatically included in the model. |

By default `intcep` = 1.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown when `intcep` is less than 0 or greater than 1

---

• *setObservationMax*
`public void` **setObservationMax**`( int` **nmax** `)`

– **Description**
Sets the maximum number of observations that can be handled in the linear programming.

– **Parameters**

* `nmax` – An `int` scalar which sets the maximum number of observations that can be handled in the linear programming. An illegal argument exception is thrown if `nmax` is less than 0. If this member function is not called, `nmax` is set to the number of observations.

– **Throws**

* `java.lang.IllegalArgumentException` – is thrown when `nmax` is less than 0

---

• *setOptionalDistributionParameterColumn*
`public void` **setOptionalDistributionParameterColumn**`( int` **ipar** `)`

– **Description**
Sets the column number in `x` that contains an optional distribution parameter for each observation.

– **Parameters**

* `ipar` – An `int` scalar which indicates the column number in `x` that contains an optional distribution parameter for each observation. The distribution parameter values are interpreted as follows depending on the model chosen:

| Model | Meaning of `x[i][ipar]` |
|---|---|
| 0 | The Poisson parameter is given by $x[i][ipar] \times e^{\rho}$. |
| 1 | The number of successes required in the negative binomial is given by `x[i][ipar]`. |
| 2 | `x[i][ipar] is not used.` |
| 3-5 | The number of trials in the binomial distribution is given by `x[i][ipar]`. |

By default the distribution parameter is assumed to be 1.

- **Throws**
  - `java.lang.IllegalArgumentException` – is thrown when `ipar` is less than 0 or greater than or equal to the number of columns of `x`

---

- *setUpperBound*

`public void` **setUpperBound**`( int` **maxcl** `)`

- **Description**
  Sets the upper bound on the sum of the number of distinct values taken on by each classification variable.
- **Parameters**
  - `maxcl` – An `int` scalar specifying the upper bound on the sum of the number of distinct values taken on by each classification variable. If this member function is not called, an upper bound of 1 is used.
- **Throws**
  - `java.lang.IllegalArgumentException` – is thrown when `maxcl` is less than 1 and the number of classification variables is greater than 0

---

- *setUpperEndpointColumn*

`public void` **setUpperEndpointColumn**`( int` **ilt** `)`

- **Description**
  Sets the column number in `x` that contains the upper endpoint of the observation interval for full interval and left interval observations.
- **Parameters**
  - `ilt` – An `int` scalar which indicates the column number in `x` that contains the upper endpoint of the observation interval for full interval and left interval observations. By default all observations are treated as "point" observations.
- **Throws**
  - `java.lang.IllegalArgumentException` – is thrown when `ilt` is less than 0 or greater than or equal to the number of columns of `x`

---

- *solve*
`public double[][]` **`solve`**`( ) throws`
`com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException,`
`com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException,`
`com.imsl.stat.CategoricalGenLinModel.ClassificationVariableValueException,`
`com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException`

  - **Description**
    Returns the parameter estimates and associated statistics for a CategoricalGenLinModel object.
  - **Returns** – An `nCoef` row by 4 column `double` matrix containing the parameter estimates and associated statistics. Here, `nCoef` is the number of coefficients in the model. The statistics returned are as follows:

| **Column** | **Statistic** |
|---|---|
| 0 | Coefficient estimate. |
| 1 | Estimated standard deviation of the estimated coefficient. |
| 2 | Asymptotic normal score for testing that the coefficient is zero. |
| 3 | $\rho$ - value associated with the normal score in column 2. |

  - **Throws**
    * `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException` – is thrown when the number of values taken by each classification variable has been set by the user to be less than or equal to 1
    * `com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException` – is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, `maxcl`
    * `com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException` – is thrown if the number of observations to be deleted has grown too large

### Example: Mortality of beetles.

The first example is from Prentice (1976) and involves the mortality of beetles after exposure to various concentrations of carbon disul-
phide. Both a logit and a probit fit are produced for linear model $\mu + \beta x$. The data is given as

| Covariate(x) | N | y |
|---|---|---|
| 1.755 | 62 | 18 |
| 1.784 | 56 | 28 |
| 1.811 | 63 | 52 |
| 1.836 | 59 | 53 |
| 1.861 | 62 | 61 |
| 1.883 | 60 | 60 |

```
import java.io.*;
import com.imsl.stat.*;
import com.imsl.math.*;

public class CategoricalGenLinModelEx1 {
    public static void main(String argv[]) throws Exception {

        // Set up a PrintMatrix object for later use.
        PrintMatrixFormat mf;
        PrintMatrix p;
        p = new PrintMatrix();
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();

        double[][] x  = {
            {1.69, 59.0, 6.0},
            {1.724, 60.0, 13.0},
            {1.755, 62.0, 18.0},
            {1.784, 56.0, 28.0},
            {1.811, 63.0, 52.0},
            {1.836, 59.0, 53.0},
            {1.861, 62.0, 61.0},
            {1.883, 60.0, 60.0},
       };
        CategoricalGenLinModel CATGLM3, CATGLM4;
        // MODEL3
        CATGLM3 = new CategoricalGenLinModel(x,
                  CategoricalGenLinModel.MODEL3);
        CATGLM3.setLowerEndpointColumn(2);
        CATGLM3.setOptionalDistributionParameterColumn(1);
        CATGLM3.setInfiniteEstimateMethod(1);
        CATGLM3.setModelIntercept(1);
        int[] nvef = {1};
```

```
        int[] indef  = {0};
        CATGLM3.setEffects(indef, nvef);
        CATGLM3.setUpperBound(1);

        System.out.println("MODEL3");
        p.setTitle("Coefficient Statistics");
        p.print(CATGLM3.solve());
        System.out.println("Log likelihood " +
        CATGLM3.getOptimizedCriterion());
        p.setTitle("Asymptotic Coefficient Covariance");
        p.setMatrixType(1);
        p.print(CATGLM3.getCovarianceMatrix());
        p.setMatrixType(0);
        p.setTitle("Case Analysis");
        p.print(CATGLM3.getCaseAnalysis());
        p.setTitle("Last Coefficient Update");
        p.print(CATGLM3.getLastParameterUpdates());
        p.setTitle("Covariate Means");
        p.print(CATGLM3.getDesignVariableMeans());
        p.setTitle("Observation Codes");
        p.print(CATGLM3.getExtendedLikelihoodObservations());
        System.out.println("Number of Missing Values " +
           CATGLM3.getNRowsMissing());

        // MODEL4
        CATGLM4 = new CategoricalGenLinModel(x,
           CategoricalGenLinModel.MODEL4);
        CATGLM4.setLowerEndpointColumn(2);
        CATGLM4.setOptionalDistributionParameterColumn(1);
        CATGLM4.setInfiniteEstimateMethod(1);
        CATGLM4.setModelIntercept(1);
        CATGLM4.setEffects(indef, nvef);
        CATGLM4.setUpperBound(1);
        CATGLM4.solve();

        System.out.println("MODEL4");
        System.out.println("Log likelihood " +
           CATGLM4.getOptimizedCriterion());
        p.setTitle("Coefficient Statistics");
        p.print(CATGLM4.getParameters());
    }
}
```

### Output

```
MODEL3
    Coefficient Statistics
      0        1        2     3
0  -60.757  5.188  -11.712  0
1   34.299  2.916   11.761  0

Log likelihood -18.77817904233396
Asymptotic Coefficient Covariance
      0        1
0   26.912  -15.124
1             8.505

             Case Analysis
     0      1       2      3      4
0  0.058   2.593  1.792  0.267   1.448
1  0.164   3.139  2.871  0.347   1.093
2  0.363  -4.498  3.786  0.311  -1.188
3  0.606  -5.952  3.656  0.232  -1.628
4  0.795   1.89   3.202  0.269   0.59
5  0.902  -0.195  2.288  0.238  -0.085
6  0.956   1.743  1.619  0.198   1.077
7  0.979   1.278  1.119  0.138   1.143

Last Coefficient Update
   0
0  0
1  0

Covariate Means
     0
0  1.793
1  0

Observation Codes
   0
0  0
1  0
2  0
3  0
```

```
4  0
5  0
6  0
7  0

Number of Missing Values 0
MODEL4
Log likelihood -18.232354574384562
    Coefficient Statistics
      0       1       2     3
0  -34.944  2.641  -13.231  0
1   19.737  1.485   13.289  0
```

## Example: Poisson Model.

In this example, the following data illustrate the Poisson model when all types of interval data are present. The example also illustrates the use of classification variables and the detection of potentially infinite estimates (which turn out here to be finite). These potential estimates lead to the two iteration summaries. The input data is

| **ilt** | **irt** | **icen** | **Class 1** | **Class 2** |
|---|---|---|---|---|
| 0 | 5 | 0 | 1 | 0 |
| 9 | 4 | 3 | 0 | 0 |
| 0 | 4 | 1 | 0 | 0 |
| 9 | 0 | 2 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 |

A linear model $\mu + \beta_1 x_1 + \beta_2 x_2$ is fit where $x_1 = 1$ if the Class 1 variable is 0, $x_1 = 1$, otherwise, and the $x_2$ variable is similarly defined

```
import java.io.*;
import com.imsl.stat.*;
import com.imsl.math.*;

public class CategoricalGenLinModelEx2 {
    public static void main(String argv[]) throws Exception {

        // Set up a PrintMatrix object for later use.
        PrintMatrixFormat mf;
        PrintMatrix p;
        p = new PrintMatrix();
        mf = new PrintMatrixFormat();
```

```
mf.setNoRowLabels();
mf.setNoColumnLabels();

double[][] x  = {
    {0.0, 5.0, 0.0, 1.0, 0.0},
    {9.0, 4.0, 3.0, 0.0, 0.0},
    {0.0, 4.0, 1.0, 0.0, 0.0},
    {9.0, 0.0, 2.0, 1.0, 1.0},
    {0.0, 1.0, 0.0, 0.0, 1.0},
};
CategoricalGenLinModel CATGLM;
CATGLM = new CategoricalGenLinModel(x,
         CategoricalGenLinModel.MODEL0);
CATGLM.setUpperEndpointColumn(0);
CATGLM.setLowerEndpointColumn(1);
CATGLM.setOptionalDistributionParameterColumn(1);
CATGLM.setCensorColumn(2);
CATGLM.setInfiniteEstimateMethod(0);
CATGLM.setModelIntercept(1);
int[] indcl = {3, 4};
CATGLM.setClassificationVariableColumn(indcl);
int[] nvef = {1, 1};
int[] indef  = {3, 4};
CATGLM.setEffects(indef, nvef);
CATGLM.setUpperBound(4);


p.setTitle("Coefficient Statistics");
p.print(CATGLM.solve());
System.out.println("Log likelihood " +
CATGLM.getOptimizedCriterion());
p.setTitle("Asymptotic Coefficient Covariance");
p.setMatrixType(1);
p.print(CATGLM.getCovarianceMatrix());
p.setMatrixType(0);
p.setTitle("Case Analysis");
p.print(CATGLM.getCaseAnalysis());
p.setTitle("Last Coefficient Update");
p.print(CATGLM.getLastParameterUpdates());
p.setTitle("Covariate Means");
p.print(CATGLM.getDesignVariableMeans());
p.setTitle("Distinct Values For Each Class Variable");
```

```
        p.print(CATGLM.getClassificationVariableValues());
        System.out.println("Number of Missing Values " +
           CATGLM.getNRowsMissing());
   }
}
```

### Output

```
     Coefficient Statistics
     0       1      2       3
0  -0.549  1.171  -0.469  0.64
1   0.549  0.61    0.9    0.368
2   0.549  1.083   0.507  0.612

Log likelihood -3.1146384925784414
Asymptotic Coefficient Covariance
     0       1       2
0   1.372  -0.372  -1.172
1           0.372   0.172
2                   1.172

             Case Analysis
     0      1       2      3      4
0  5      -0      2.236  1      -0
1  6.925  -0.412  2.108  0.764  -0.196
2  6.925   0.412  1.173  0.236   0.351
3  0       0      0      0       ?
4  1      -0      1      1      -0

Last Coefficient Update
   0
0  -0
1   0
2   0

Covariate Means
    0
0  0.6
1  0.6
2  0
```

```
Distinct Values For Each Class Variable
   0
0  0
1  1
2  0
3  1

Number of Missing Values 0
```

# Chapter 16

# Nonparametric Statistics

## Classes



### Usage Notes

Much of what is considered nonparametric statistics is included in other chapters. Topics of possible interest in other chapters are: nonparametric measures of location and scale (see "Basic Statistics"), nonparametric measures in a contingency table (see "Categorical and Discrete Data Analysis") and tests of goodness of fit and randomness (see "Tests of Goodness of Fit and Randomness").

### Missing Values

Most classes described in this chapter automatically handle missing values (NaN, "Not a Number"; see `Double.NaN`).

### Tied Observations

The `WilcoxonRankSum` class described in this chapter contains a set method, `setFuzz`. Observations that are within fuzz of each other in absolute value are said to be tied. If `fuzz` = 0.0, observations must be identically equal before they are considered to be tied. Other positive values of fuzz allow for numerical imprecision or roundoff error.

## *class* **SignTest**

Performs a sign test.

Class `SignTest` tests hypotheses about the proportion $p$ of a population that lies below a value $q$, where $p$ corresponds to `percentage` and $q$ corresponds to `percentile` in the `setPercentage` and `setPercentile` methods, respectively. In continuous distributions, this can be a test that $q$ is the 100 $p$-th percentile of the population from which `x` was obtained. To carry out testing, `SignTest` tallies the number of values above $q$ in the number of positive differences $x[j-1] -$ percentile for $j = 1, 2, \ldots,$ x.length. The binomial probability of the number of values above $q$ in the number of positive differences $x[j-1] -$ percentile for $j = 1, 2, \ldots, \ldots,$ x.length or more values above $q$ is then computed using the proportion $p$ and the sample size in `x` (adjusted for the missing observations and ties).

Hypothesis testing is performed as follows for the usual null and alternative hypotheses:

- $H_0 : Pr(x \leq q) \geq p$ (the $p$-th quantile is at least $q$)
  $H_1 : Pr(x \leq q) < p$
  Reject $H_0$ if *probability* is less than or equal to the significance level
- $H_0 : Pr(x \leq q) \leq p$ (the $p$-th quantile is at least $q$)
  $H_1 : Pr(x \leq q) > p$
  Reject $H_0$ if *probability* is greater than or equal to 1 minus the significance level
- $H_0 : Pr(x = q) = p$ (the $p$-th quantile is $q$)
  $H_1 : Pr((x \leq q) < p)$ or $Pr((x \leq q) > p)$
  Reject $H_0$ if *probability* is less than or equal to half the significance level or greater than or equal to 1 minus half the significance level

The assumptions are as follows:

1. They are independent and identically distributed.
2. Measurement scale is at least ordinal; i.e., an ordering less than, greater than, and equal to exists in the observations.

Many uses for the sign test are possible with various values of $p$ and $q$. For example, to perform a matched sample test that the difference of the medians of $y$ and $z$ is 0.0, let $p = 0.5$, $q = 0.0$, and $x_i = y_i - z_i$ in matched observations $y$ and $z$. To test that the median difference is $c$, let $q = c$.

## Declaration

public class com.imsl.stat.SignTest
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

---

- *SignTest*
  `public` **SignTest**`( double[] x )`
  - **Description**
    Constructor for `SignTest`.
  - **Parameters**
    * `x` – A `double` array containing the data.

## Methods

---

- *compute*
  `public final double` **compute**`( )`
  - **Description**
    Performs a sign test.
  - **Returns** – A `double` scalar containing the Binomial probability of `getNumPositiveDev` or more positive differences in `x.length` - number of zero differences trials. Call this value probability. If using default values, the null hypothesis is that the median equals 0.0.

---

- *getNumPositiveDev*
  `public int` **getNumPositiveDev**`( )`
  - **Description**
    Returns the number of positive differences.
  - **Returns** – An `int` scalar containing the number of positive differences `x[j-1]-percentile` for j = 1, 2, ..., `x.length`.

---

- *getNumZeroDev*
  `public int` **getNumZeroDev**`( )`

– **Description**
Returns the number of zero differences.

– **Returns** – An `int` scalar containing the number of zero differences (ties) `x[j-1]-percentile` for j = 1, 2, ..., `x.length`.

---

- *setPercentage*
`public void` **setPercentage**`( double` **percentage** `)`

  – **Description**
  Sets the percentage percentile of the population.

  – **Parameters**

    * `percentage` – A `double` scalar containing the value in the range (0, 1). `percentile` is the 100 * `percentage` percentile of the population. Default: `percentage` = 0.5.

---

- *setPercentile*
`public void` **setPercentile**`( double` **percentile** `)`

  – **Description**
  Sets the hypothesized percentile of the population.

  – **Parameters**

    * `percentile` – A `double` scalar containing the hypothesized percentile of the population from which `x` was drawn. Default: `percentile` = 0.0

### Example 1: Sign Test

This example tests the hypothesis that at least 50 percent of a population is negative. Because $0.18 < 0.95$, the null hypothesis at the 5-percent level of significance is not rejected.

```
import java.text.*;
import com.imsl.stat.*;

public class SignTestEx1 {
    public static void main(String args[]) {
        double[] x = {92.0, 139.0, -6.0, 10.0, 81.0, -11.0, 45.0, -25.0, -4.0,
        22.0, 2.0, 41.0, 13.0, 8.0, 33.0, 45.0, -33.0, -45.0, -12.0};
        SignTest st = new SignTest(x);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(6);

        System.out.println("Probability = " + nf.format(st.compute()));
```

```
    }
}
```

### Output

```
Probability = 0.179642
```

### Example 2: Sign Test

This example tests the null hypothesis that at least 75 percent of a population is negative. Because $0.923 < 0.95$, the null hypothesis at the 5-percent level of significance is rejected.

```
import java.text.*;
import com.imsl.stat.*;

public class SignTestEx2 {
    public static void main(String args[]) {
        double[] x = {92.0, 139.0, -6.0, 10.0, 81.0, -11.0, 45.0, -25.0, -4.0,
        22.0, 2.0, 41.0, 13.0, 8.0, 33.0, 45.0, -33.0, -45.0, -12.0};
        SignTest st = new SignTest(x);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(6);

        st.setPercentage(0.75);
        st.setPercentile(0.0);
        System.out.println("Probability = " + nf.format(st.compute()));
        System.out.println("Number of positive deviations = " +
        st.getNumPositiveDev());
        System.out.println("Number of ties = " + st.getNumZeroDev());
    }
}
```

### Output

```
Probability = 0.922543
Number of positive deviations = 12
Number of ties = 0
```

# *class* WilcoxonRankSum

Performs a Wilcoxon rank sum test.

Class `WilcoxonRankSum` performs the Wilcoxon rank sum test for identical population distribution functions. The Wilcoxon test is a linear transformation of the Mann-Whitney $U$ test. If the difference between the two populations can be attributed solely to a difference in location, then the Wilcoxon test becomes a test of equality of the population means (or medians) and is the nonparametric equivalent of the two-sample $t$-test. Class `WilcoxonRankSum` obtains ranks in the combined sample after first eliminating missing values from the data. The rank sum statistic is then computed as the sum of the ranks in the `x` sample. Three methods for handling ties are used. (A tie is counted when two observations are within `fuzz` of each other.) Method 1 uses the largest possible rank for tied observations in the smallest sample, while Method 2 uses the smallest possible rank for these observations. Thus, the range of possible rank sums is obtained.

Method 3 for handling tied observations between samples uses the average rank of the tied observations. Asymptotic standard normal scores are computed for the $W$ score (based on a variance that has been adjusted for ties) when average ranks are used (see Conover 1980, p. 217), and the probability associated with the two-sided alternative is computed.

**Hypothesis Tests**

In each of the following tests, the first line gives the hypothesis (and its alternative) under the assumptions 1 to 3 below, while the second line gives the hypothesis when assumption 4 is also true. The rejection region is the same for both hypotheses and is given in terms of Method 3 for handling ties. If another method for handling ties is desired, another output statistic, `stat[0]` or `stat[3]`,should be used, where `stat` is the array containing the statistics returned from the `getStatistics` method.

| Test | Null Hypothesis | Alternative Hypothesis | Action |
|---|---|---|---|
| 1 | $H_0 : \Pr(x1 < x2) = 0.5$ $H_0 : E(x1) = E(x2)$ | $H_1 : \Pr(x1 < x2) \neq 0.5$ $H_1 : E(x1) \neq E(x2)$ | Reject if `stat[9]` is less than the significance level of the test. Alternatively, reject the null hypothesis if `stat[6]` is too large or too small. |
| 2 | $H_0 : \Pr(x1 < x2) \leq 0.5$ $H_0 : E(x1) \geq E(x2)$ | $H_1 : \Pr(x1 < x2) \neq 0.5$ $H_1 : E(x1) < E(x2)$ | Reject if `stat[6]` is too small |
| 3 | $H_0 : \Pr(x1 < x2) \geq 0.5$ $H_0 : E(x1) \leq E(x2)$ | $H_1 : \Pr(x1 < x2) < 0.5$ $H_1 : E(x1) > E(x2)$ | Reject if `stat[6]` is too large |

**Assumptions**

1. $x$ and $y$ contain random samples from their respective populations.
2. All observations are mutually independent.
3. The measurement scale is at least ordinal (i.e., an ordering less than, greater than, or equal to exists among the observations).
4. If *f(x)* and *g(y)* are the distribution functions of $x$ and $y$, then *g(y) = f(x + c)* for some constant $c$(i.e., the distribution of $y$ is, at worst, a translation of the distribution of $x$).

The p-value is calculated using the large-sample normal approximation. This approximate calculation is only valid when the size of one or both samples is greater than 50. For smaller samples, see the exact tables for the Wilcoxon Rank Sum Test.

## Declaration

public class com.imsl.stat.WilcoxonRankSum
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *WilcoxonRankSum*
  `public` **WilcoxonRankSum**`( double[] x, double[] y )`

– **Description**
Constructor for `WilcoxonRankSum`.
– **Parameters**
  * `x` – A `double` array containing the first sample.
  * `y` – A `double` array containing the second sample.

## Methods

---

- *compute*
`public final double` **compute**`( )`
  – **Description**
  Performs a Wilcoxon rank sum test.
  – **Returns** – A `double` scalar containing the two-sided p-value for the Wilcoxon rank sum statistic that is computed with average ranks used in the case of ties.

---

- *getStatistics*
`public double[]` **getStatistics**`( )`
  – **Description**
  Returns the statistics.
  – **Returns** – A `double` array of length 10 containing the following statistics:

| Row | Statistics |
|---|---|
| 0 | Wilcoxon $W$ statistic (the sum of the ranks of the $x$ observations) adjusted for ties in such a manner that $W$ is as small as possible |
| 1 | 2 x E(W) - W, where E(W) is the expected value of $W$ |
| 2 | probability of obtaining a statistic less than or equal to min{W, 2 x E(W) - W} |
| 3 | $W$ statistic adjusted for ties in such a manner that $W$ is as large as possible |
| 4 | 2 x E(W) - W, where E(W) is the expected value of W, adjusted for ties in such a manner that $W$ is as large as possible |
| 5 | probability of obtaining a statistic less than or equal to min{W, 2 x E(W) - W}, adjusted for ties in such a manner that $W$ is as large as possible |
| 6 | $W$ statistic with average ranks used in case of ties |
| 7 | estimated standard error of Row 6 under the null hypothesis of no difference |
| 8 | standard normal score associated with Row 6 |
| 9 | two-sided p-value associated with Row 8 |

- *setFuzz*
`public void` **setFuzz**`( double` **fuzz** `)`
  - **Description**
    Sets the nonnegative constant used to determine ties in computing ranks in the combined samples.
  - **Parameters**
    * `fuzz` – A `double` scalar containing the nonnegative constant used to determine ties in computing ranks in the combined samples. A tie is declared when two observations in the combined sample are within `fuzz` of each other. Default:

$$\text{fuzz} = 100 \times 2.2204460492503131e - 16 \times \max(|x_{i1}|, |x_{j2}|)$$

### Example 1: Wilcoxon Rank Sum Test

The following example is taken from Conover (1980, p. 224). It involves the mixing time of two mixing machines using a total of 10 batches of a certain kind of batter, five batches for each machine. The null hypothesis is not rejected at the 5-percent level of significance.

```
import java.text.*;
import com.imsl.*;
import com.imsl.stat.*;

public class WilcoxonRankSumEx1 {
    public static void main(String args[]) {
        double[] x = {7.3, 6.9, 7.2, 7.8, 7.2};
        double[] y = {7.4, 6.8, 6.9, 6.7, 7.1};

        WilcoxonRankSum wilcoxon = new WilcoxonRankSum(x, y);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);

        // Trun off printing of warning messages.
        Warning.setOut(null);

        System.out.println("p-value = " + nf.format(wilcoxon.compute()));
    }
}
```

### Output

```
p-value = 0.1412
```

### Example 2: Wilcoxon Rank Sum Test

The following example uses the same data as in example 1. Now, all the statistics are displayed.

```
import java.text.*;
import com.imsl.*;
```

```
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class WilcoxonRankSumEx2 {
    public static void main(String args[]) {
        double[] x = {7.3, 6.9, 7.2, 7.8, 7.2};
        double[] y = {7.4, 6.8, 6.9, 6.7, 7.1};
        String[] labels = {
            "Wilcoxon W statistic .....................",
            "2*E(W) - W ...............................",
            "p-value ................................. ",
            "Adjusted Wilcoxon statistic ..............",
            "Adjusted 2*E(W) - W ......................",
            "Adjusted p-value ........................ ",
            "W statistics for averaged ranks...........",
            "Standard error of W (averaged ranks) ...... ",
            "Standard normal score of W (averaged ranks) ",
            "Two-sided p-value of W (averaged ranks) ... "
        };

        WilcoxonRankSum wilcoxon = new WilcoxonRankSum(x, y);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(3);

        // Trun off printing of warning messages.
        Warning.setOut(null);
        wilcoxon.compute();
        double[] stat = wilcoxon.getStatistics();

        for (int i = 0; i < 10; i++) {
            System.out.println(labels[i] + " " + nf.format(stat[i]));
        }
    }
}
```

## Output

```
Wilcoxon W statistic ..................... 34.000
2*E(W) - W ............................... 21.000
p-value .................................  0.110
Adjusted Wilcoxon statistic .............. 35.000
```

```
Adjusted 2*E(W) - W ...................... 20.000
Adjusted p-value .........................  0.075
W statistics for averaged ranks............ 34.500
Standard error of W (averaged ranks) ......  4.758
Standard normal score of W (averaged ranks)  1.471
Two-sided p-value of W (averaged ranks) ...  0.141
```

# Chapter 17

# Tests of Goodness of Fit

Classes



## Usage Notes

The classes in this chapter are used to test for goodness of fit. The goodness-of-fit tests are described in Conover (1980). There is a goodness-of-fit test for general distributions and a chi-squared test. The user supplies the hypothesized cumulative distribution function for the test. There is a class that can be used to test specifically for the normal distribution.

The chi-squared goodness-of-fit test may be used with discrete as well as continuous distributions. The chi-squared goodness-of-fit test allows for missing values (NaN, not a number) in the input data.

## *class* ChiSquaredTest

Chi-squared goodness-of-fit test.

`ChiSquaredTest` performs a chi-squared goodness-of-fit test that a random sample of observations is distributed according to a specified theoretical cumulative distribution. The theoretical distribution, which may be continuous, discrete, or a mixture of discrete and continuous distributions, is specified via a user-defined function $F$ where $F$ implements `CdfFuntion`. Because the user is allowed to specify a range for the observations in the `setRange` method, a test that is conditional upon the specified range is performed.

`ChiSquaredTest` can be constructed in two different ways. The intervals can be specified via the array cutpoints. Otherwise, the number of cutpoints can be given and equiprobable intervals computed by the constructor. The observations are divided into these intervals. Regardless of the method used to obtain them, the intervals are such that the lower endpoint is not included in the interval while the upper endpoint is always included. The user should determine the cutpoints when the cumulative distribution function has discrete elements since `ChiSquaredTest` cannot determine them in this case.

By default, the lower and upper endpoints of the first and last intervals are $-\infty$ and $+\infty$, respectively. The method `setRange` can be used to change the range.

A tally of counts is maintained for the observations in $x$ as follows:

If the cutpoints are specified by the user, the tally is made in the interval to which $x_i$ belongs, using the user-specified endpoints.

If the cutpoints are determined by the class then the cumulative probability at $x_i$, $F(x_i)$, is computed using `cdf`.

The tally for $x_i$ is made in interval number $\lfloor mF(x) + 1 \rfloor$, where $m$ is the number of categories and $\lfloor . \rfloor$ is the function that takes the greatest integer that is no larger than the argument of the function. If the cutpoints are specified by the user, the tally is made in the interval to which $x_i$ belongs using the endpoints specified by the user. Thus, if the computer time required to calculate the cumulative distribution function is large, user-specified cutpoints may be preferred in order to reduce the total computing time.

If the expected count in any cell is less than 1, then a rule of thumb is that the chi-squared approximation may be suspect. A warning message to this effect is issued in this case, as well as when an expected value is less than 5.

## Declaration

public class com.imsl.stat.ChiSquaredTest
**extends** java.lang.Object

## Inner Classes

### *class* ChiSquaredTest.NotCDFException

The function is not a Cumulative Distribution Function (CDF).

#### Declaration

public static class com.imsl.stat.ChiSquaredTest.NotCDFException
**extends** com.imsl.IMSLRuntimeException (page 1242)

#### Constructor

---

- *ChiSquaredTest.NotCDFException*
  `public` **ChiSquaredTest.NotCDFException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* ChiSquaredTest.NoObservationsException

There are no observations.

#### Declaration

public static class com.imsl.stat.ChiSquaredTest.NoObservationsException
**extends** com.imsl.IMSLRuntimeException (page 1242)

#### Constructor

---

- *ChiSquaredTest.NoObservationsException*
  `public` **ChiSquaredTest.NoObservationsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* ChiSquaredTest.DidNotConvergeException

The iteration did not converge

#### Declaration

public static class com.imsl.stat.ChiSquaredTest.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ChiSquaredTest.DidNotConvergeException*
  `public` **ChiSquaredTest.DidNotConvergeException**`( java.lang.String` **message** `)`

- *ChiSquaredTest.DidNotConvergeException*
  `public` **ChiSquaredTest.DidNotConvergeException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### Constructors

---

- *ChiSquaredTest*
  `public` **ChiSquaredTest**`( CdfFunction` **cdf**`, double[]` **cutpoints**`, int` **nParameters** `) throws com.imsl.stat.ChiSquaredTest.NotCDFException`
  - **Description**
    Constructor for the Chi-squared goodness-of-fit test.
  - **Parameters**
    - `cdf` – a `CdfFunction` object that implements the CdfFunction interface
    - `cutpoints` – a `double` array containing the cutpoints
    - `nParameters` – an `int` which specifies the number of parameters estimated in computing the Cdf

- *ChiSquaredTest*
  `public` **ChiSquaredTest**`( CdfFunction` **cdf**`, int` **nCutpoints**`, int` **nParameters** `) throws com.imsl.stat.ChiSquaredTest.NotCDFException, com.imsl.stat.InverseCdf.DidNotConvergeException`
  - **Description**
    Constructor for the Chi-squared goodness-of-fit test
  - **Parameters**
    - `cdf` – a `CdfFunction` object that implements the CdfFunction interface
    - `nCutpoints` – an `int`, the number of cutpoints
    - `nParameters` – an `int` which specifies the number of parameters estimated in computing the Cdf

## Methods

---

- *getCellCounts*
  `public double[]` **getCellCounts**`( )`
  - **Description**
    Returns the cell counts.
  - **Returns** – a `double` array which contains the number of actual observations in each cell.

---

- *getChiSquared*
  `public double` **getChiSquared**`( ) throws com.imsl.stat.ChiSquaredTest.NotCDFException`
  - **Description**
    Returns the chi-squared statistic.
  - **Returns** – a `double`, the chi-squared statistic

---

- *getCutpoints*
  `public double[]` **getCutpoints**`( )`
  - **Description**
    Returns the cutpoints.
  - **Returns** – a `double` array which contains the cutpoints

---

- *getDegreesOfFreedom*
  `public double` **getDegreesOfFreedom**`( ) throws com.imsl.stat.ChiSquaredTest.NotCDFException`
  - **Description**
    Returns the degrees of freedom in chi-squared.
  - **Returns** – a `double`, the degrees of freedom in the chi-squared statistic

---

- *getExpectedCounts*
  `public double[]` **getExpectedCounts**`( )`
  - **Description**
    Returns the expected counts.
  - **Returns** – a `double` array which contains the number of expected observations in each cell.

---

- *getP*
  `public double` **getP**`( ) throws`
  `com.imsl.stat.ChiSquaredTest.NotCDFException`

  - **Description**
    Returns the $p$-value for the chi-squared statistic.
  - **Returns** – a `double`, the $p$-value for the chi-squared statistic

---

- *setCutpoints*
  `public void` **setCutpoints**`( double[]` **cutpoints** `)`

  - **Description**
    Sets the cutpoints. The intervals defined by the cutpoints are such that the lower endpoint is not included while the upper endpoint is included in the interval.
  - **Parameters**
    * `cutpoints` – a `double` array which contains the cutpoints

---

- *setRange*
  `public void` **setRange**`( double` **lower**`, double` **upper** `) throws`
  `com.imsl.stat.ChiSquaredTest.NotCDFException`

  - **Description**
    Sets endpoints of the range of the distribution. Points outside of the range are ignored so that distributions conditional on the range can be used. In this case, the point lower is excluded from the first interval, but the point upper is included in the last interval. By default, a range on the whole real line is used.
  - **Parameters**
    * `lower` – a `double`, the lower range limit
    * `upper` – a `double`, the upper range limit

---

- *update*
  `public void` **update**`( double[]` **x**`, double[]` **freq** `) throws`
  `com.imsl.stat.ChiSquaredTest.NotCDFException`

  - **Description**
    Adds new observations to the test.
  - **Parameters**
    * `x` – a `double` array which contains the new observations to be added to the test
    * `freq` – a `double` array which contains the frequencies of the corresponding new observations in x

---

- *update*
  `public synchronized void` **`update`**`( double` **`x`**`, double` **`freq`** `) throws com.imsl.stat.ChiSquaredTest.NotCDFException`

  – **Description**
    Adds a new observation to the test.

  – **Parameters**
    * `x` – a `double`, the new observation to be added to the test
    * `freq` – a `double`, the frequency of the new observation, x

### Example: The Chi-squared Goodness-of-fit Test

In this example, a discrete binomial random sample of size 1000 with binomial parameter $p = 0.3$ and binomial sample size 5 is generated via Random.nextBinomial. Random.setSeed is first used to set the seed. After the ChiSquaredTest constructor is called, the random observations are added to the test one at a time to simulate streaming data. The Chi-squared statistic, $p$-value, and Degrees of freedom are then computed and printed.

```
import com.imsl.stat.*;

public class ChiSquaredTestEx1 {
    public static void main(String args[]) {
        // Seed the random number generator
        Random rn = new Random();
        rn.setSeed(123457);
        rn.setMultiplier(16807);

        // Construct a ChiSquaredTest object
        CdfFunction bindf = new CdfFunction() {
            public double cdf(double x) {
                return Cdf.binomial((int)x, 5, 0.3);
            }
        };

        double cutp[] = {0.5, 1.5, 2.5, 3.5, 4.5};
        int nParameters = 0;
        ChiSquaredTest cst = new ChiSquaredTest(bindf, cutp, nParameters);
        for (int i = 0; i < 1000; i++) {
            cst.update(rn.nextBinomial(5, 0.3), 1.0);
        }

        // Print goodness-of-fit test statistics
```

```
        System.out.println("The Chi-squared statistic is "
        + cst.getChiSquared());
        System.out.println("The P-value is "+cst.getP());
        System.out.println("The Degrees of freedom are "
        + cst.getDegreesOfFreedom());

    }
}
```

## Output

```
The Chi-squared statistic is 4.79629666357389
The P-value is 0.44124295720552564
The Degrees of freedom are 5.0
```

*Warning* com.imsl.stat.ChiSquaredTest: An expected value is less than five.

# *class* NormalityTest

Performs a test for normality.

Three methods are provided for testing normality: the Shapiro-Wilk W test, the Lilliefors test, and the chi-squared test.

**Shapiro-Wilk W Test**

The Shapiro-Wilk $W$ test is thought by D'Agostino and Stevens (1986, p. 406) to be one of the best omnibus tests of normality. The function is based on the approximations and code given by Royston (1982a, b, c). It can be used in samples as large as 2,000 or as small as 3. In the Shapiro and Wilk test, $W$ is given by

$$W = \left(\sum a_i x_{(i)}\right)^2 / \left(\sum (x_i - \bar{x})^2\right)$$

where $x_{(i)}$ is the $i$-th largest order statistic and $x$ is the sample mean. Royston (1982) gives approximations and tabled values that can be used to compute the coefficients $a_i, i = 1, \ldots, n$, and obtains the significance level of the $W$ statistic.

**Lilliefors Test**

This function computes Lilliefors test and its $p$-values for a normal distribution in which both the mean and variance are estimated. The one-sample, two-sided Kolmogorov-Smirnov statistic $D$ is first computed. The $p$-values are then computed using an analytic approximation given by Dallal and Wilkinson (1986). Because Dallal and Wilkinson give approximations in the range (0.01, 0.10) if the computed probability of a greater $D$ is less than 0.01, the $p$-value is set to 0.50. Note that because parameters are estimated, $p$-values in Lilliefors test are not the same as in the Kolmogorov-Smirnov Test.

Observations should not be tied. If tied observations are found, an informational message is printed. A general reference for the Lilliefors test is Conover (1980). The original reference for the test for normality is Lilliefors (1967).

**Chi-Squared Test**

This function computes the chi-squared statistic, its $p$-value, and the degrees of freedom of the test. Argument $n$ finds the number of intervals into which the observations are to be divided. The intervals are equiprobable except for the first and last interval, which are infinite in length.

If more flexibility is desired for the specification of intervals, the same test can be performed with class `ChiSquaredTest`.

## Declaration

public class com.imsl.stat.NormalityTest
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Class

### *class* NormalityTest.NoVariationInputException

There is no variation in the input data.

#### Declaration

public static class com.imsl.stat.NormalityTest.NoVariationInputException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *NormalityTest.NoVariationInputException*
  `public` **NormalityTest.NoVariationInputException**`(` `java.lang.String` **message** `)`

- *NormalityTest.NoVariationInputException*
  `public` **NormalityTest.NoVariationInputException**`(` `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

## Constructor

---

- *NormalityTest*
  `public` **NormalityTest**`(` `double[]` **x** `)`
  - **Description**
    Constructor for `NormalityTest`.
  - **Parameters**
    * `x` – A `double` array containing the observations. `x.length` must be in the range from 3 to 2,000, inclusive, for the Shapiro-Wilk W test and must be greater than 4 for the Lilliefors test.

## Methods

---

- *ChiSquaredTest*
  `public final double` **ChiSquaredTest**`(` `int` **n** `) throws`
  `com.imsl.stat.NormalityTest.NoVariationInputException,`
  `com.imsl.stat.InverseCdf.DidNotConvergeException`
  - **Description**
    Performs the chi-squared goodness-of-fit test.
  - **Parameters**
    * `n` – An `int` scalar containing the number of cells into which the observations are to be tallied.
  - **Returns** – A `double` scalar containing the p-value for the chi-squared goodness-of-fit test.
  - **Throws**
    * `com.imsl.stat.NormalityTest.NoVariationInputException` – is thrown if there is no variation in the input data.
    * `DidNotConvergeException` – is thrown if the iteration did not converge.

---

- *getChiSquared*
  `public double` **getChiSquared**`(` `)`

– **Description**
Returns the chi-square statistic for the chi-squared goodness-of-fit test.

– **Returns** – A `double` scalar containing the chi-square statistic. Returns `Double.NaN` for other tests.

---

- *getDegreesOfFreedom*
`public double` **getDegreesOfFreedom**`( )`

  – **Description**
  Returns the degrees of freedom for the chi-squared goodness-of-fit test.

  – **Returns** – A `double` scalar containing the degrees of freedom. Returns `Double.NaN` for other tests.

---

- *getMaxDifference*
`public double` **getMaxDifference**`( )`

  – **Description**
  Returns the maximum absolute difference between the empirical and the theoretical distributions for the Lilliefors test.

  – **Returns** – A `double` scalar containing the maximum absolute difference between the empirical and the theoretical distributions. Returns `Double.NaN` for other tests.

---

- *getShapiroWilkW*
`public double` **getShapiroWilkW**`( )`

  – **Description**
  Returns the Shapiro-Wilk W statistic for the Shapiro-Wilk W test.

  – **Returns** – A `double` scalar containing the Shapiro-Wilk W statistic. Returns `Double.NaN` for other tests.

---

- *LillieforsTest*
`public final double` **LillieforsTest**`( ) throws com.imsl.stat.NormalityTest.NoVariationInputException, com.imsl.stat.InverseCdf.DidNotConvergeException`

  – **Description**
  Performs the Lilliefors test.

  – **Returns** – A `double` scalar containing the p-value for the Lilliefors test. Probabilities less than 0.01 are reported as 0.01, and probabilities greater than 0.10 for the normal distribution are reported as 0.5. Otherwise, an approximate probability is computed.

  – **Throws**

* `com.imsl.stat.NormalityTest.NoVariationInputException` – is thrown if there is no variation in the input data.
* `DidNotConvergeException` – is thrown if the iteration did not converge.

---

- *ShapiroWilkWTest*
  `public final double` **ShapiroWilkWTest**`( ) throws com.imsl.stat.NormalityTest.NoVariationInputException, com.imsl.stat.InverseCdf.DidNotConvergeException`
  - **Description**
    Performs the Shapiro-Wilk W test.
  - **Returns** – A `double` scalar containing the p-value for the Shapiro-Wilk W test.
  - **Throws**
    * `com.imsl.stat.NormalityTest.NoVariationInputException` – is thrown if there is no variation in the input data.
    * `DidNotConvergeException` – is thrown if the iteration did not converge.

### Example: Shapiro-Wilk W Test

The following example is taken from Conover (1980, pp. 195, 364). The data consists of 50 two-digit numbers taken from a telephone book. The $W$ test fails to reject the null hypothesis of normality at the .05 level of significance.

```
import java.text.*;
import com.imsl.*;
import com.imsl.stat.*;

public class NormalityTestEx1 {
    public static void main(String args[]) throws Exception {
        double x[] = {23.0, 36.0, 54.0, 61.0, 73.0, 23.0, 37.0, 54.0, 61.0,
        73.0, 24.0, 40.0, 56.0, 62.0, 74.0, 27.0, 42.0, 57.0, 63.0, 75.0, 29.0,
        43.0, 57.0, 64.0, 77.0, 31.0, 43.0, 58.0, 65.0, 81.0, 32.0, 44.0, 58.0,
        66.0, 87.0, 33.0, 45.0, 58.0, 68.0, 89.0, 33.0, 48.0, 58.0, 68.0, 93.0,
        35.0, 48.0, 59.0, 70.0, 97.0};

        NormalityTest nt = new NormalityTest(x);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);

        System.out.println("p-value = " + nf.format(nt.ShapiroWilkWTest()));
        System.out.println("Shapiro Wilk W Statistic = " +
        nf.format(nt.getShapiroWilkW()));
```

```
    }
}
```

### Output

```
p-value = 0.2309
Shapiro Wilk W Statistic = 0.9642
```

# Chapter 18

# Time Series and Forecasting

Classes



## Usage Notes

The classes in this chapter assume the time series does not contain any missing observations. If missing values are present, they should be set to NaN (see `Double.NaN`), and the classes will return an appropriate error message. To enable fitting of the model, the missing values must be replaced by appropriate estimates.

## General Methodology

A major component of the model identification step concerns determining if a given time series is stationary. The sample correlation functions computed by the AutoCorrelation class methods `getAutoCorrelations` and `getPartialAutoCorrelations` may be used to diagnose the presence of nonstationarity in the data, as well as to indicate the type of transformation required to induce stationarity.

The "raw" data and sample correlation functions provide insight into the nature of the underlying model. Typically, this information is displayed in graphical form via time series plots, plots of the lagged data, and various correlation function plots.

## ARIMA Model (Autoregressive Integrated Moving Average)

A small, yet comprehensive, class of stationary time-series models consists of the nonseasonal ARMA processes defined by

$$\phi\left(B\right)\left(W_t - \mu\right) = \theta\left(B\right)A_t, \qquad t \in Z$$

where $Z = \ldots, -2, -1, 0, 1, 2, \ldots$ denotes the set of integers, $B$ is the backward shift operator defined by $B^k W_t = W_{t-k}, \mu$ is the mean of $W_t$, and the following equations are true:

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p, p \geq 0$$

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \cdots - \theta_q B^q, q \geq 0$$

The model is of order $(p,\ q)$ and is referred to as an ARMA $(p,\ q)$ model.

An equivalent version of the ARMA $(p,\ q)$ model is given by

$$\phi(B)W_t = \theta_0 + \theta(B)A_i, \quad t \in Z$$

where $\theta_0$ is an overall constant defined by the following:

$$\theta_0 = \mu\left(1 - \sum_{i=1}^{p}\phi_i\right)$$

See Box and Jenkins (1976, pp. 92-93) for a discussion of the meaning and usefulness of the overall constant.

If the "raw" data, $\{Z_t\}$, are homogeneous and nonstationary, then differencing using the Difference class induces stationarity, and the model is called ARIMA (AutoRegressive Integrated Moving Average). Parameter estimation is performed on the stationary time series $W_t, = \Delta^d Z_t$, where $\Delta^d = (1 - B)^d$ is the backward difference operator with period 1 and order $d$, d $> 0$.

Typically, the method of moments includes use of `METHOD_OF_MOMENTS` in a call to the `compute` method in the ARMA class for preliminary parameter estimates. These estimates can be used as initial values into the least-squares procedure by using `LEAST_SQUARES` in a call to the `compute` method in the ARMA class. Other initial estimates provided by the user can be used. The least-squares procedure can be used to compute conditional or unconditional least-squares estimates of the parameters, depending on the choice of the backcasting length. The parameter estimates from either the method of moments or least-squares procedures can be used in the `forecast` method. The functions for preliminary parameter estimation, least-squares parameter estimation, and forecasting follow the approach of Box and Jenkins (1976, Programs 2-4, pp. 498-509).

## *class* AutoCorrelation

Computes the sample autocorrelation function of a stationary time series.

`AutoCorrelation` estimates the autocorrelation function of a stationary time series given a sample of $n$ observations $\{X_t\}$ for t $= 1, 2, \ldots,$ n.

Let

$$\hat{\mu} = \text{xmean}$$

be the estimate of the mean $\mu$ of the time series $\{X_t\}$ where

$$\hat{\mu} = \begin{cases} pa \quad \begin{array}{ll} \mu & \text{for } \mu \text{ known} \\ \frac{1}{n}\sum\limits_{t=1}^{n} X_t & \text{for } \mu \text{ unknown} \end{array} \end{cases}$$

The autocovariance function $\sigma(k)$ is estimated by

$$\hat{\sigma}\left(k\right) = \frac{1}{n}\sum_{t=1}^{n-k}\left(X_t - \hat{\mu}\right)\left(X_{t+k} - \hat{\mu}\right), \qquad \text{k=0,1,\ldots,K}$$

where $K =$ `maximum_lag`. Note that $\hat{\sigma}(0)$ is an estimate of the sample variance. The autocorrelation function $\rho(k)$ is estimated by

$$\hat{\rho}(k) = \frac{\hat{\sigma}(k)}{\hat{\sigma}(0)}, \qquad k = 0, 1, \ldots, K$$

Note that $\hat{\rho}(0) \equiv 1$ by definition.

The standard errors of sample autocorrelations may be optionally computed according to the *getStandardErrors* method argument `stderrMethod`. One method (Bartlett 1946) is based on a general asymptotic expression for the variance of the sample autocorrelation coefficient of a stationary time series with independent, identically distributed normal errors. The theoretical formula is

$$\mathrm{var}\{\hat{\rho}(k)\} = \frac{1}{n} \sum_{i=-\infty}^{\infty} \left[\rho^2(i) + \rho(i-k)\rho(i+k) - 4\rho(i)\rho(k)\rho(i-k) + 2\rho^2(i)\rho^2(k)\right]$$

where $\hat{\rho}(k)$ assumes $\mu$ is unknown. For computational purposes, the autocorrelations $\rho(k)$ are replaced by their estimates $\hat{\rho}(k)$ for $|k| \leq K$, and the limits of summation are bounded because of the assumption that $\rho(k) = 0$ for all $k$ such that $|k| > K$.

A second method (Moran 1947) utilizes an exact formula for the variance of the sample autocorrelation coefficient of a random process with independent, identically distributed normal errors. The theoretical formula is

$$var\{\hat{\rho}(k)\} = \frac{n-k}{n(n+2)}$$

where $\mu$ is assumed to be equal to zero. Note that this formula does not depend on the autocorrelation function.

The method `getPartialAutoCorrelations` estimates the partial autocorrelations of the stationary time series given K = `maximum_lag` sample autocorrelations $\hat{\rho}(k)$ for $k$=0,1,...,K. Consider the AR($k$) process defined by

$$X_t = \phi_{k1} X_{t-1} + \phi_{k2} X_{t-2} + \cdots + \phi_{kk} X_{t-k} + A_t$$

where $\phi_{kj}$ denotes the $j$-th coefficient in the process. The set of estimates $\{\hat{\phi}_{kk}\}$ for $k = 1$, ..., K is the sample partial autocorrelation function. The autoregressive parameters $\{\hat{\phi}_{kj}\}$ for $j = 1$, ..., $k$ are approximated by Yule-Walker estimates for successive AR($k$) models where $k = 1$, ..., K. Based on the sample Yule-Walker equations

$$\hat{\rho}(j) = \hat{\phi}_{k1}\hat{\rho}(j-1) + \hat{\phi}_{k2}\hat{\rho}(j-2) + \cdots + \hat{\phi}_{kk}\hat{\rho}(j-k), \qquad \text{j = 1,2,\ldots,k}$$

a recursive relationship for $k$=1, ..., K was developed by Durbin (1960). The equations are given by

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & \text{for k = 1} \\ \dfrac{\hat{\rho}(k) - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(k-j)}{1 - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(j)} & \text{for k = 2, \ldots, K} \end{cases}$$

and

$$\hat{\phi}_{kj} = \begin{cases} \hat{\phi}_{k-1,j} - \hat{\phi}_{kk}\hat{\phi}_{k-1,k-j} & \text{for j} = 1, 2, \ldots, \text{k} - 1 \\ \hat{\phi}_{kk} & \text{for j} = \text{k} \end{cases}$$

This procedure is sensitive to rounding error and should not be used if the parameters are near the nonstationarity boundary. A possible alternative would be to estimate $\{\phi_{kk}\}$ for successive AR($k$) models using least or maximum likelihood. Based on the hypothesis that the true process is AR($p$), Box and Jenkins (1976, page 65) note

$$\text{var}\{\hat{\phi}_{kk}\} \simeq \frac{1}{n} \quad \text{k} \geq \text{p} + 1$$

See Box and Jenkins (1976, pages 82-84) for more information concerning the partial autocorrelation function.

## Declaration

public class com.imsl.stat.AutoCorrelation
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Class

### *class* AutoCorrelation.NonPosVariancesException

The problem is ill-conditioned.

#### Declaration

public static class com.imsl.stat.AutoCorrelation.NonPosVariancesException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *AutoCorrelation.NonPosVariancesException*
  `public` **AutoCorrelation.NonPosVariancesException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `NonPosVariancesException` with the specified detail message. A detail message is a `String` that describes this particular exception.

- **Parameters**
  - `message` – the detail message

---

- *AutoCorrelation.NonPosVariancesException*
  `public` **AutoCorrelation.NonPosVariancesException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an `NonPosVariancesException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    - `key` – the key of the error message in the resource bundle
    - `arguments` – an array containing arguments used within the error message string

## Fields

---

- public static final `int` **BARTLETTS_FORMULA**
  - Indicates standard error computation using Bartlett's formula.
- public static final `int` **MORANS_FORMULA**
  - Indicates standard error computation using Moran's formula.

## Constructor

---

- *AutoCorrelation*
  `public` **AutoCorrelation**`( double[]` **x**`, int` **maximum_lag** `)`
  - **Description**
    Constructor to compute the sample autocorrelation function of a stationary time series.
  - **Parameters**
    - `x` – a one-dimensional `double` array containing the stationary time series
    - `maximum_lag` – an `int` containing the maximum lag of autocovariance, autocorrelations, and standard errors of autocorrelations to be computed. `maximum_lag` must be greater than or equal to 1 and less than the number of observations in `x`

## Methods

- *getAutoCorrelations*
  `public double[]` **getAutoCorrelations**`( )`
  - **Description**
    Returns the autocorrelations of the time series `x`.
  - **Returns** – a `double` array of length `maximum_lag` +1 containing the autocorrelations of the time series `x`. The *0*-th element of this array is 1. The *k*-th element of this array contains the autocorrelation of lag k where $k = 1, \ldots,$ `maximum_lag`.

- *getAutoCovariances*
  `public double[]` **getAutoCovariances**`( ) throws com.imsl.stat.AutoCorrelation.NonPosVariancesException`
  - **Description**
    Returns the variance and autocovariances of the time series `x`.
  - **Returns** – a `double` array of length `maximum_lag` +1 containing the variances and autocovariances of the time series `x`. The *0*-th element of the array contains the variance of the time series `x`. The *k*-th element contains the autocovariance of lag k where $k = 1, \ldots,$ `maximum_lag`.
  - **Throws**
    * `com.imsl.stat.AutoCorrelation.NonPosVariancesException` – is thrown if the problem is ill-conditioned

- *getMean*
  `public double` **getMean**`( )`
  - **Description**
    Returns the mean of the time series `x`.
  - **Returns** – a `double` containing the mean

- *getPartialAutoCorrelations*
  `public double[]` **getPartialAutoCorrelations**`( )`
  - **Description**
    Returns the sample partial autocorrelation function of the stationary time series `x`.
  - **Returns** – a `double` array of length `maximum_lag` containing the partial autocorrelations of the time series `x`.

- *getStandardErrors*
  `public double[]` **getStandardErrors**`( int` **stderrMethod** `)`
  - **Description**
    Returns the standard errors of the autocorrelations of the time series `x`. Method of computation for standard errors of the autocorrelation is chosen by the `stderrMethod` parameter. If `stderrMethod` is set to `BARTLETTS_FORMULA`, Bartlett's formula is used to compute the standard errors of autocorrelations. If `stderrMethod` is set to `MORANS_FORMULA`, Moran's formula is used to compute the standard errors of autocorrelations.
  - **Parameters**
    * `stderrMethod` – an `int` specifying the method to compute the standard errors of autocorrelations of the time series `x`
  - **Returns** – a `double` array of length `maximum_lag` containing the standard errors of the autocorrelations of the time series `x`

---

- *getVariance*
  `public double` **getVariance**`( )`
  - **Description**
    Returns the variance of the time series `x`.
  - **Returns** – a `double` containing the variance of the time series `x`

---

- *setMean*
  `public void` **setMean**`( double` **mean** `)`
  - **Description**
    Estimate mean of the time series `x`.
  - **Parameters**
    * `mean` – a `double` containing the estimate mean of the time series `x`.

### Example 1: AutoCorrelation

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. This example computes the estimated autocovariances, estimated autocorrelations, and estimated standard errors of the autocorrelations using both Bartletts and Moran formulas.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
```

```
public class AutoCorrelationEx1 {
    public static void main(String args[]) throws Exception {
        double[] x = {100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5,
        154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9,
        132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16,
        6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1,
        8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4,
        23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5,
        67, 71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2,
        85.8, 63.2, 36.8, 24.2, 10.7, 15, 40.1, 61.5, 98.5,
        124.3, 95.9, 66.5, 64.5, 54.2, 39, 20.6, 6.7, 4.3, 22.8,
        54.8, 93.8, 95.7, 77.2, 59.1, 44, 47, 30.5, 16.3, 7.3,
        37.3, 73.9};

        AutoCorrelation ac = new AutoCorrelation(x, 20);


        new PrintMatrix("AutoCovariances are:  ").print
                       (ac.getAutoCovariances());
        System.out.println();
        new PrintMatrix("AutoCorrelations are:  ").print
                       (ac.getAutoCorrelations());
        System.out.println("Mean = "+ac.getMean());
        System.out.println();
        new PrintMatrix("Standard Error using Bartlett are:  ").print
                       (ac.getStandardErrors(ac.BARTLETTS_FORMULA));
        System.out.println();
        new PrintMatrix("Standard Error using Moran are:  ").print
                       (ac.getStandardErrors(ac.MORANS_FORMULA));
        System.out.println();
        new PrintMatrix("Partial AutoCovariances:  ").print
                       (ac.getPartialAutoCorrelations());
        ac.setMean(50);
        new PrintMatrix("AutoCovariances are:  ").print
                       (ac.getAutoCovariances());
        System.out.println();
        new PrintMatrix("AutoCorrelations are:  ").print
                       (ac.getAutoCorrelations());
        System.out.println();
        new PrintMatrix("Standard Error using Bartlett are:  ").print
                       (ac.getStandardErrors(ac.BARTLETTS_FORMULA));
```

```
    }
}
```

### Output

```
AutoCovariances are:
         0
 0  1,382.908
 1  1,115.029
 2    592.004
 3     95.297
 4   -235.952
 5   -370.011
 6   -294.255
 7    -60.442
 8    227.633
 9    458.381
10    567.841
11    546.122
12    398.937
13    197.757
14     26.891
15    -77.281
16   -143.733
17   -202.048
18   -245.372
19   -230.816
20   -142.879


AutoCorrelations are:
      0
 0   1
 1   0.806
 2   0.428
 3   0.069
 4  -0.171
 5  -0.268
 6  -0.213
 7  -0.044
 8   0.165
```

```
 9   0.331
10   0.411
11   0.395
12   0.288
13   0.143
14   0.019
15  -0.056
16  -0.104
17  -0.146
18  -0.177
19  -0.167
20  -0.103

Mean = 46.976000000000006

Standard Error using Bartlett are:
      0
 0  0.035
 1  0.096
 2  0.157
 3  0.206
 4  0.231
 5  0.229
 6  0.209
 7  0.178
 8  0.146
 9  0.134
10  0.151
11  0.174
12  0.191
13  0.195
14  0.196
15  0.196
16  0.196
17  0.199
18  0.205
19  0.209


Standard Error using Moran are:
      0
 0  0.099
```

```
 1  0.098
 2  0.098
 3  0.097
 4  0.097
 5  0.096
 6  0.095
 7  0.095
 8  0.094
 9  0.094
10  0.093
11  0.093
12  0.092
13  0.092
14  0.091
15  0.091
16  0.09
17  0.09
18  0.089
19  0.089


Partial AutoCovariances:
      0
 0   0.806
 1  -0.635
 2   0.078
 3  -0.059
 4  -0.001
 5   0.172
 6   0.109
 7   0.11
 8   0.079
 9   0.079
10   0.069
11  -0.038
12   0.081
13   0.033
14  -0.035
15  -0.131
16  -0.155
17  -0.119
18  -0.016
```

```
19  -0.004

AutoCovariances are:
        0
 0  1,392.053
 1  1,126.524
 2    604.162
 3    106.754
 4   -225.882
 5   -361.026
 6   -286.57
 7    -53.76
 8    235.966
 9    470.786
10    584.014
11    564.764
12    418.363
13    216.104
14     43.125
15    -63.468
16   -131.501
17   -189.063
18   -229.689
19   -212.156
20   -121.569


AutoCorrelations are:
      0
 0   1
 1   0.809
 2   0.434
 3   0.077
 4  -0.162
 5  -0.259
 6  -0.206
 7  -0.039
 8   0.17
 9   0.338
10   0.42
11   0.406
12   0.301
```

```
13   0.155
14   0.031
15  -0.046
16  -0.094
17  -0.136
18  -0.165
19  -0.152
20  -0.087


Standard Error using Bartlett are:
      0
 0  0.034
 1  0.097
 2  0.159
 3  0.21
 4  0.236
 5  0.233
 6  0.212
 7  0.18
 8  0.147
 9  0.134
10  0.148
11  0.172
12  0.19
13  0.197
14  0.198
15  0.198
16  0.198
17  0.201
18  0.207
19  0.21
```

## *class* **CrossCorrelation**

Computes the sample cross-correlation function of two stationary time series.

`CrossCorrelation` estimates the cross-correlation function of two jointly stationary time series given a sample of $n =$ `x.length` observations $\{X_t\}$ and $\{Y_t\}$ for $t = 1,2, ..., n$.

Let

$$\hat{\mu}_x = \text{xmean}$$

be the estimate of the mean $\mu_X$ of the time series $\{X_t\}$ where

$$\hat{\mu}_X = \begin{cases} \mu_X & \text{for } \mu_X \text{ known} \\ \frac{1}{n}\sum\limits_{t=1}^{n} X_t & \text{for } \mu_X \text{ unknown} \end{cases}$$

The autocovariance function of $\{X_t\}$, $\sigma_X(k)$, is estimated by

$$\hat{\sigma}_X(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu}_X)(X_{t+k} - \hat{\mu}_X), \qquad \text{k=0,1,...,K}$$

where K = `maximum_lag`. Note that $\hat{\sigma}_X(0)$ is equivalent to the sample variance of `x` returned by method `getVarianceX`. The autocorrelation function $\rho_X(k)$ is estimated by

$$\hat{\rho}_X(k) = \frac{\hat{\sigma}_X(k)}{\hat{\sigma}_X(0)}, \qquad k = 0, 1, \ldots, K$$

Note that $\hat{\rho}_x(0) \equiv 1$ by definition. Let

$$\hat{\mu}_Y = \text{ymean}, \hat{\sigma}_Y(k), \text{and} \hat{\rho}_Y(k)$$

be similarly defined.

The cross-covariance function $\sigma_{XY}(k)$ is estimated by

$$\hat{\sigma}_{XY}(k) = \begin{cases} \frac{1}{n}\sum\limits_{t=1}^{n-k}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = 0, 1, \ldots, K \\ \frac{1}{n}\sum\limits_{t=1-k}^{n}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = -1, -2, \ldots, -K \end{cases}$$

The cross-correlation function $\rho_{XY}(k)$ is estimated by

$$\hat{\rho}_{XY}(k) = \frac{\hat{\sigma}_{XY}(k)}{[\hat{\sigma}_X(0)\hat{\sigma}_Y(0)]^{\frac{1}{2}}} \quad k = 0, \pm 1, \ldots, \pm K$$

The standard errors of the sample cross-correlations may be optionally computed according to the *getStandardErrors* method argument `stderrMethod`. One method is based on a general asymptotic expression for the variance of the sample cross-correlation coefficient of two jointly stationary time series with independent, identically distributed normal errors given by Bartlet (1978, page 352). The theoretical formula is

$$\begin{aligned} \text{var}\{\hat{\rho}_{XY}(k)\} = \tfrac{1}{n-k}\sum_{i=-\infty}^{\infty} [\ &\rho_X(i) + \rho_{XY}(i-k)\rho_{XY}(i+k) \\ &-2\rho_{XY}(k)\{\rho_X(i)\rho_{XY}(i+k) + \rho_{XY}(-i)\rho_Y(i+k)\} \\ &+\rho_{XY}^2(k)\{\rho_X(i) + \tfrac{1}{2}\rho_X^2(i) + \tfrac{1}{2}\rho_Y^2(i)\}\ ] \end{aligned}$$

For computational purposes, the autocorrelations $\rho_X(k)$ and $\rho_Y(k)$ and the cross-correlations $\rho_{XY}(k)$ are replaced by their corresponding estimates for $|k| \le K$, and the limits of summation are equal to zero for all $k$ such that $|k| > K$.

A second method evaluates Bartlett's formula under the additional assumption that the two series have no cross-correlation. The theoretical formula is

$$\mathrm{var}\{\hat{\rho}_{XY}(k)\} = \frac{1}{n-k} \sum_{i=-\infty}^{\infty} \rho_X(i)\rho_Y(i) \qquad k \ge 0$$

For additional special cases of Bartlett's formula, see Box and Jenkins (1976, page 377).

An important property of the cross-covariance coefficient is $\sigma_{XY}(k) = \sigma_{YX}(-k)$ for $k \ge 0$. This result is used in the computation of the standard error of the sample cross-correlation for lag $k < 0$. In general, the cross-covariance function is not symmetric about zero so both positive and negative lags are of interest.

## Declaration

public class com.imsl.stat.CrossCorrelation
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Class

### *class* **CrossCorrelation.NonPosVariancesException**

The problem is ill-conditioned.

#### Declaration

public static class com.imsl.stat.CrossCorrelation.NonPosVariancesException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *CrossCorrelation.NonPosVariancesException*
  `public` **CrossCorrelation.NonPosVariancesException**( `java.lang.String` **message** )

- *CrossCorrelation.NonPosVariancesException*
  `public` **CrossCorrelation.NonPosVariancesException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## Fields

- public static final `int` **BARTLETTS_FORMULA**
  - Indicates standard error computation using Bartlett's formula.
- public static final `int` **BARTLETTS_FORMULA_NOCC**
  - Indicates standard error computation using Bartlett's formula with the assumption of no cross-correlation.

## Constructor

- *CrossCorrelation*
  `public` **CrossCorrelation**`( double[]` **x**`, double[]` **y**`, int` **maximum_lag** `)`
  - **Description**
    Constructor to compute the sample cross-correlation function of two stationary time series.
  - **Parameters**
    * `x` – A one-dimensional `double` array containing the first stationary time series.
    * `y` – A one-dimensional `double` array containing the second stationary time series.
    * `maximum_lag` – An `int` containing the maximum lag of the cross-covariance and cross-correlations to be computed. `maximum_lag` must be greater than or equal to 1 and less than the minimum of the number of observations of `x` and `y`.

## Methods

- *getAutoCorrelationX*
  `public double[]` **getAutoCorrelationX**`( ) throws com.imsl.stat.CrossCorrelation.NonPosVariancesException`

– **Description**
Returns the autocorrelations of the time series `x`.

– **Returns** – A `double` array of length `maximum_lag` +1 containing the autocorrelations of the time series `x`. The *0*-th element of this array is 1. The *k*-th element of this array contains the autocorrelation of lag $k$ where $k = 1, ...,$ `maximum_lag`.

---

- *getAutoCorrelationY*
`public double[]` **getAutoCorrelationY**`( ) throws com.imsl.stat.CrossCorrelation.NonPosVariancesException`

  – **Description**
  Returns the autocorrelations of the time series `y`.

  – **Returns** – A `double` array of length `maximum_lag` +1 containing the autocorrelations of the time series `y`. The *0*-th element of this array is 1. The *k*-th element of this array contains the autocorrelation of lag $k$ where $k = 1, ...,$ `maximum_lag`.

---

- *getAutoCovarianceX*
`public double[]` **getAutoCovarianceX**`( ) throws com.imsl.stat.CrossCorrelation.NonPosVariancesException`

  – **Description**
  Returns the autocovariances of the time series `x`.

  – **Returns** – A `double` array of length `maximum_lag` +1 containing the variances and autocovariances of the time series `x`. The *0*-th element of the array contains the variance of the time series `x`. The *k*-th element contains the autocovariance of lag $k$ where $k = 1, ...,$ `maximum_lag`.

---

- *getAutoCovarianceY*
`public double[]` **getAutoCovarianceY**`( ) throws com.imsl.stat.CrossCorrelation.NonPosVariancesException`

  – **Description**
  Returns the autocovariances of the time series `y`.

  – **Returns** – A `double` array of length `maximum_lag` +1 containing the variances and autocovariances of the time series `y`. The *0*-th element of the array contains the variance of the time series `x`. The *k*-th element contains the autocovariance of lag $k$ where $k = 1, ...,$ `maximum_lag`.

---

- *getCrossCorrelation*
`public double[]` **getCrossCorrelation**`( ) throws com.imsl.stat.CrossCorrelation.NonPosVariancesException`

– **Description**
Returns the cross-correlations between the time series `x` and `y`.

– **Returns** – A `double` array of length 2 * `maximum_lag` +1 containing the cross-correlations between the time series `x` and `y`. The cross-correlation between `x` and `y` at lag $k$, where $k$ = `-maximum_lag` ,..., 0, 1,...,`maximum_lag`, corresponds to output array indices `0, 1,..., (2*maximum_lag)`.

---

- *getCrossCovariance*

`public double[]` **getCrossCovariance**`( )`

– **Description**
Returns the cross-covariances between the time series `x` and `y`.

– **Returns** – A `double` array of length 2 * `maximum_lag` +1 containing the cross-covariances between the time series `x` and `y`. The cross-covariance between `x` and `y` at lag $k$, where $k$ = `-maximum_lag` ,..., 0, 1,...,`maximum_lag`, corresponds to output array indices `0, 1,..., (2*maximum_lag)`.

---

- *getMeanX*

`public double` **getMeanX**`( )`

– **Description**
Returns the mean of the time series `x`.

– **Returns** – A `double` containing the mean of the time series `x`.

---

- *getMeanY*

`public double` **getMeanY**`( )`

– **Description**
Returns the mean of the time series `y`.

– **Returns** – A `double` containing the mean of the time series `y`.

---

- *getStandardErrors*

`public double[]` **getStandardErrors**`( int` **stderrMethod** `) throws com.imsl.stat.CrossCorrelation.NonPosVariancesException`

– **Description**
Returns the standard errors of the cross-correlations between the time series `x` and `y`. Method of computation for standard errors of the cross-correlation is determined by the `stderrMethod` parameter. If `stderrMethod` is set to BARTLETTS_FORMULA, Bartlett's formula is used to compute the standard errors of cross-correlations. If `stderrMethod` is set to BARTLETTS_FORMULA_NOCC, Bartlett's formula is used to compute the standard errors of cross-correlations, with the assumption of no cross-correlation.

– **Parameters**

  * `stderrMethod` – An `int` specifying the method to compute the standard errors of cross-correlations between the time series `x` and `y`.

– **Returns** – A `double` array of length 2 * `maximum_lag` + 1 containing the standard errors of the cross-correlations between the time series `x` and `y`. The standard error of cross-correlations between `x` and `y` at lag $k$, where $k =$ `-maximum_lag`,..., 0, 1,..., `maximum_lag`, corresponds to output array indices 0, 1,..., `(2*maximum_lag)`.

---

• *getVarianceX*
`public double` **getVarianceX**`( ) throws`
`com.imsl.stat.CrossCorrelation.NonPosVariancesException`

– **Description**
Returns the variance of time series `x`.

– **Returns** – A `double` containing the variance of the time series `x`.

---

• *getVarianceY*
`public double` **getVarianceY**`( ) throws`
`com.imsl.stat.CrossCorrelation.NonPosVariancesException`

– **Description**
Returns the variance of time series `y`.

– **Returns** – A `double` containing the variance of the time series `y`.

---

• *setMeanX*
`public void` **setMeanX**`( double` **mean** `)`

– **Description**
Estimate of the mean of time series `x`.

– **Parameters**

  * `mean` – A `double` containing the estimate mean of the time series `x`.

---

• *setMeanY*
`public void` **setMeanY**`( double` **mean** `)`

– **Description**
Estimate of the mean of time series `y`.

– **Parameters**

  * `mean` – A `double` containing the estimate mean of the time series `y`.

### Example 1: CrossCorrelation

Consider the Gas Furnace Data (Box and Jenkins 1976, pages 532-533) where $X$ is the input gas rate in cubic feet/minute and $Y$ is the percent $CO_2$ in the outlet gas. The `CrossCorrelation` methods `getCrossCovariance` and `getCrossCorrelation` are used to compute the cross-covariances and cross-correlations between time series $X$ and $Y$ with lags from `-maximum_lag` = -10 through lag `maximum_lag` = 10. In addition, the estimated standard errors of the estimated cross-correlations are computed. In the first invocation of method `getStandardErrors` stderrMethod = BARTLETTS_FORMULA, the standard errors are based on the assumption that autocorrelations and cross-correlations for lags greater than `maximum_lag` or less than `-maximum_lag` are zero, In the second invocation of method getStandardErrors with stderrMethod = BARTLETTS_FORMULA_NOCC, the standard errors are based on the additional assumption that all cross-correlations for `X` and `Y` are zero.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class CrossCorrelationEx1 {

    public static void main(String args[]) throws Exception {
        double[] x2 = {100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5,
        154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9,
        132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16,
        6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1,
        8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4,
        23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5,
        67, 71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2,
        85.8, 63.2, 36.8, 24.2, 10.7, 15, 40.1, 61.5, 98.5,
        124.3, 95.9, 66.5, 64.5, 54.2, 39, 20.6, 6.7, 4.3, 22.8,
        54.8, 93.8, 95.7, 77.2, 59.1, 44, 47, 30.5, 16.3, 7.3,
        37.3, 73.9};

        double[] x = {-0.109, 0.0, 0.178, 0.339, 0.373, 0.441, 0.461,
          0.348, 0.127, -0.18, -0.588, -1.055, -1.421, -1.52, -1.302,
          -0.814, -0.475, -0.193, 0.088, 0.435, 0.771, 0.866, 0.875,
          0.891, 0.987, 1.263, 1.775, 1.976, 1.934, 1.866, 1.832,
          1.767, 1.608, 1.265, 0.79, 0.36, 0.115, 0.088, 0.331,
          0.645, 0.96, 1.409, 2.67, 2.834, 2.812, 2.483, 1.929,
          1.485, 1.214, 1.239, 1.608, 1.905, 2.023, 1.815, 0.535,
          0.122, 0.009, 0.164, 0.671, 1.019, 1.146, 1.155,
          1.112, 1.121, 1.223, 1.257, 1.157, 0.913, 0.62, 0.255,
```

```
-0.28, -1.08, -1.551, -1.799, -1.825, -1.456, -0.944,
-0.57, -0.431, -0.577, -0.96, -1.616, -1.875, -1.891,
-1.746, -1.474, -1.201, -0.927, -0.524, 0.04, 0.788, 0.943,
0.93, 1.006, 1.137, 1.198, 1.054, 0.595, -0.08, -0.314,
-0.288, -0.153, -0.109, -0.187, -0.255, -0.229, -0.007,
0.254, 0.33, 0.102, -0.423,
-1.139, -2.275, -2.594, -2.716, -2.51, -1.79, -1.346,
-1.081, -0.91, -0.876, -0.885, -0.8, -0.544, -0.416,
-0.271, 0.0, 0.403, 0.841, 1.285, 1.607, 1.746, 1.683,
1.485, 0.993, 0.648, 0.577, 0.577, 0.632, 0.747, 0.9,
0.993, 0.968, 0.79, 0.399, -0.161, -0.553, -0.603, -0.424,
-0.194, -0.049, 0.06, 0.161, 0.301, 0.517, 0.566, 0.56,
0.573, 0.592, 0.671, 0.933, 1.337, 1.46, 1.353, 0.772,
0.218,-0.237, -0.714, -1.099, -1.269, -1.175, -0.676,
0.033, 0.556, 0.643, 0.484, 0.109, -0.31, -0.697, -1.047,
-1.218, -1.183, -0.873, -0.336, 0.063, 0.084, 0.0, 0.001,
0.209, 0.556, 0.782, 0.858, 0.918, 0.862, 0.416, -0.336,
-0.959, -1.813, -2.378, -2.499, -2.473, -2.33, -2.053,
-1.739, -1.261, -0.569, -0.137, -0.024, -0.05, -0.135,
-0.276, -0.534, -0.871, -1.243, -1.439, -1.422, -1.175,
-0.813, -0.634, -0.582, -0.625, -0.713,
-0.848, -1.039, -1.346, -1.628, -1.619, -1.149,
-0.488, -0.16, -0.007, -0.092, -0.62, -1.086, -1.525,
-1.858, -2.029, -2.024, -1.961, -1.952, -1.794, -1.302,
-1.03, -0.918, -0.798, -0.867, -1.047, -1.123, -0.876,
-0.395, 0.185, 0.662, 0.709, 0.605, 0.501, 0.603, 0.943,
1.223, 1.249, 0.824, 0.102, 0.025, 0.382,
0.922, 1.032, 0.866, 0.527, 0.093, -0.458, -0.748,
-0.947, -1.029, -0.928, -0.645, -0.424, -0.276, -0.158,
-0.033, 0.102, 0.251, 0.28, 0.0, -0.493, -0.759, -0.824,
-0.74, -0.528, -0.204, 0.034, 0.204, 0.253, 0.195, 0.131,
0.017, -0.182, -0.262};
double[] y = {53.8, 53.6, 53.5, 53.5, 53.4, 53.1, 52.7, 52.4, 52.2,
52.0, 52.0, 52.4, 53.0, 54.0, 54.9, 56.0, 56.8, 56.8, 56.4,
55.7, 55.0, 54.3, 53.2, 52.3, 51.6, 51.2, 50.8, 50.5, 50.0,
49.2, 48.4, 47.9, 47.6, 47.5, 47.5, 47.6, 48.1, 49.0, 50.0,
51.1, 51.8, 51.9, 51.7, 51.2, 50.0, 48.3, 47.0, 45.8, 45.6,
46.0, 46.9, 47.8, 48.2, 48.3, 47.9, 47.2, 47.2,
48.1, 49.4, 50.6, 51.5, 51.6, 51.2, 50.5, 50.1, 49.8, 49.6,
49.4, 49.3, 49.2, 49.3, 49.7, 50.3, 51.3, 52.8, 54.4, 56.0,
56.9, 57.5, 57.3, 56.6, 56.0, 55.4, 55.4, 56.4, 57.2, 58.0,
58.4, 58.4, 58.1, 57.7, 57.0, 56.0, 54.7, 53.2, 52.1, 51.6,
```

```
  51.0, 50.5,50.4, 51.0, 51.8, 52.4, 53.0, 53.4, 53.6, 53.7,
  53.8, 53.8, 53.8, 53.3, 53.0, 52.9, 53.4, 54.6, 56.4, 58.0,
  59.4, 60.2, 60.0, 59.4, 58.4, 57.6, 56.9, 56.4, 56.0, 55.7,
  55.3, 55.0, 54.4, 53.7, 52.8, 51.6, 50.6, 49.4, 48.8, 48.5,
  48.7, 49.2, 49.8, 50.4, 50.7, 50.9, 50.7, 50.5, 50.4, 50.2,
  50.4, 51.2, 52.3, 53.2, 53.9, 54.1, 54.0, 53.6, 53.2, 53.0,
  52.8, 52.3,51.9, 51.6, 51.6, 51.4, 51.2, 50.7, 50.0, 49.4, 49.3,
  49.7, 50.6, 51.8, 53.0, 54.0, 55.3, 55.9, 55.9, 54.6, 53.5,
  52.4, 52.1, 52.3, 53.0, 53.8, 54.6, 55.4, 55.9, 55.9, 55.2,
  54.4, 53.7, 53.6, 53.6, 53.2, 52.5, 52.0, 51.4, 51.0, 50.9,
  52.4, 53.5, 55.6, 58.0, 59.5, 60.0, 60.4, 60.5, 60.2, 59.7,
  59.0, 57.6, 56.4, 55.2, 54.5, 54.1, 54.1, 54.4,
  55.5, 56.2, 57.0, 57.3, 57.4, 57.0, 56.4, 55.9, 55.5, 55.3,
  55.2, 55.4, 56.0, 56.5, 57.1, 57.3, 56.8, 55.6, 55.0, 54.1,
  54.3, 55.3, 56.4, 57.2, 57.8, 58.3, 58.6, 58.8, 58.8, 58.6,
  58.0, 57.4, 57.0, 56.4, 56.3, 56.4, 56.4, 56.0, 55.2, 54.0,
  53.0, 52.0,51.6, 51.6, 51.1, 50.4, 50.0, 50.0, 52.0, 54.0,
  55.1, 54.5, 52.8, 51.4, 50.8, 51.2, 52.0, 52.8, 53.8, 54.5,
  54.9, 54.9, 54.8, 54.4, 53.7, 53.3, 52.8, 52.6, 52.6, 53.0,
  54.3, 56.0, 57.0, 58.0, 58.6, 58.5, 58.3, 57.8, 57.3, 57.0};
  CrossCorrelation cc;

System.out.println("*****************************");
cc = new CrossCorrelation(x, y,10);
System.out.println("Mean = "+cc.getMeanX());
System.out.println("Mean = "+cc.getMeanY());
System.out.println("Xvariance  = "+cc.getVarianceX());
System.out.println("Yvariance  = "+cc.getVarianceY());
new PrintMatrix("CrossCovariances are:  ").print
               (cc.getCrossCovariance());
new PrintMatrix("CrossCorrelations are:  ").print
               (cc.getCrossCorrelation());
new PrintMatrix("Standard Errors using Bartlett are:  ").print
               (cc.getStandardErrors(cc.BARTLETTS_FORMULA));
new PrintMatrix("Standard Errors using Bartlett #2 are:  ").print
               (cc.getStandardErrors(cc.BARTLETTS_FORMULA_NOCC));
new PrintMatrix("AutoCovariances of X are:  ").print
               (cc.getAutoCovarianceX());
new PrintMatrix("AutoCovariances of Y are:  ").print
               (cc.getAutoCovarianceY());
new PrintMatrix("AutoCorrelations of X are:  ").print
               (cc.getAutoCorrelationX());
```

```
        new PrintMatrix("AutoCorrelations of Y are:  ").print
                    (cc.getAutoCorrelationY());
    }

}
```

### Output

```
*****************************
Mean = -0.05683445945945951
Mean = 53.50912162162156
Xvariance  = 1.1469379016503833
Yvariance  = 10.218937066289259
CrossCovariances are:
      0
 0  -0.405
 1  -0.508
 2  -0.614
 3  -0.705
 4  -0.776
 5  -0.831
 6  -0.891
 7  -0.981
 8  -1.125
 9  -1.347
10  -1.659
11  -2.049
12  -2.482
13  -2.885
14  -3.165
15  -3.253
16  -3.131
17  -2.839
18  -2.453
19  -2.053
20  -1.695

CrossCorrelations are:
      0
 0  -0.118
 1  -0.149
```

```
 2  -0.179
 3  -0.206
 4  -0.227
 5  -0.243
 6  -0.26
 7  -0.286
 8  -0.329
 9  -0.393
10  -0.484
11  -0.598
12  -0.725
13  -0.843
14  -0.925
15  -0.95
16  -0.915
17  -0.829
18  -0.717
19  -0.6
20  -0.495

Standard Errors using Bartlett are:
      0
 0  0.158
 1  0.156
 2  0.153
 3  0.149
 4  0.145
 5  0.141
 6  0.138
 7  0.136
 8  0.132
 9  0.124
10  0.108
11  0.087
12  0.064
13  0.047
14  0.044
15  0.048
16  0.049
17  0.048
18  0.053
19  0.072
```

```
20  0.094

Standard Errors using Bartlett #2 are:
      0
 0  0.163
 1  0.162
 2  0.162
 3  0.162
 4  0.162
 5  0.161
 6  0.161
 7  0.161
 8  0.161
 9  0.16
10  0.16
11  0.16
12  0.161
13  0.161
14  0.161
15  0.161
16  0.162
17  0.162
18  0.162
19  0.162
20  0.163

AutoCovariances of X are:
      0
 0  1.147
 1  1.092
 2  0.957
 3  0.782
 4  0.609
 5  0.467
 6  0.365
 7  0.298
 8  0.261
 9  0.244
10  0.239

AutoCovariances of Y are:
      0
```

```
 0  10.219
 1   9.92
 2   9.157
 3   8.099
 4   6.949
 5   5.871
 6   4.961
 7   4.252
 8   3.736
 9   3.376
10   3.132
```

```
AutoCorrelations of X are:
      0
 0  1
 1  0.952
 2  0.834
 3  0.682
 4  0.531
 5  0.408
 6  0.318
 7  0.26
 8  0.228
 9  0.213
10  0.208
```

```
AutoCorrelations of Y are:
      0
 0  1
 1  0.971
 2  0.896
 3  0.793
 4  0.68
 5  0.574
 6  0.485
 7  0.416
 8  0.366
 9  0.33
10  0.307
```

# *class* **MultiCrossCorrelation**

Computes the multichannel cross-correlation function of two mutually stationary multichannel time series.

`MultiCrossCorrelation` estimates the multichannel cross-correlation function of two mutually stationary multichannel time series. Define the multichannel time series $X$ by

$$X = (X_1, X_2, \ldots, X_p)$$

where

$$X_j = (X_{1j}, X_{2j}, \ldots, X_{nj})^T, \quad j = 1, 2, \ldots, p$$

with $n =$ `x.length` and $p =$ `x[0].length`. Similarly, define the multichannel time series $Y$ by

$$Y = (Y_1, Y_2, \ldots, Y_q)$$

where

$$Y_j = (Y_{1j}, Y_{2j}, \ldots, Y_{mj})^T, \quad j = 1, 2, \ldots, q$$

with $m =$ `y.length` and $q =$ `y[0].length`. The columns of $X$ and $Y$ correspond to individual channels of multichannel time series and may be examined from a univariate perspective. The rows of $X$ and $Y$ correspond to observations of $p$-variate and $q$-variate time series, respectively, and may be examined from a multivariate perspective. Note that an alternative characterization of a multivariate time series $X$ considers the columns to be observations of the multivariate time series while the rows contain univariate time series. For example, see Priestley (1981, page 692) and Fuller (1976, page 14).

Let $\hat{\mu}_X =$ `xmean` be the row vector containing the means of the channels of $X$. In particular,

$$\hat{\mu}_X = (\hat{\mu}_{X_1}, \hat{\mu}_{X_2}, \ldots, \hat{\mu}_{X_p})$$

where for $j = 1, 2, \ldots, p$

$$\hat{\mu}_{X_j} = \begin{cases} \mu_{X_j} & \text{for } \mu_{X_j} \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_{tj} & \text{for } \mu_{X_j} \text{ unknown} \end{cases}$$

Let $\hat{\mu}_Y =$ `ymean` be similarly defined. The cross-covariance of lag $k$ between channel $i$ of $X$ and channel $j$ of $Y$ is estimated by

$$\hat{\sigma}_{X_iY_j}(k) = \begin{cases} \frac{1}{N}\sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = 0, 1, \ldots, K \\ \frac{1}{N}\sum_t (X_{ti} - \hat{\mu}_{X_i})(Y_{t+k,j} - \hat{\mu}_{Y_j}) & k = -1, -2, \ldots, -K \end{cases}$$

where $i = 1, \ldots, p$, $j = 1, \ldots, q$, and $K =$ `maximum_lag`. The summation on $t$ extends over all possible cross-products with $N$ equal to the number of cross-products in the sum.

Let $\hat{\sigma}_X(0) =$ `xvar`, where `xvar` is the variance of $X$, be the row vector consisting of estimated variances of the channels of $X$. In particular,

$$\hat{\sigma}_X(0) = (\hat{\sigma}_{X_1}(0), \hat{\sigma}_{X_2}(0), \ldots, \hat{\sigma}_{X_p}(0))$$

where

$$\hat{\sigma}_{X_j}(0) = \frac{1}{n}\sum_{t=1}^{n}\left(X_{tj} - \hat{\mu}_{X_j}\right)^2, \qquad \text{j=0,1,\ldots,p}$$

Let $\hat{\sigma}_Y(0) =$ `yvar`, where `yvar` is the variance of $Y$, be similarly defined. The cross-correlation of lag $k$ between channel $i$ of $X$ and channel $j$ of $Y$ is estimated by

$$\hat{\rho}_{X_j Y_j}(k) = \frac{\hat{\sigma}_{X_j Y_j(k)}}{[\hat{\sigma}_{X_i}(0)\hat{\sigma}_{X_j}(0)]^{\frac{1}{2}}} \qquad k = 0, \pm 1, \ldots, \pm K$$

## Declaration

public class com.imsl.stat.MultiCrossCorrelation
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Class

### *class* **MultiCrossCorrelation.NonPosVariancesException**

The problem is ill-conditioned.

#### Declaration

public static class com.imsl.stat.MultiCrossCorrelation.NonPosVariancesException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *MultiCrossCorrelation.NonPosVariancesException*
  `public` **MultiCrossCorrelation.NonPosVariancesException**`(`
  `java.lang.String` **message** `)`
- *MultiCrossCorrelation.NonPosVariancesException*
  `public` **MultiCrossCorrelation.NonPosVariancesException**`(`
  `java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## Constructor

- *MultiCrossCorrelation*
  `public` **MultiCrossCorrelation**`( double[][] x, double[][] y, int maximum_lag )`
  - **Description**
    Constructor to compute the multichannel cross-correlation function of two mutually stationary multichannel time series.
  - **Parameters**
    * `x` – A two-dimensional `double` array containing the first multichannel stationary time series. Each row of `x` corresponds to an observation of a multivariate time series and each column of `x` corresponds to a univariate time series.
    * `y` – A two-dimensional `double` array containing the second multichannel stationary time series. Each row of `y` corresponds to an observation of a multivariate time series and each column of `y` corresponds to a univariate time series.
    * `maximum_lag` – An `int` containing the maximum lag of the cross-covariance and cross-correlations to be computed. `maximum_lag` must be greater than or equal to 1 and less than the minimum number of observations of `x` and `y`.

## Methods

- *getCrossCorrelation*
  `public double[][][]` **getCrossCorrelation**`( ) throws com.imsl.stat.MultiCrossCorrelation.NonPosVariancesException`
  - **Description**
    Returns the cross-correlations between the channels of `x` and `y`.
  - **Returns** – A `double` array of size 2 * `maximum_lag` +1 by `x[0].length` by `y[0].length` containing the cross-correlations between the time series `x` and `y`. The cross-correlation between channel $i$ of the `x` series and channel $j$ of the `y` series at lag $k$, where $k$ = `-maximum_lag`, ..., 0, 1, ..., `maximum_lag`, corresponds to output array element with index [k][i][j] where $k$= 0,1,...,`(2*maximum_lag)`, $i$ = 1, ..., `x[0].length`, and $j$ = 1, ..., `y[0].length`.

- *getCrossCovariance*
  `public double[][][]` **getCrossCovariance**`( ) throws com.imsl.stat.MultiCrossCorrelation.NonPosVariancesException`

- **Description**
  Returns the cross-covariances between the channels of x and y.
- **Returns** – A double array of size 2 * maximum_lag +1 by x[0].length by y[0].length containing the cross-covariances between the time series x and y. The cross-covariances between channel $i$ of the x series and channel $j$ of the y series at lag $k$ where $k$ = -maximum_lag, ..., 0, 1, ..., maximum_lag, corresponds to output array element with index [k][i][j] where $k$= 0,1,...,(2*maximum_lag), $i$ = 1, ..., x[0].length, and $j$ = 1, ..., y[0].length.

---

- *getMeanX*
  `public double[] getMeanX( )`

  - **Description**
    Returns the mean of each channel of x.
  - **Returns** – A one-dimensional double containing the mean of each channel in the time series x.

---

- *getMeanY*
  `public double[] getMeanY( )`

  - **Description**
    Returns the mean of each channel of y.
  - **Returns** – A one-dimensional double containing the estimate mean of each channel in the time series y.

---

- *getVarianceX*
  `public double[] getVarianceX( ) throws com.imsl.stat.MultiCrossCorrelation.NonPosVariancesException`

  - **Description**
    Returns the variances of the channels of x.
  - **Returns** – A one-dimensional double containing the variances of each channel in the time series x.

---

- *getVarianceY*
  `public double[] getVarianceY( ) throws com.imsl.stat.MultiCrossCorrelation.NonPosVariancesException`

  - **Description**
    Returns the variances of the channels of y.
  - **Returns** – A one-dimensional double containing the variances of each channel in the time series y.

---

- *setMeanX*
  `public void` **setMeanX**`( double[]` **mean** `)`
  - **Description**
    Estimate of the mean of each channel of `x`.
  - **Parameters**
    * `mean` – A one-dimensional `double` containing the estimate of the mean of each channel in time series `x`.

---

- *setMeanY*
  `public void` **setMeanY**`( double[]` **mean** `)`
  - **Description**
    Estimate of the mean of each channel of `y`.
  - **Parameters**
    * `mean` – A one-dimensional `double` containing the estimate of the mean of each channel in the time series `y`.

### Example 1: MultiCrossCorrelation

Consider the Wolfer Sunspot Data (Y ) (Box and Jenkins 1976, page 530) along with data on northern light activity (X1) and earthquake activity (X2) (Robinson 1967, page 204) to be a three-channel time series. Methods `getCrossCovariance` and `getCrossCorrelation` are used to compute the cross-covariances and cross-correlations between $X_1$ and Y and between $X_2$ and Y with lags from `-maximum_lag` = -10 through lag `maximum_lag` = 10.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.Matrix;

public class MultiCrossCorrelationEx1 {

    public static void main(String args[]) throws Exception {
        int i;
        double x[][] = {{  155.0,   66.0},
        {  113.0,   62.0},
        {    3.0,   66.0},
        {   10.0,  197.0},
        {    0.0,   63.0},
        {    0.0,    0.0},
        {   12.0,  121.0},
        {   86.0,    0.0},
```

```
{  102.0,  113.0},
{   20.0,   27.0},
{   98.0,  107.0},
{  116.0,   50.0},
{   87.0,  122.0},
{  131.0,  127.0},
{  168.0,  152.0},
{  173.0,  216.0},
{  238.0,  171.0},
{  146.0,   70.0},
{    0.0,  141.0},
{    0.0,   69.0},
{    0.0,  160.0},
{    0.0,   92.0},
{   12.0,   70.0},
{    0.0,   46.0},
{   37.0,   96.0},
{   14.0,   78.0},
{   11.0,  110.0},
{   28.0,   79.0},
{   19.0,   85.0},
{   30.0,  113.0},
{   11.0,   59.0},
{   26.0,   86.0},
{    0.0,  199.0},
{   29.0,   53.0},
{   47.0,   81.0},
{   36.0,   81.0},
{   35.0,  156.0},
{   17.0,   27.0},
{    0.0,   81.0},
{    3.0,  107.0},
{    6.0,  152.0},
{   18.0,   99.0},
{   15.0,  177.0},
{    0.0,   48.0},
{    3.0,   70.0},
{    9.0,  158.0},
{   64.0,   22.0},
{  126.0,   43.0},
{   38.0,  102.0},
{   33.0,  111.0},
```

```
{   71.0,   90.0},
{   24.0,   86.0},
{   20.0,  119.0},
{   22.0,   82.0},
{   13.0,   79.0},
{   35.0,  111.0},
{   84.0,   60.0},
{  119.0,  118.0},
{   86.0,  206.0},
{   71.0,  122.0},
{  115.0,  134.0},
{   91.0,  131.0},
{   43.0,   84.0},
{   67.0,  100.0},
{   60.0,   99.0},
{   49.0,   99.0},
{  100.0,   69.0},
{  150.0,   67.0},
{  178.0,   26.0},
{  187.0,  106.0},
{   76.0,  108.0},
{   75.0,  155.0},
{  100.0,   40.0},
{   68.0,   75.0},
{   93.0,   99.0},
{   20.0,   86.0},
{   51.0,  127.0},
{   72.0,  201.0},
{  118.0,   76.0},
{  146.0,   64.0},
{  101.0,   31.0},
{   61.0,  138.0},
{   87.0,  163.0},
{   53.0,   98.0},
{   69.0,   70.0},
{   46.0,  155.0},
{   47.0,   97.0},
{   35.0,   82.0},
{   74.0,   90.0},
{  104.0,  122.0},
{   97.0,   70.0},
{  106.0,   96.0},
```

```
{  113.0,  111.0},
{  103.0,   42.0},
{   68.0,   97.0},
{   67.0,   91.0},
{   82.0,   64.0},
{   89.0,   81.0},
{  102.0,  162.0},
{  110.0,  137.0}};

double y[][] = {{ 101.0},
{  82.0},
{  66.0},
{  35.0},
{  31.0},
{   7.0},
{  20.0},
{  92.0},
{ 154.0},
{ 126.0},
{  85.0},
{  68.0},
{  38.0},
{  23.0},
{  10.0},
{  24.0},
{  83.0},
{ 132.0},
{ 131.0},
{ 118.0},
{  90.0},
{  67.0},
{  60.0},
{  47.0},
{  41.0},
{  21.0},
{  16.0},
{   6.0},
{   4.0},
{   7.0},
{  14.0},
{  34.0},
{  45.0},
```

```
{  43.0},
{  48.0},
{  42.0},
{  28.0},
{  10.0},
{   8.0},
{   2.0},
{   0.0},
{   1.0},
{   5.0},
{  12.0},
{  14.0},
{  35.0},
{  46.0},
{  41.0},
{  30.0},
{  24.0},
{  16.0},
{   7.0},
{   4.0},
{   2.0},
{   8.0},
{  17.0},
{  36.0},
{  50.0},
{  62.0},
{  67.0},
{  71.0},
{  48.0},
{  28.0},
{   8.0},
{  13.0},
{  57.0},
{ 122.0},
{ 138.0},
{ 103.0},
{  86.0},
{  63.0},
{  37.0},
{  24.0},
{  11.0},
{  15.0},
```

```
{  40.0},
{  62.0},
{  98.0},
{ 124.0},
{  96.0},
{  66.0},
{  64.0},
{  54.0},
{  39.0},
{  21.0},
{   7.0},
{   4.0},
{  23.0},
{  55.0},
{  94.0},
{  96.0},
{  77.0},
{  59.0},
{  44.0},
{  47.0},
{  30.0},
{  16.0},
{   7.0},
{  37.0},
{  74.0}};



MultiCrossCorrelation mcc = new MultiCrossCorrelation(x, y, 10);

new PrintMatrix("Mean of X :  ").print(mcc.getMeanX());
new PrintMatrix("Variance of X :  ").print(mcc.getVarianceX());
new PrintMatrix("Mean of Y :  ").print(mcc.getMeanY());
new PrintMatrix("Variance of Y :  ").print(mcc.getVarianceY());
double[][][] ccv = new double[21][2][1];
double[][][] cc = new double[21][2][1];

ccv = mcc.getCrossCovariance();
System.out.println("Multichannel cross-covariance between X and Y");
for (i=0; i<21; i++) {
    System.out.println("Lag K = "+(i-10));
    new PrintMatrix("CrossCovariances : ").print(ccv[i]);
```

```
        }
        cc = mcc.getCrossCorrelation();
        System.out.println("Multichannel cross-correlation between X and Y");
        for (i=0; i<21; i++) {
            System.out.println("Lag K = "+(i-10));
            new PrintMatrix("CrossCorrelations : ").print(cc[i]);
        }
    }

}
```

## Output

```
Mean of X :
     0
0  63.43
1  97.97

Variance of X :
       0
0  2,643.685
1  1,978.429

Mean of Y :
     0
0  46.94

Variance of Y :
       0
0  1,383.756

Multichannel cross-covariance between X and Y
Lag K = -10
CrossCovariances :
      0
0  -20.512
1   70.713

Lag K = -9
CrossCovariances :
     0
```

```
0  65.024
1  38.136

Lag K = -8
CrossCovariances :
      0
0  216.637
1  135.578

Lag K = -7
CrossCovariances :
      0
0  246.794
1  100.362

Lag K = -6
CrossCovariances :
      0
0  142.128
1   44.968

Lag K = -5
CrossCovariances :
      0
0   50.697
1  -11.809

Lag K = -4
CrossCovariances :
     0
0  72.685
1  32.693

Lag K = -3
CrossCovariances :
      0
0  217.854
1  -40.119

Lag K = -2
CrossCovariances :
      0
```

```
0   355.821
1  -152.649

Lag K = -1
CrossCovariances :
      0
0   579.653
1  -212.95

Lag K = 0
CrossCovariances :
      0
0   821.626
1  -104.752

Lag K = 1
CrossCovariances :
      0
0  810.131
1   55.16

Lag K = 2
CrossCovariances :
      0
0  628.385
1   84.775

Lag K = 3
CrossCovariances :
      0
0  438.272
1   75.963

Lag K = 4
CrossCovariances :
      0
0  238.793
1  200.383

Lag K = 5
CrossCovariances :
      0
```

```
0  143.621
1  282.986

Lag K = 6
CrossCovariances :
      0
0  252.974
1  234.393

Lag K = 7
CrossCovariances :
      0
0  479.468
1  223.034

Lag K = 8
CrossCovariances :
      0
0  724.912
1  124.457

Lag K = 9
CrossCovariances :
      0
0  924.971
1  -79.517

Lag K = 10
CrossCovariances :
      0
0   922.759
1  -279.286

Multichannel cross-correlation between X and Y
Lag K = -10
CrossCorrelations :
     0
0  -0.011
1   0.043

Lag K = -9
CrossCorrelations :
```

```
     0
0  0.034
1  0.023

Lag K = -8
CrossCorrelations :
     0
0  0.113
1  0.082

Lag K = -7
CrossCorrelations :
     0
0  0.129
1  0.061

Lag K = -6
CrossCorrelations :
     0
0  0.074
1  0.027

Lag K = -5
CrossCorrelations :
     0
0   0.027
1  -0.007

Lag K = -4
CrossCorrelations :
     0
0  0.038
1  0.02

Lag K = -3
CrossCorrelations :
     0
0   0.114
1  -0.024

Lag K = -2
CrossCorrelations :
```

```
     0
0   0.186
1  -0.092

Lag K = -1
CrossCorrelations :
     0
0   0.303
1  -0.129

Lag K = 0
CrossCorrelations :
     0
0   0.43
1  -0.063

Lag K = 1
CrossCorrelations :
     0
0  0.424
1  0.033

Lag K = 2
CrossCorrelations :
     0
0  0.329
1  0.051

Lag K = 3
CrossCorrelations :
     0
0  0.229
1  0.046

Lag K = 4
CrossCorrelations :
     0
0  0.125
1  0.121

Lag K = 5
CrossCorrelations :
```

```
     0
0  0.075
1  0.171

Lag K = 6
CrossCorrelations :
     0
0  0.132
1  0.142

Lag K = 7
CrossCorrelations :
     0
0  0.251
1  0.135

Lag K = 8
CrossCorrelations :
     0
0  0.379
1  0.075

Lag K = 9
CrossCorrelations :
     0
0   0.484
1  -0.048

Lag K = 10
CrossCorrelations :
     0
0   0.482
1  -0.169
```

## *class* **ARMA**

Computes least-square estimates of parameters for an ARMA model.

Class `ARMA` computes estimates of parameters for a nonseasonal ARMA model given a sample of observations, $\{W_t\}$, for $t = 1, 2, \ldots, n$, where $n =$ `z.length`.

Two methods of parameter estimation, method of moments and least squares, are provided. The user can choose a method using the `setMethod` method. If the user wishes to use the least-squares algorithm, the preliminary estimates are the method of moments estimates by default. Otherwise, the user can input initial estimates by using the `setInitialEstimates` method. The following table lists the appropriate methods for both the method of moments and least-squares algorithm:

| **Least Squares** | **Both Method of Moment and Least Squares** |
|---|---|
| | `setCenter` |
| `setARLags` | `setMethod` |
| `setMALags` | `setRelativeError` |
| `setBackcasting` | `setMaxIterations` |
| `setConvergenceTolerance` | `setMeanEstimate` |
| `setInitialEstimates` | `getMeanEstimate` |
| `getResidual` | `getAutocovariance` |
| `getSSResidual` | `getVariance` |
| `getParamEstimatesCovariance` | `getConstant` |
| | `getAR` |
| | `getMA` |

### Method of Moments Estimation

Suppose the time series $\{Z_t\}$ is generated by an ARMA $(p,\ q)$ model of the form

$$\phi(B)Z_t = \theta_0 + \theta(B)A_t$$

$$\text{for } \ t \in \{0, \pm 1, \pm 2, \ldots\}$$

Let $\hat{\mu} =$ zMean be the estimate of the mean $\mu$ of the time series $\{Z_t\}$, where $\hat{\mu}$ equals the following:

$$\hat{\mu} = \begin{cases} \mu & \text{for } \mu \text{ known} \\ \frac{1}{n}\sum\limits_{t=1}^{n} Z_t & \text{for } \mu \text{ unknown} \end{cases}$$

The autocovariance function is estimated by

$$\hat{\sigma}\left(k\right) = \frac{1}{n}\sum_{t=1}^{n-k}\left(Z_t - \hat{\mu}\right)\left(Z_{t+k} - \hat{\mu}\right)$$

for $k = 0, 1, \ldots, K$, where $K = p + q$. Note that $\hat{\sigma}(0)$ is an estimate of the sample variance.

Given the sample autocovariances, the function computes the method of moments estimates of the autoregressive parameters using the extended Yule-Walker equations as follows:

$$\hat{\Sigma}\hat{\phi} = \hat{\sigma}$$

where

$$\hat{\phi} = \left(\hat{\phi}_1,\ \ldots,\ \hat{\phi}_p\right)^T$$

$$\hat{\Sigma}_{ij} = \hat{\sigma}\left(|q + i - j|\right),\ \ i, j = 1,\ \ldots,\ p$$

$$\hat{\sigma}_i = \hat{\sigma}\left(q + i\right),\ \ i = 1,\ \ldots,\ p$$

The overall constant $\theta_0$ is estimated by the following:

$$\hat{\theta}_0 = \begin{cases} \hat{\mu} \ \text{ for } p = 0 \\ \hat{\mu}\left(1 - \sum\limits_{i=1}^{p} \hat{\phi}_i\right) \ \text{ for } p > 0 \end{cases}$$

The moving average parameters are estimated based on a system of nonlinear equations given $K = p + q + 1$ autocovariances, $\sigma(k)$ for $k = 1, \ldots, K$, and $p$ autoregressive parameters $\phi_i$ for $i = 1, \ldots, p$.

Let $Z'_t = \phi(B)Z_t$. The autocovariances of the derived moving average process $Z'_t = \theta(B)A_t$ are estimated by the following relation:

$$\hat{\sigma}'\left(k\right) = \begin{cases} \hat{\sigma}\left(k\right) \ \text{ for } p = 0 \\ \sum\limits_{i=0}^{p}\sum\limits_{j=0}^{p} \hat{\phi}_i\hat{\phi}_j\left(\hat{\sigma}\left(|k + i - j|\right)\right) \ \text{ for } \ p \geq 1, \hat{\phi}_0 \equiv -1 \end{cases}$$

The iterative procedure for determining the moving average parameters is based on the relation

$$\sigma\left(k\right) = \begin{cases} \left(1 + \theta_1^2 + \ \ldots \ + \theta_q^2\right)\sigma_A^2 \ \text{ for } \ k = 0 \\ \left(-\theta_k + \theta_1\theta_{k+1} + \ \ldots \ + \theta_{q-k}\theta_q\right)\sigma_A^2 \ \text{ for } \ k \geq 1 \end{cases}$$

where $\sigma(k)$ denotes the autocovariance function of the original $Z_t$ process.

Let $\tau = (\tau_0, \tau_1, \ldots, \tau_q)^T$ and $f = (f_0, f_1, \ldots, f_q)^T$, where

$$\tau_j = \begin{cases} \sigma_A & \text{for } j = 0 \\ -\theta_j/\tau_0 & \text{for } j = 1, \ldots, q \end{cases}$$

and

$$f_j = \sum_{i=0}^{q-j} \tau_i \tau_{i+j} - \hat{\sigma}'(j) \quad \text{for } j = 0, 1, \ldots, q$$

Then, the value of $\tau$ at the (i + 1)-th iteration is determined by the following:

$$\tau^{i+1} = \tau^i - \left(T^i\right)^{-1} f^i$$

The estimation procedure begins with the initial value

$$\tau^0 = (\sqrt{\hat{\sigma}'(0)}, \quad 0, \ldots, 0)^T$$

and terminates at iteration $i$ when either $\left\|f^i\right\|$ is less than `relativeError` or $i$ equals `iterations`. The moving average parameter estimates are obtained from the final estimate of $\tau$ by setting

$$\hat{\theta}_j = -\tau_j/\tau_0 \quad \text{for } j = 1, \ldots, q$$

The random shock variance is estimated by the following:

$$\hat{\sigma}_A^2 = \begin{cases} \hat{\sigma}(0) - \sum_{i=1}^{p} \hat{\phi}_i \hat{\sigma}(i) & \text{for } q = 0 \\ \tau_0^2 & \text{for } q \geq 0 \end{cases}$$

See Box and Jenkins (1976, pp. 498-500) for a description of a function that performs similar computations.

**Least-squares Estimation**

Suppose the time series $\{Z_t\}$ is generated by a nonseasonal ARMA model of the form,

$$\phi(B)(Z_t - \mu) = \theta(B)A_t \quad \text{for } t \in \{0, \pm 1, \pm 2, \ldots\}$$

where $B$ is the backward shift operator, $\mu$ is the mean of $Z_t$, and

$$\phi(B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ \dots\ - \phi_p B^{l_\phi(p)} \qquad \text{for } p \geq 0$$

$$\theta(B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ \dots\ - \theta_q B^{l_\theta(q)} \qquad \text{for } q \geq 0$$

with $p$ autoregressive and $q$ moving average parameters. Without loss of generality, the following is assumed:

$$1 \leq l_\phi(1) \leq l_\phi(2) \leq \ldots \leq l_\phi(p)$$

$$1 \leq l_\theta(1) \leq l_\theta(2) \leq \ldots \leq l_\theta(q)$$

so that the nonseasonal ARMA model is of order $(p', q')$, where $p' = l_\theta(p)$ and $q' = l_\theta(q)$. Note that the usual hierarchical model assumes the following:

$$l_\phi(i) = i, 1 \leq i \leq p$$

$$l_\theta(j) = j, 1 \leq j \leq q$$

Consider the sum-of-squares function

$$S_T(\mu, \phi, \theta) = \sum_{-T+1}^{n} [A_t]^2$$

where

$$[A_t] = E[A_t \,|(\mu, \phi, \theta, Z)]$$

and $T$ is the backward origin. The random shocks $\{A_t\}$ are assumed to be independent and identically distributed

$$N\left(0, \sigma_A^2\right)$$

random variables. Hence, the log-likelihood function is given by

$$l(\mu, \phi, \theta, \sigma_A) = f(\mu, \phi, \theta) - n \ln(\sigma_A) - \frac{S_T(\mu, \phi, \theta)}{2\sigma_A^2}$$

where $f(\mu, \phi, \theta)$ is a function of $\mu, \phi,$ and $\theta$.

For $T = 0$, the log-likelihood function is conditional on the past values of both $Z_t$ and $A_t$ required to initialize the model. The method of selecting these initial values usually introduces transient bias into the model (Box and Jenkins 1976, pp. 210-211). For $T = \infty$, this dependency vanishes, and estimation problem concerns maximization of the unconditional log-likelihood function. Box and Jenkins (1976, p. 213) argue that

$$S_\infty (\mu, \phi, \theta) / \left(2\sigma_A^2\right)$$

dominates

$$l \left(\mu, \phi, \theta, \sigma_A^2\right)$$

The parameter estimates that minimize the sum-of-squares function are called least-squares estimates. For large $n$, the unconditional least-squares estimates are approximately equal to the maximum likelihood-estimates.

In practice, a finite value of $T$ will enable sufficient approximation of the unconditional sum-of-squares function. The values of $[A_T]$ needed to compute the unconditional sum of squares are computed iteratively with initial values of $Z_t$ obtained by back forecasting. The residuals (including backcasts), estimate of random shock variance, and covariance matrix of the final parameter estimates also are computed. ARIMA parameters can be computed by using `Difference` with `ARMA`.

**Forecasting**

The Box-Jenkins forecasts and their associated probability limits for a nonseasonal ARMA model are computed given a sample of $n =$ `z.length`, $\{Z_t\}$ for $t = 1, 2, \ldots, n$.

Suppose the time series $Z_t$ is generated by a nonseasonal ARMA model of the form

$$\phi(B)Z_t = \theta_0 + \theta(B)A_t$$

for $t \in \{0, \pm 1, \pm 2, \ldots\}$, where $B$ is the backward shift operator, $\theta_0$ is the constant, and

$$\phi (B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ \ldots \ - \phi_p B^{l_\phi(p)}$$

$$\theta (B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ \ldots \ - \theta_q B^{l_\theta(q)}$$

with $p$ autoregressive and $q$ moving average parameters. Without loss of generality, the following is assumed:

$$1 \le l_\phi(1) \le l_\phi(2) \le \ldots l_\phi(p)$$

$$1 \le l_\theta(1) \le l_\theta(2) \le \ldots \le l_\theta(q)$$

so that the nonseasonal ARMA model is of order $(p', q')$, where $p' = l_\theta(p)$ and $q' = l_\theta(q)$. Note that the usual hierarchical model assumes the following:

$$l_\phi(i) = i, 1 \le i \le p$$

$$l_\theta(j) = j, 1 \le j \le q$$

The Box-Jenkins forecast at origin $t$ for lead time l of $Z_{t+1}$ is defined in terms of the difference equation

$$\hat{Z}_t\left(l\right) = \theta_0 + \phi_1\left[Z_{t+l-l_\phi(1)}\right] + \ \ldots\ + \phi_p\left[Z_{t+l-l_\phi(p)}\right]$$

$$+\left[A_{t+l}\right] - \theta_1\left[A_{t+l-l_\theta(1)}\right] \ - \ ... \ - \ \left[A_{t+l}\right] - \theta_1\left[A_{t+l-l\theta(1)}\right] - ... - \theta_q\left[A_{t+l-l_\theta(q)}\right]$$

where the following is true:

$$\left[Z_{t+k}\right] = \begin{cases} Z_{t+k} \ \text{ for } \ k = 0, \ -1, \ -2, \ \ldots \\ \hat{Z}_t\left(k\right) \ \text{ for } \ k = 1, \ 2, \ \ldots \end{cases}$$

$$\left[A_{t+k}\right] = \begin{cases} Z_{t+k} - \hat{Z}_{t+k-1}\left(1\right) \ \text{ for } \ k = 0, \ -1, \ -2, \ ... \\ 0 \ \text{ for } \ k = 1, \ 2, \ ... \end{cases}$$

The $100(1 - \alpha)$ percent probability limits for $Z_{t+l}$ are given by

$$\hat{Z}_t\left(l\right) \pm z_{1/2}\left\{1 + \sum_{j=1}^{l-1}\psi_j^2\right\}^{1/2} \sigma_A$$

where $z_{(1-\alpha/2)}$ is the $100(1 - \alpha/2)$ percentile of the standard normal distribution

$$\sigma_A^2$$

and

$$\{\psi_j^2\}$$

are the parameters of the random shock form of the difference equation. Note that the forecasts are computed for lead times $l = 1, 2, \ldots, L$ at origins $t = (n - b), (n - b + 1), \ldots, n$, where $L =$ nPredict and $b =$ backwardOrigin.

The Box-Jenkins forecasts minimize the mean-square error

$$E\left[Z_{t+l} - \hat{Z}_t\left(l\right)\right]^2$$

Also, the forecasts can be easily updated according to the following equation:

$$\hat{Z}_{t+1}\left(l\right) = \hat{Z}_t\left(l+1\right) + \psi_l A_{t+1}$$

This approach and others are discussed in Chapter 5 of Box and Jenkins (1976).

## Declaration

public class com.imsl.stat.ARMA
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* ARMA.TooManyCallsException

The number of calls to the function has exceeded the maximum number of iterations.

#### Declaration

public static class com.imsl.stat.ARMA.TooManyCallsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *ARMA.TooManyCallsException*
  `public` **ARMA.TooManyCallsException**`( java.lang.String` **message** `)`

- **Description**
  Constructs an `TooManyCallsException` with the specified detail message. A detail message is a `String` that describes this particular exception.
- **Parameters**
  * `message` – the detail message

---

- *ARMA.TooManyCallsException*
  `public` **ARMA.TooManyCallsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an `TooManyCallsException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    * `key` – the key of the error message in the resource bundle
    * `arguments` – an array containing arguments used within the error message string

## *class* ARMA.IncreaseErrRelException

The bound for the relative error is too small.

### Declaration

public static class com.imsl.stat.ARMA.IncreaseErrRelException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ARMA.IncreaseErrRelException*
  `public` **ARMA.IncreaseErrRelException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `IncreaseErrRelException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  - **Parameters**
    * `message` – the detail message

---

- *ARMA.IncreaseErrRelException*
  `public` **ARMA.IncreaseErrRelException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an `IncreaseErrRelException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    * `key` – the key of the error message in the resource bundle
    * `arguments` – an array containing arguments used within the error message string

## *class* ARMA.NewInitialGuessException

The iteration has not made good progress.

### Declaration

public static class com.imsl.stat.ARMA.NewInitialGuessException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ARMA.NewInitialGuessException*
  `public` **ARMA.NewInitialGuessException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `NewInitialGuessException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  - **Parameters**
    * `message` – the detail message

- *ARMA.NewInitialGuessException*
  `public` **ARMA.NewInitialGuessException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an `NewInitialGuessException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**

* `key` – the key of the error message in the resource bundle
* `arguments` – an array containing arguments used within the error message string

## *class* **ARMA.MatrixSingularException**

The input matrix is singular.

### Declaration

public static class com.imsl.stat.ARMA.MatrixSingularException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ARMA.MatrixSingularException*
  `public` **ARMA.MatrixSingularException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `MatrixSingularException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  - **Parameters**
    * `message` – the detail message

---

- *ARMA.MatrixSingularException*
  `public` **ARMA.MatrixSingularException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an `MatrixSingularException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    * `key` – the key of the error message in the resource bundle
    * `arguments` – an array containing arguments used within the error message string

## *class* **ARMA.TooManyITNException**

Maximum number of iterations exceeded.

**Declaration**

public static class com.imsl.stat.ARMA.TooManyITNException
**extends** com.imsl.IMSLException (page 1240)

**Constructors**

---

- *ARMA.TooManyITNException*
  `public` **ARMA.TooManyITNException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `TooManyITNException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  - **Parameters**
    - `message` – the detail message

- *ARMA.TooManyITNException*
  `public` **ARMA.TooManyITNException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* ARMA.TooManyFcnEvalException

Maximum number of function evaluations exceeded.

**Declaration**

public static class com.imsl.stat.ARMA.TooManyFcnEvalException
**extends** com.imsl.IMSLException (page 1240)

**Constructors**

---

- *ARMA.TooManyFcnEvalException*
  `public` **ARMA.TooManyFcnEvalException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `TooManyFcnEvalException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  - **Parameters**

* `message` – the detail message

---

- *ARMA.TooManyFcnEvalException*
  `public` **ARMA.TooManyFcnEvalException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an `TooManyFcnEvalException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    * `key` – the key of the error message in the resource bundle
    * `arguments` – an array containing arguments used within the error message string

## *class* ARMA.TooManyJacobianEvalException

Maximum number of Jacobian evaluations exceeded.

### Declaration

public static class com.imsl.stat.ARMA.TooManyJacobianEvalException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ARMA.TooManyJacobianEvalException*
  `public` **ARMA.TooManyJacobianEvalException**`( java.lang.String` **message** `)`
  - **Description**
    Constructs an `TooManyJacobianEvalException` with the specified detail message. A detail message is a `String` that describes this particular exception.
  - **Parameters**
    * `message` – the detail message

---

- *ARMA.TooManyJacobianEvalException*
  `public` **ARMA.TooManyJacobianEvalException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

– **Description**
Constructs an `TooManyJacobianEvalException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

– **Parameters**
  * `key` – the key of the error message in the resource bundle
  * `arguments` – an array containing arguments used within the error message string

## *class* ARMA.IllConditionedException

The problem is ill-conditioned.

### Declaration

public static class com.imsl.stat.ARMA.IllConditionedException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ARMA.IllConditionedException*
`public` **ARMA.IllConditionedException**`( java.lang.String` **message** `)`

  – **Description**
  Constructs an `IllConditionedException` with the specified detail message. A detail message is a `String` that describes this particular exception.

  – **Parameters**
    * `message` – the detail message

- *ARMA.IllConditionedException*
`public` **ARMA.IllConditionedException**`( java.lang.String` **key**`,`
`java.lang.Object[]` **arguments** `)`

  – **Description**
  Constructs an `IllConditionedException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

  – **Parameters**
    * `key` – the key of the error message in the resource bundle
    * `arguments` – an array containing arguments used within the error message string

## Fields

- public static final `int` **METHOD_OF_MOMENTS**
  - Indicates autoregressive and moving average parameters are estimated by a method of moments procedure.
- public static final `int` **LEAST_SQUARES**
  - Indicates autoregressive and moving average parameters are estimated by a least-squares procedure.

## Constructor

- *ARMA*
  `public` **ARMA**`( int` **p**`, int` **q**`, double[]` **z** `)`
  - **Description**
    Constructor for `ARMA`.
  - **Parameters**
    * `p` – an `int` scalar containing the number of autoregressive (AR) parameters
    * `q` – an `int` scalar containing the number of moving average (MA) parameters
    * `z` – a `double` array containing the observations
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `p`, `q`, and `z.length` are not consistent.

## Methods

- *compute*
  `public final void` **compute**`( ) throws`
  `com.imsl.stat.ARMA.MatrixSingularException,`
  `com.imsl.stat.ARMA.TooManyCallsException,`
  `com.imsl.stat.ARMA.IncreaseErrRelException,`
  `com.imsl.stat.ARMA.NewInitialGuessException,`
  `com.imsl.stat.ARMA.IllConditionedException,`
  `com.imsl.stat.ARMA.TooManyITNException,`

`com.imsl.stat.ARMA.TooManyFcnEvalException`,
`com.imsl.stat.ARMA.TooManyJacobianEvalException`

- **Description**
  Computes least-square estimates of parameters for an ARMA model.
- **Throws**
  - `com.imsl.stat.ARMA.MatrixSingularException` – is thrown if the input matrix is singular
  - `com.imsl.stat.ARMA.TooManyCallsException` – is thrown if the number of calls to the function has exceeded
  - `com.imsl.stat.ARMA.IncreaseErrRelException` – is thrown if the bound for the relative error is too small
  - `com.imsl.stat.ARMA.NewInitialGuessException` – is thrown if the iteration has not made good progress
  - `com.imsl.stat.ARMA.IllConditionedException` – is thrown if the problem is ill-conditioned
  - `com.imsl.stat.ARMA.TooManyITNException` – is thrown if the maximum number of iterations exceeded
  - `com.imsl.stat.ARMA.TooManyFcnEvalException` – is thrown if the maximum number of function evaluations exceeded
  - `com.imsl.stat.ARMA.TooManyJacobianEvalException` – is thrown if the maximum number of Jacobian evaluations exceeded

---

- *forecast*
  `public final double[][]` **`forecast`**`( int` **`nPredict`** `)`

  - **Description**
    Computes forecasts and their associated probability limits for an ARMA model.
  - **Parameters**
    - `nPredict` – an `int` scalar containing the maximum lead time for forecasts. `nPredict` must be greater than 0.
  - **Returns** – a `double` matrix of dimensions of `nPredict` by `backwardOrigin + 1` containing the forecasts. Return `NULL` if the least-square estimates of parameters is not computed.

---

- *getAR*
  `public double[]` **`getAR`**`( )`

  - **Description**
    Returns the final autoregressive parameter estimates.
  - **Returns** – a `double` array of length `p` containing the final autoregressive parameter estimates

---

- *getAutoCovariance*
  `public double[]` **getAutoCovariance**`( )`
  - **Description**
    Returns the autocovariances of the time series `z`.
  - **Returns** – a `double` array containing the autocovariances of lag `k`, where `k = 1, ..., p + q + 1`

---

- *getConstant*
  `public double` **getConstant**`( )`
  - **Description**
    Returns the constant parameter estimate.
  - **Returns** – a `double` scalar containing the constant parameter estimate

---

- *getDeviations*
  `public double[]` **getDeviations**`( )`
  - **Description**
    Returns the deviations from each forecast that give the `confidence` percent probability limits.
  - **Returns** – a `double` array of length `nPredict` containing the deviations from each forecast that give the `confidence` percent probability limits

---

- *getMA*
  `public double[]` **getMA**`( )`
  - **Description**
    Returns the final moving average parameter estimates.
  - **Returns** – a `double` array of length `q` containing the final moving average parameter estimates

---

- *getMeanEstimate*
  `public double` **getMeanEstimate**`( )`
  - **Description**
    Returns an update of the mean of the time series `z`.
  - **Returns** – a `double` scalar containing an update of the mean of the time series `z`. If the time series is not centered about its mean, and least-squares algorithm is used, `zMean` is not used in parameter estimation.

---

- *getParamEstimatesCovariance*
  `public double[][]` **getParamEstimatesCovariance**`( )`

– **Description**
Returns the covariances of parameter estimates.

– **Returns** – a `double` matrix of dimensions of `np` by `np`, where `np = p + q + 1` if `z` is centered about `zMean`, and `np = p + q` if `z` is not centered, containing the covariances of parameter estimates. The ordering of variables is `zMean`, `ar`, and `ma`.

---

- *getPsiWeights*
`public double[]` **getPsiWeights**`( )`

  – **Description**
  Returns the psi weights of the infinite order moving average form of the model.

  – **Returns** – a `double` array of length `nPredict` containing the psi weights of the infinite order moving average form of the model.

---

- *getResidual*
`public double[]` **getResidual**`( )`

  – **Description**
  Returns the residuals.

  – **Returns** – a `double` array of length `z.length - Math.max(arLags[i]) + length` containing the residuals (including backcasts) at the final parameter estimate point in the first `z.length - Math.max(arLags[i]) + nb`, where `nb` is the number of values backcast. This method is only applicable using least-squares algorithm.

---

- *getSSResidual*
`public double` **getSSResidual**`( )`

  – **Description**
  Returns the sum of squares of the random shock.

  – **Returns** – a `double` scalar containing the sum of squares of the random shock, $\mathrm{residual}[0]^2 + \ldots + \mathrm{residual}[\mathrm{na} - 1]^2$, where `residual` is the array return from the `getResidual` method and `na = residual.length` . This method is only applicable using least-squares algorithm.

---

- *getVariance*
`public double` **getVariance**`( )`

  – **Description**
  Returns the variance of the time series `z`.

  – **Returns** – a `double` scalar containing the variance of the time series `z`

---

- *setARLags*
  `public void` **setARLags**`( int[]` **arLags** `)`
  - **Description**
    Sets the order of the autoregressive parameters.
  - **Parameters**
    * `arLags` – an `int` array of length `p` containing the order of the autoregressive parameters. The elements of `arLags` must be greater than or equal to 1. Default: `arLags = [1, 2, ..., p]`

---

- *setBackcasting*
  `public void` **setBackcasting**`( int` **length**`, double` **tolerance** `)`
  - **Description**
    Sets backcasting option.
  - **Parameters**
    * `length` – an `int` scalar containing the maximum length of backcasting and must be greater than or equal to 0. Default: `length = 10`.
    * `tolerance` – a `double` scalar containing the tolerance level used to determine convergence of the backcast algorithm. Typically, `tolerance` is set to a fraction of an estimate of the standard deviation of the time series. Default: `tolerance = 0.01 *` standard deviation of `z`.

---

- *setBackwardOrigin*
  `public void` **setBackwardOrigin**`( int` **backwardOrigin** `)`
  - **Description**
    Sets the maximum backward origin.
  - **Parameters**
    * `backwardOrigin` – an `int` scalar specifying the maximum backward origin. `backwardOrigin` must be greater than or equal to 0 and less than or equal to `z.length - Math.max(maxar, maxma)`, where `maxar = Math.max(arLags[i])`, `maxma = Math.max(maLags[j])`, and forecasts at `origins z.length - backwardOrigin` through `z.length` are generated. Default: `backwardOrigin = 0`.

---

- *setCenter*
  `public void` **setCenter**`( boolean` **center** `)`
  - **Description**
    Sets center option.
  - **Parameters**

* `center` – a `boolean` scalar. If `false` is specified, the time series is not centered about its mean, `zMean`. If `true` is specified, the time series is centered about its mean. Default: `center = true`.

---

- *setConfidence*
  `public void` **setConfidence**`( double` **confidence** `)`
  - **Description**
    Sets the confidence percent probability limits of the forecasts.
  - **Parameters**
    * `confidence` – a `double` scalar specifying the confidence percent probability limits of the forecasts. Typical choices for confidence are 0.90, 0.95, and 0.99. `confidence` must be greater than 0.0 and less than 1.0. Default: `confidence = 0.95`.

---

- *setConvergenceTolerance*
  `public void` **setConvergenceTolerance**`( double` **convergenceTolerance** `)`
  - **Description**
    Sets the tolerance level used to determine convergence of the nonlinear least-squares algorithm.
  - **Parameters**
    * `convergenceTolerance` – a `double` scalar containing the tolerance level used to determine convergence of the nonlinear least-squares algorithm. `convergenceTolerance` represents the minimum relative decrease in sum of squares between two iterations required to determine convergence. Hence, `convergenceTolerance` must be greater than or equal to 0. The default value is $\max(10^{-20}, \mathrm{eps}^{2/3})$, where `eps = 2.2204460492503131e-16`.

---

- *setInitialEstimates*
  `public void` **setInitialEstimates**`( double[]` **ar**`, double[]` **ma** `)`
  - **Description**
    Sets preliminary estimates.
  - **Parameters**
    * `ar` – a `double` array of length `p` containing preliminary estimates of the autoregressive parameters. `ar` is computed internally if this method is not used. This method is only applicable using least-squares algorithm.
    * `ma` – a `double` array of length `q` containing preliminary estimates of the moving average parameters. `ma` is computed internally if this method is not used. This method is only applicable using least-squares algorithm.

---

- *setMALags*
  `public void` **setMALags**`( int[]` **maLags** `)`

– **Description**
Sets the order of the moving average parameters.

– **Parameters**

* `maLags` – an `int` array of length `q` containing the order of the moving average parameters. The `maLags` elements must be greater than or equal to 1. Default: `maLags = [1, 2, ..., q]`

---

- *setMaxIterations*
`public void` **setMaxIterations**`( int` **iterations** `)`

– **Description**
Sets the maximum number of iterations.

– **Parameters**

* `iterations` – an `int` scalar specifying the maximum number of iterations allowed in the nonlinear equation solver used in both the method of moments and least-squares algorithms. Default: `interations = 200`.

---

- *setMeanEstimate*
`public void` **setMeanEstimate**`( double` **zMean** `)`

– **Description**
Sets an initial estimate of the mean of the time series `z`.

– **Parameters**

* `zMean` – a `double` scalar containing an initial estimate of the mean of the time series `z`. If the time series is not centered about its mean, and least-squares algorithm is used, `zMean` is not used in parameter estimation.

---

- *setMethod*
`public void` **setMethod**`( int` **method** `)`

– **Description**
Sets the method to be used by the class.

– **Parameters**

* `method` – an `int` scalar specifying the method to be use. If `ARMA.METHOD_OF_MOMENTS` is specified, the autoregressive and moving average parameters are estimated by a method of moments procedure. If `ARMA.LEAST_SQUARES` is specified, the autoregressive and moving average parameters are estimated by a least-squares procedure. Default `method = ARMA.METHOD_OF_MOMENTS`.

---

- *setRelativeError*
`public void` **setRelativeError**`( double` **relativeError** `)`

– **Description**
  Sets the stopping criterion for use in the nonlinear equation solver.
– **Parameters**
  * `relativeError` – a `double` scalar containing the stopping criterion for use in the nonlinear equation solver used in both the method of moments and least-squares algorithms. Default: `relativeError = 2.2204460492503131e-14`.

## Example 1: ARMA

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. The method of moments estimates

$$\hat{\theta}_0, \hat{\phi}_1, \hat{\phi}_2, \text{and}\, \hat{\theta}_1$$

for the ARMA(2, 1) model

$$z_t = \theta_0 + \phi_1 z_{t-1} + \phi_2 z_{t-2} - \theta_1 A_{t-1} + A_t$$

where the errors $A_t$ are independently normally distributed with mean zero and variance

$$\sigma_A^2$$

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class ARMAEx1 {
    public static void main(String args[]) throws Exception {
        double[] z = {100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5,
        154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9,
        132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16,
        6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1,
        8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4,
        23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5,
        67, 71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2,
        85.8, 63.2, 36.8, 24.2, 10.7, 15, 40.1, 61.5, 98.5,
        124.3, 95.9, 66.5, 64.5, 54.2, 39, 20.6, 6.7, 4.3, 22.8,
```

```
        54.8, 93.8, 95.7, 77.2, 59.1, 44, 47, 30.5, 16.3, 7.3,
        37.3, 73.9};

        ARMA arma = new ARMA(2, 1, z);
        arma.setRelativeError(0.0);
        arma.setMaxIterations(0);
        arma.compute();

        new PrintMatrix("AR estimates are:  ").print(arma.getAR());
        System.out.println();
        new PrintMatrix("MA estimate is:  ").print(arma.getMA());
    }
}
```

## Output

```
AR estimates are:
     0
0   1.244
1  -0.575


MA estimate is:
     0
0  -0.124
```

## Example 2: ARMA

The data for this example are the same as that for Example 1. Preliminary method of moments estimates are computed by default, and the method of least squares is used to find the final estimates. Note that at the end of the output, a warning message appears. In most cases, this warning message can be ignored. There are three general reasons this warning can occur:

1. Convergence is declared using the criterion based on tolerance, but the gradient of the residual sum-of-squares function is nonzero. This occurs in this example. Either the message can be ignored or `tolerance` can be reduced to allow more iterations and a slightly more accurate solution.

2. Convergence is declared based on the fact that a very small step was taken, but the gradient of the residual sum-of-squares function was nonzero. This message can usually be ignored. Sometimes, however, the algorithm is making very slow progress and is not near a minimum.
3. Convergence is not declared after 100 iterations.

Trying a smaller value for `tolerance` can help determine what caused the error message.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class ARMAEx2 {
    public static void main(String args[]) throws Exception {
        double[] arInit = {1.24426e0, -5.75149e-1};
        double[] maInit = {-1.24094e-1};
        double[] z = {100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5,
        154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9,
        132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16,
        6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1,
        8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4,
        23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5,
        67, 71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2,
        85.8, 63.2, 36.8, 24.2, 10.7, 15, 40.1, 61.5, 98.5,
        124.3, 95.9, 66.5, 64.5, 54.2, 39, 20.6, 6.7, 4.3, 22.8,
        54.8, 93.8, 95.7, 77.2, 59.1, 44, 47, 30.5, 16.3, 7.3,
        37.3, 73.9};

        ARMA arma = new ARMA(2, 1, z);
        arma.setMethod(arma.LEAST_SQUARES);
        arma.setInitialEstimates(arInit, maInit);
        arma.setConvergenceTolerance(0.125);
        arma.setMeanEstimate(46.976);
        arma.compute();

        new PrintMatrix("AR estimates are:  ").print(arma.getAR());
        System.out.println();
        new PrintMatrix("MA estimate is:  ").print(arma.getMA());
    }
}
```

## Output

```
AR estimates are:
     0
0   1.393
1  -0.734


MA estimate is:
     0
0  -0.137
```

*Warning* com.imsl.stat.ARMA: Relative function convergence - Both the scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance “convergence_tolerance” = 0.065. com.imsl.stat.ARMA: Least squares estimation of the parameters has failed to converge. Increase “length” and/or “tolerance” and/or “convergence_tolerance”. The estimates of the parameters at the last iteration may be used as new starting values.

## Example 3: Forecasting

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. The data set for this example consists of the number of sunspots observed from 1770 through 1869. Method `forecast` in class ARMA computes forecasts and 95-percent probability limits for the forecasts for an ARMA(2, 1) model fit using the method of moments option. With `backward_origin` = 3, `forecast` method provides forecasts given the data through 1866, 1867, 1868, and 1869, respectively. The deviations from the forecast for computing probability limits, and the psi weights can be used to update forecasts when more data is available. For example, the forecast for the 102-nd observation (year 1871) given the data through the 100-th observation (year 1869) is 77.21; and 95-percent probability limits are given by $77.21 \pm 56.30$. After observation 101 ( $Z_{101}$ for year 1870) is available, the forecast can be updated by using

$$\hat{Z}_t\left(l\right) \pm z_{\alpha/2}\left\{1 + \sum_{j=1}^{l-1} \psi_j^2\right\}^{1/2} \sigma_A$$

with the psi weight ($\psi_1 = 1.37$) and the one-step-ahead forecast error for observation

$101(Z_{101} - 83.72)$ to give the following:

$$77.21 + 1.37 \times (Z_{101} - 83.72)$$

Since this updated forecast is one step ahead, the 95-percent probability limits are now given by the forecast $\pm 33.22$.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class ARMAEx3 {
    public static void main(String args[]) throws Exception {
        double[] z = {100.8, 81.6, 66.5, 34.8, 30.6, 7, 19.8, 92.5,
        154.4, 125.9, 84.8, 68.1, 38.5, 22.8, 10.2, 24.1, 82.9,
        132, 130.9, 118.1, 89.9, 66.6, 60, 46.9, 41, 21.3, 16,
        6.4, 4.1, 6.8, 14.5, 34, 45, 43.1, 47.5, 42.2, 28.1, 10.1,
        8.1, 2.5, 0, 1.4, 5, 12.2, 13.9, 35.4, 45.8, 41.1, 30.4,
        23.9, 15.7, 6.6, 4, 1.8, 8.5, 16.6, 36.3, 49.7, 62.5, 67,
        71, 47.8, 27.5, 8.5, 13.2, 56.9, 121.5, 138.3, 103.2,
        85.8, 63.2, 36.8, 24.2, 10.7, 15, 40.1, 61.5, 98.5, 124.3,
        95.9, 66.5, 64.5, 54.2, 39, 20.6, 6.7, 4.3, 22.8, 54.8,
        93.8, 95.7, 77.2, 59.1, 44, 47, 30.5, 16.3, 7.3, 37.3,
        73.9};
        PrintMatrixFormat pmf = new PrintMatrixFormat();

        ARMA arma = new ARMA(2, 1, z);
        arma.setRelativeError(0.0);
        arma.setMaxIterations(0);
        arma.compute();

        System.out.println("Method of Moments initial estimates:");
        new PrintMatrix("AR estimates are:  ").print(arma.getAR());
        System.out.println();
        new PrintMatrix("MA estimate is:  ").print(arma.getMA());
        arma.setBackwardOrigin(3);

        String[] labels = { "Forecast From 1866", "Forecast From 1867",
        "Forecast From 1868", "Forecast From 1869"};
        pmf.setColumnLabels(labels);
        new PrintMatrix("forecasts:  ").print(pmf, arma.forecast(12));
```

```
        String[] devlabel = {"Dev. for prob. limits"};
        pmf.setColumnLabels(devlabel);
        new PrintMatrix().print(pmf, arma.getDeviations());

        pmf = new PrintMatrixFormat();
        String[] psilabel = {"Psi"};
        pmf.setColumnLabels(psilabel);
        new PrintMatrix().print(pmf, arma.getPsiWeights());
    }
}
```

## Output

```
Method of Moments initial estimates:
AR estimates are:
     0
0   1.244
1  -0.575


MA estimate is:
     0
0  -0.124

                                    forecasts:
    Forecast From 1866  Forecast From 1867  Forecast From 1868  Forecast From 1869
 0        18.283              16.615              55.189              83.72
 1        28.918              32.019              62.761              77.209
 2        41.01               45.827              61.892              63.461
 3        49.939              54.15               56.457              50.099
 4        54.094              56.562              50.194              41.38
 5        54.128              54.778              45.527              38.217
 6        51.782              51.17               43.322              39.296
 7        48.842              47.707              43.263              42.458
 8        46.533              45.474              44.458              45.772
 9        45.352              44.686              45.978              48.076
10        45.21               44.991              47.183              49.037
11        45.713              45.823              47.807              48.908

    Dev. for prob. limits
```

| | |
|---|---|
| 0 | 33.218 |
| 1 | 56.298 |
| 2 | 67.617 |
| 3 | 70.643 |
| 4 | 70.751 |
| 5 | 71.087 |
| 6 | 71.907 |
| 7 | 72.534 |
| 8 | 72.75 |
| 9 | 72.765 |
| 10 | 72.778 |
| 11 | 72.823 |

| | Psi |
|---|---|
| 0 | 1.368 |
| 1 | 1.127 |
| 2 | 0.616 |
| 3 | 0.118 |
| 4 | -0.208 |
| 5 | -0.326 |
| 6 | -0.286 |
| 7 | -0.169 |
| 8 | -0.045 |
| 9 | 0.041 |
| 10 | 0.077 |
| 11 | 0.072 |

## *class* **Difference**

Differences a seasonal or nonseasonal time series.

Class `Difference` performs m = `periods.length` successive backward differences of period $s_i = \text{periods}[i-1]$ and order $d_i = \text{orders}[i-1]$ for $i = 1, \ldots, m$ on the $n =$ `z.length` observations $\{Z_t\}$ for $t = 1, 2, \ldots, n$.

Consider the backward shift operator $B$ given by

$$B^k Z_t = Z_{t-k}$$

for all $k$. Then, the *backward difference operator* with period $s$ is defined by the following:

$$\Delta_s Z_t = (1 - B^s)\, Z_t = Z_t - Z_{t-s} \quad \text{for } s \geq 0$$

Note that $B_s Z_t$ and $\Delta_s Z_t$ are defined only for $t = (s+1), \ldots, n$. Repeated differencing with period $s$ is simply

$$\Delta_s^d Z_t = (1 - B^s)^d\, Z_t = \sum_{j=0}^{d} \frac{d!}{j!\,(d-j)!} (-1)^j\, B^{sj} Z_t$$

where $d \geq 0$ is the order of differencing. Note that

$$\Delta_s^d Z_t$$

is defined only for $t = (sd+1), \ldots, n$.

The general difference formula used in the class `Difference` is given by

$$W_T = \begin{cases} \text{NaN} & \text{for } t = 1, \ldots, n_L \\ \Delta_{s_1}^{d_1} \Delta_{s_2}^{d2} \ldots \Delta_{s_m}^{d_m} Z_t & \text{for } t = n_L + 1, \ldots, n \end{cases}$$

where $n_L$ represents the number of observations "lost" because of differencing and NaN represents the missing value code. Note that

$$n_L = \sum_j s_j d_j$$

A homogeneous, stationary time series can be arrived at by appropriately differencing a homogeneous, nonstationary time series (Box and Jenkins 1976, p. 85). Preliminary application of an appropriate transformation followed by differencing of a series can enable model identification and parameter estimation in the class of homogeneous stationary autoregressive moving average models.

## Declaration

public class com.imsl.stat.Difference
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *Difference*
  `public` **Difference**`( )`

  - **Description**
    Constructor for `Difference`.

## Methods

---

- *compute*
  `public final double[]` **compute**`( double[]` **z**`, int[]` **periods** `) throws java.lang.IllegalArgumentException`

  - **Description**
    Computes a Difference series.
  - **Parameters**
    * `z` – a `double` array containing the time series.
    * `periods` – an `int` array containing the periods at which z is to be differenced.
  - **Returns** – a `double` array containing the differenced series.

---

- *excludeFirst*
  `public void` **excludeFirst**`( boolean` **exclude** `)`

  - **Description**
    If set to true, the observations lost due to differencing will be excluded. The differenced series will be the length of the number of observations minus the number of observations lost. If set to false, the observations lost due to differencing will be set to NaN (Not a number) and included in the differenced series. The default is to set the lost observations to NaN.
  - **Parameters**
    * `exclude` – a `boolean` specifying whether or not to exclude lost observations due to differencing.

---

- *getObservationsLost*
  `public int` **getObservationsLost**`( )`

  - **Description**
    Returns the number of observations lost because of differencing the time series.
  - **Returns** – an `int` containing the number of observations lost because of differencing the time series `z`.

---

- *setOrders*
  `public void setOrders( int[] orders )`

  - **Description**
    Sets the orders for the Difference object
  - **Parameters**
    * `orders` – an `int` array of length equal to length of `periods`, containing the order of each difference given in periods. The elements of orders must be greater than or equal to 0.

### Example 1: Difference

This example uses the Airline Data (Box and Jenkins 1976, p. 531) consisting of the monthly total number of international airline passengers from January 1949 through December 1960. Difference is used to compute ...

$$W_t = \Delta_1 \Delta_{12} Z_t = (Z_t - Z_{t-12}) - (Z_{t-1} - Z_{t-13})$$

for t= 14, 15, ...,24.

```
import com.imsl.stat.*;

public class  DifferenceEx1 {
    public static void main(String args[]) {

        int periods[] = {1, 12};
        int nLost;
        double[] z = {
            112.0,118.0,132.0,129.0,121.0,135.0,
            148.0,148.0,136.0,119.0,104.0,118.0,
            115.0,126.0,141.0,135.0,125.0,149.0,
            170.0,170.0,158.00,133.0,114.0,140.0
        };

        Difference diff = new Difference();
        double[] out = diff.compute(z, periods);
        nLost = diff.getObservationsLost();

        System.out.println("Observations Lost = " + nLost);

        for (int i = 0; i < out.length; i++)
            System.out.println(out[i]);
```

```
    }
}
```

### Output

```
Observations Lost = 13
NaN
NaN
NaN
NaN
NaN
NaN
NaN
NaN
NaN
NaN
NaN
NaN
NaN
5.0
1.0
-3.0
-2.0
10.0
8.0
0.0
0.0
-8.0
-4.0
12.0
```

### Example 2: Difference

This example uses the same data as Example 1. The first number of lost observations are excluded from W due to differencing, and the number of lost observations is also output.

```
import com.imsl.stat.*;

public class  DifferenceEx2 {
    public static void main(String args[]) {
```

```
        int periods[] = {1, 12};
        int nLost;
        double[] z={
            112.0,118.0,132.0,129.0,121.0,135.0,
            148.0,148.0,136.0,119.0,104.0,118.0,
            115.0,126.0,141.0,135.0,125.0,149.0,
            170.0,170.0,158.00,133.0,114.0,140.0
        };

        Difference diff = new Difference();
        diff.excludeFirst(true);
        double[] out = diff.compute(z, periods);
        nLost = diff.getObservationsLost();

        System.out.println("The number of observation lost = "
        + nLost);
        for (int i=0; i < out.length; i++)
            System.out.println(out[i]);
    }
}
```

## Output

```
The number of observation lost = 13
5.0
1.0
-3.0
-2.0
10.0
8.0
0.0
0.0
-8.0
-4.0
12.0
```

## *class* **GARCH**

Computes estimates of the parameters of a GARCH(p,q) model.

The Generalized Autoregressive Conditional Heteroskedastic (GARCH) model is defined as

$$y_t = z_t \sigma_t$$

$$\sigma_t^2 = \sigma^2 + \sum_{i=1}^{p} \beta_i \sigma_{t-i}^2 + \sum_{i=1}^{q} \alpha_i y_{t-i}^2$$

where $z_t$‘s are independent and identically distributed standard normal random variables,

$$\sigma > 0, \beta_i \geq 0, \alpha_i \geq 0$$

and

$$\sum_{i=1}^{p} \beta_i + \sum_{i=1}^{q} \alpha_i < 1$$

The above model is denoted as GARCH(p, q). The $p$ is the autoregressive lag and the $q$ is the moving average lag. When $\beta_i = 0, i = 1, 2, \ldots, p$, the above model reduces to ARCH(q) which was proposed by Engle (1982). The nonnegativity conditions on the parameters implied a nonnegative variance and the condition on the sum of the $\beta_i$‘s and $\alpha_i$‘s is required for wide sense stationarity.

In the empirical analysis of observed data, GARCH(1,1) or GARCH(1,2) models have often found to appropriately account for conditional heteroskedasticity (Palm 1996). This finding is similar to linear time series analysis based on ARMA models.

It is important to notice that for the above models positive and negative past values have a symmetric impact on the conditional variance. In practice, many series may have strong asymmetric influence on the conditional variance. To take into account this phenomena, Nelson (1991) put forward Exponential GARCH (EGARCH). Lai (1998) proposed and studied some properties of a general class of models that extended linear relationship of the conditional variance in ARCH and GARCH into nonlinear fashion.

The maximal likelihood method is used in estimating the parameters in GARCH(p,q). The log-likelihood of the model for the observed series $\{Y_t\}$ with length $m$ is

$$\log(L) = \frac{m}{2}\log(2\pi) - \frac{1}{2}\sum_{t=1}^{m} y_t^2/\sigma_t^2 - \frac{1}{2}\sum_{t=1}^{m}\log\sigma_t^2,$$

$$\text{where } \sigma_t^2 = \sigma^2 + \sum_{i=1}^{p}\beta_i\sigma_{t-i}^2 + \sum_{i=1}^{q}\alpha_i y_{t-i}^2.$$

In the model, if $q = 0$, the model GARCH is singular such that the estimated Hessian matrix $H$ is singular.

The initial values of the parameter array $x[\,]$ entered in array `xguess[ ]` must satisfy certain constraints. The first element of `xguess` refers to sigma and must be greater than zero and less than `maxSigma`. The remaining $p+q$ initial values must each be greater than or equal to zero but less than one.

To guarantee stationarity in model fitting,

$$\sum_{i=1}^{p+q} x(i) < 1,$$

is checked internally. The initial values should be selected from the values between zero and one. The value of Akaike Information Criterion is computed by

$$2 \times \log(\mathrm{L}) + 2 \times (\mathrm{p} + \mathrm{q} + 1),$$

where log(L) is the value of the log-likelihood function at the estimated parameters.

In fitting the optimal model, the class com.imsl.math.MinConGenLin, is modified to find the maximal likelihood estimates of the parameters in the model. Statistical inferences can be performed outside of the class `GARCH` based on the output of the log-likelihood function (`getlogLikelihood` method), the Akaike Information Criterion (`getAkaike` method), and the variance-covariance matrix (`getVarCovarMatrix` method).

## Declaration

public class com.imsl.stat.GARCH
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Inner Classes

### *class* GARCH.VarsDeterminedException

The variables are determined by the equality constraints.

#### Declaration

public static class com.imsl.stat.GARCH.VarsDeterminedException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *GARCH.VarsDeterminedException*
  `public` **GARCH.VarsDeterminedException**`( java.lang.String` **message** `)`

- *GARCH.VarsDeterminedException*
  `public` **GARCH.VarsDeterminedException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

### *class* GARCH.TooManyIterationsException

Number of function evaluations exceeded 1000.

#### Declaration

public static class com.imsl.stat.GARCH.TooManyIterationsException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *GARCH.TooManyIterationsException*
  `public` **GARCH.TooManyIterationsException**`( java.lang.String` **message**
  `)`

- *GARCH.TooManyIterationsException*
  `public` **GARCH.TooManyIterationsException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

## *class* **GARCH.NoVectorXException**

No vector X satisfies all of the constraints.

### Declaration

public static class com.imsl.stat.GARCH.NoVectorXException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

- *GARCH.NoVectorXException*
  `public` **GARCH.NoVectorXException**`( java.lang.String` **message** `)`
- *GARCH.NoVectorXException*
  `public` **GARCH.NoVectorXException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

## *class* **GARCH.EqConstrInconsistentException**

The equality constraints and the bounds on the variables are found to be inconsistent.

### Declaration

public static class com.imsl.stat.GARCH.EqConstrInconsistentException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

- *GARCH.EqConstrInconsistentException*
  `public` **GARCH.EqConstrInconsistentException**`( java.lang.String`
  **message** `)`
- *GARCH.EqConstrInconsistentException*
  `public` **GARCH.EqConstrInconsistentException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

### *class* **GARCH.ConstrInconsistentException**

The equality constraints are inconsistent.

#### Declaration

public static class com.imsl.stat.GARCH.ConstrInconsistentException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

---

- *GARCH.ConstrInconsistentException*
  `public` **GARCH.ConstrInconsistentException**`( java.lang.String` **message** `)`

- *GARCH.ConstrInconsistentException*
  `public` **GARCH.ConstrInconsistentException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

### Constructor

---

- *GARCH*
  `public` **GARCH**`( int` **p**`, int` **q**`, double[]` **y**`, double[]` **xguess** `)`
  - **Description**
    Constructor for `GARCH`.
  - **Parameters**
    * `p` – An `int` scalar containing the number of autoregressive (AR) parameters.
    * `q` – An `int` scalar containing the number of moving average (MA) parameters.
    * `y` – A `double` array containing the observed time series data.
    * `xguess` – A `double` array of length `p` + `q` + 1 containing the initial values for the parameter array.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the dimensions of `y`, and `xguess` are not consistent.

## Methods

- *compute*
`public final void` **compute**`( ) throws com.imsl.stat.GARCH.ConstrInconsistentException, com.imsl.stat.GARCH.EqConstrInconsistentException, com.imsl.stat.GARCH.NoVectorXException, com.imsl.stat.GARCH.TooManyIterationsException, com.imsl.stat.GARCH.VarsDeterminedException`
  - **Description**
    Computes estimates of the parameters of a GARCH(p,q) model.
  - **Throws**
    * `com.imsl.stat.GARCH.EqConstrInconsistentException` – is thrown if the equality constraints are inconsistent.
    * `com.imsl.stat.GARCH.EqConstrInconsistentException` – is thrown if the equality constraints and the bounds on the variables are found to be inconsistent.
    * `com.imsl.stat.GARCH.NoVectorXException` – is thrown if no vector X satisfies all of the constraints.
    * `com.imsl.stat.GARCH.TooManyIterationsException` – is thrown if the number of function evaluations exceeded 1000.
    * `com.imsl.stat.GARCH.VarsDeterminedException` – is thrown if the variables are determined by the equality constraints.

- *getAkaike*
`public double` **getAkaike**`( )`
  - **Description**
    Returns the value of Akaike Information Criterion evaluated at the estimated parameter array.
  - **Returns** – a `double` scalar containing the value of Akaike Information Criterion evaluated at the estimated parameter array.

- *getAR*
`public double[]` **getAR**`( )`
  - **Description**
    Returns the estimated values of autoregressive (AR) parameters.
  - **Returns** – a `double` array of size `p` containing the estimated values of autoregressive (AR) parameters.

- *getLogLikelihood*
  `public double` **getLogLikelihood**`( )`
  - **Description**
    Returns the value of Log-likelihood function evaluated at the estimated parameter array.
  - **Returns** – a `double` scalar containing the value of Log-likelihood function evaluated at the estimated parameter array.

---

- *getMA*
  `public double[]` **getMA**`( )`
  - **Description**
    Returns the estimated values of moving average (MA) parameters.
  - **Returns** – a `double` array of size `q` containing the estimated values of moving average (MA) parameters.

---

- *getSigma*
  `public double` **getSigma**`( )`
  - **Description**
    Returns the estimated value of sigma squared.
  - **Returns** – a `double` scalar containing the estimated value of sigma squared.

---

- *getVarCovarMatrix*
  `public double[][]` **getVarCovarMatrix**`( )`
  - **Description**
    Returns the variance-covariance matrix.
  - **Returns** – a `double` matrix of size `p + q + 1` by `p + q + 1` containing the variance-covariance matrix.

---

- *getX*
  `public double[]` **getX**`( )`
  - **Description**
    Returns the estimated parameter array, `x`.
  - **Returns** – a `double` array of size `p + q + 1` containing the estimated values of sigma squared, the AR parameters, and the MA parameters.

---

- *setMaxSigma*
  `public void` **setMaxSigma**`( double` **maxSigma** `)`
  - **Description**
    Sets the value of the upperbound on the first element (sigma) of the array of returned estimated coefficients.

– **Parameters**
  * `maxSigma` – A `double` scalar containing the value of the upperbound on the first element (sigma) of the array of returned estimated coefficients.
    Default = 10.

## Example: GARCH

The data for this example are generated to follow a GARCH(p,q) process by using a random number generation function *sgarch*. The data set is analyzed and estimates of sigma, the AR parameters, and the MA parameters are returned. The values of the Log-likelihood function and the Akaike Information Criterion are returned.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class GARCHEx1 {
    static private void sgarch(int p, int q, int m, double[] x, double[] y,
    double[] z, double[] y0, double[] sigma) {
        int i, j, l;
        double s1, s2, s3;
        Random rand = new Random(182198625L);

        rand.setMultiplier(16807);
        for (i = 0; i < m+1000; i++) z[i] = rand.nextNormal();

        l = Math.max(p, q);
        l = Math.max(l, 1);
        for(i =0; i <l; i++) y0[i] = z[i] * x[0];

        /* COMPUTE THE INITIAL VALUE OF SIGMA */
        s3 = 0.0;
        if (Math.max(p, q) >= 1) {
            for(i =1; i <(p +q +1); i++) s3 += x[i];
        }
        for(i =0;i <l;i++) sigma[i] = x[0] / (1.0 - s3);
        for(i =l;i <(m +1000); i++) {
            s1 = 0.0;
            s2 = 0.0;
            if (q >= 1) {
                for(j =0;j <q;j++) s1+=x[j +1]*y0[i -j -1]*y0[i -j -1];
            }
            if (p >= 1) {
```

```
                for(j =0;j <p;j++) s2+=x[q +1 +j]*sigma[i -j -1];
            }
            sigma[i] = x[0] + s1 + s2;
            y0[i] = z[i] * Math.sqrt(sigma[i]);
        }
/*
 * DISCARD THE FIRST 1000 SIMULATED OBSERVATIONS
 */
        for(i =0;i <m;i++) y[i] = y0[1000 + i];
        return;
    }



    public static void main(String args[]) throws Exception {
        int n, p, q, m;
        double[] x = {1.3, 0.2, 0.3, 0.4};
        double[] xguess = {1.0, 0.1, 0.2, 0.3};
        double[] y = new double[1000];
        double[] wk1 = new double[2000];
        double[] wk2 = new double[2000];
        double[] wk3 = new double[2000];
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(3);

        m = 1000;
        p = 2;
        q = 1;
        n = p+q+1;
        sgarch(p, q, m, x, y, wk1, wk2, wk3);

        GARCH garch = new GARCH(p, q, y, xguess);
        garch.compute();

        System.out.println("Sigma estimate is " + nf.format(garch.getSigma()));
        System.out.println();
        new PrintMatrix("AR estimate is ").print(garch.getAR());
        new PrintMatrix("MR estimate is ").print(garch.getMA());
        System.out.println("Log-likelihood function value is " +
        nf.format(garch.getLogLikelihood()));
        System.out.println("Akaike Information Criterion value is " +
        nf.format(garch.getAkaike()));
```

```
    }
}
```

### Output

```
Sigma estimate is 1.692

AR estimate is
     0
0  0.245
1  0.337

MR estimate is
    0
0  0.31

Log-likelihood function value is -2,707.072
Akaike Information Criterion value is 5,422.144
```

## *class* **KalmanFilter**

Performs Kalman filtering and evaluates the likelihood function for the state-space model.

Class `KalmanFilter` is based on a recursive algorithm given by Kalman (1960), which has come to be known as the Kalman filter. The underlying model is known as the state-space model. The model is specified stage by stage where the stages generally correspond to time points at which the observations become available. `KalmanFilter` avoids many of the computations and storage requirements that would be necessary if one were to process all the data at the end of each stage in order to estimate the state vector. This is accomplished by using previous computations and retaining in storage only those items essential for processing of future observations.

The notation used here follows that of Sallas and Harville (1981). Let $y_k$ (input in `y` using method `update`) be the $n_k \times 1$ vector of observations that become available at time $k$. The subscript $k$ is used here rather than $t$, which is more customary in time series, to emphasize that the model is expressed in stages $k = 1, 2, \ldots$ and that these stages need not correspond to equally spaced time points. In fact, they need not correspond to time points of any kind. The *observation equation* for the state-space model is

$$y_k = Z_k b_k + e_k \quad k = 1, 2, \ldots$$

Here, $Z_k$ (input in `z` using method `update`) is an $n_k \times q$ known matrix and $b_k$ is the $q \times 1$ state vector. The state vector $b_k$ is allowed to change with time in accordance with the *state equation*

$$b_{k+1} = T_{k+1} b_k + w_{k+1} \quad k = 1, 2, \ldots$$

starting with $b_1 = \mu_1 + w_1$.

The change in the state vector from time $k$ to $k + 1$ is explained in part by the *transition matrix* $T_{k+1}$ (the identity matrix by default, or optionally using method `setTransitionMatrix`), which is assumed known. It is assumed that the $q$-dimensional $w_k s (k = 1, 2, \ldots)$ are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 Q_k$, that the $n_k$-dimensional $e_k s (k = 1, 2, \ldots)$ are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 R_k$, and that the $w_k s$ and $e_k s$ are independent of each other. Here, $\mu_1$is the mean of $b_1$ and is assumed known, $\sigma^2$ is an unknown positive scalar. $Q_{k+1}$ (input in `Q`) and $R_k$ (input in `R`) are assumed known.

Denote the estimator of the realization of the state vector $b_k$ given the observations $y_1, y_2, \ldots, y_j$ by

$$\hat{\beta}_{k|j}$$

By definition, the mean squared error matrix for

$$\hat{\beta}_{k|j}$$

is

$$\sigma^2 C_{k|j} = E(\hat{\beta}_{k|j} - b_k)(\hat{\beta}_{k|j} - b_k)^T$$

At the time of the $k$-th invocation, we have

$$\hat{\beta}_{k|k-1}$$

and

$C_{k|k-1}$, which were computed from the $k$-1-st invocation, input in `b` and `covb`, respectively. During the $k$-th invocation, `KalmanFilter` computes the filtered estimate

$$\hat{\beta}_{k|k}$$

along with $C_{k|k}$. These quantities are given by the *update equations*:

$$\hat{\beta}_{k|k} = \hat{\beta}_{k|k-1} + C_{k|k-1} Z_k^T H_k^{-1} v_k$$

$$C_{k|k} = C_{k|k-1} - C_{k|k-1} Z_k^T H_k^{-1} Z_k C_{k|k-1}$$

where

$$v_k = y_k - Z_k \hat{\beta}_{k|k-1}$$

and where

$$H_k = R_k + Z_k C_{k|k-1} Z_k^T$$

Here, $v_k$ (stored in `getPredictionError`) is the one-step-ahead prediction error, and $\sigma^2 H_k$ is the variance-covariance matrix for $v_k$. $H_k$ is obtained from method `getCovV`. The "start-up values" needed on the first invocation of `KalmanFilter` are

$$\hat{\beta}_{1|0} = \mu_1$$

and $C_{1|0} = Q_1$ input via `b` and `covb`, respectively. Computations for the $k$-th invocation are completed by `KalmanFilter` computing the one-step-ahead estimate

$$\hat{\beta}_{k+1|k}$$

along with $C_{k+1|k}$ given by the *prediction equations*:

$$\hat{\beta}_{k+1|k} = T_{k+1} \hat{\beta}_{k|k}$$

$$C_{k+1|k} = T_{k+1} C_{k|k} T_{k+1}^T + Q_{k+1}$$

If both the filtered estimates and one-step-ahead estimates are needed by the user at each time point, `KalmanFilter` can be used twice for each time point-first without methods `SetTransitionMatrix` and `setQ` to produce

$$\hat{\beta}_{k|k}$$

and $C_{k|k}$, and second without method `update` to produce

$$\hat{\beta}_{k+1|k}$$

and $C_{k+1|k}$ (Without methods `SetTransitionMatrix` and `setQ`, the prediction equations are skipped. Without method `update`, the update equations are skipped.).

Often, one desires the estimate of the state vector more than one-step-ahead, i.e., an estimate of

$$\hat{\beta}_{k|j}$$

is needed where $k > j + 1$. At time $j$, `KalmanFilter` is invoked with method `update` to compute

$$\hat{\beta}_{j+1|j}$$

Subsequent invocations of `KalmanFilter` without method `update` can compute

$$\hat{\beta}_{j+2|j},\ \hat{\beta}_{j+3|j},\ \ldots,\ \hat{\beta}_{k|j}$$

Computations for

$$\hat{\beta}_{k|j}$$

and $C_{k|j}$ assume the variance-covariance matrices of the errors in the observation equation and state equation are known up to an unknown positive scalar multiplier, $\sigma^2$. The maximum likelihood estimate of $\sigma^2$ based on the observations $y_1, y_2, \ldots, y_m$, is given by

$$\hat{\sigma}^2 = SS/N$$

where

$$N = \sum_{k=1}^{m} n_k \text{ and } SS = \sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

$N$ and $SS$ are input arguments `rank` and `SumofSquares`. Updated values are obtained from methods `getRank` and `getSumofSquares`

If $\sigma^2$ is known, the $R_k s$ and $Q_k s$ can be input as the variance-covariance matrices exactly. The earlier discussion is then simplified by letting $\sigma^2 = 1$.

In practice, the matrices $T_k$, $Q_k$, and $R_k$ are generally not completely known. They may be known functions of an unknown parameter vector $\theta$. In this case, `KalmanFilter` can be used in conjunction with an optimization class (see `MinUnconMultiVar`, JMSL Math package), to obtain a maximum likelihood estimate of $\theta$. The natural logarithm of the likelihood function for $y_1, y_2, \ldots, y_m$ differs by no more than an additive constant from

$$L(\theta, \sigma^2; y_1, y_2, \ \ldots, \ y_m) = -\frac{1}{2}N \ln \sigma^2 - \frac{1}{2}\sum_{k=1}^{m} \ln[\det(H_k)] - \frac{1}{2}\sigma^{-2}\sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

(Harvey 1981, page 14, equation 2.21).

Here,

$$\sum\nolimits_{k=1}^{m} \ln[\det(H_k)]$$

(input in `logDeterminant`, updated by `getLogDeterminant`) is the natural logarithm of the determinant of $V$ where $\sigma^2 V$ is the variance-covariance matrix of the observations.

Minimization of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ over all $\theta$ and $\sigma^2$ produces maximum likelihood estimates. Equivalently, minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ where

$$L_c(\theta; y_1, y_2, \ \ldots, \ y_m) = -\frac{1}{2}N \ln\left(\frac{SS}{N}\right) - \frac{1}{2}\sum_{k=1}^{m} \ln[\det(H_k)]$$

produces maximum likelihood estimates

$$\hat{\theta} \text{ and } \hat{\sigma}^2 = SS/N$$

Minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ instead of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$, reduces the dimension of the minimization problem by one. The two optimization problems are equivalent since

$$\hat{\sigma}^2(\theta) = SS(\theta)/N$$

minimizes $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ for all $\theta$, consequently,

$$\hat{\sigma}^2 \, (\theta)$$

can be substituted for $\sigma^2$ in $L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ to give a function that differs by no more than an additive constant from $L_c(\theta; y_1, y_2, \ldots, y_m)$.

The earlier discussion assumed $H_k$ to be nonsingular. If $H_k$ is singular, a modification for singular distributions described by Rao (1973, pages 527-528) is used. The necessary changes in the preceding discussion are as follows:

1. Replace $H_k^{-1}$ by a generalized inverse.
2. Replace $det(H_k)$ by the product of the nonzero eigenvalues of $H_k$.
3. Replace $N$ by $\sum_{k=1}^{m} \text{rank}\,(H_k)$

Maximum likelihood estimation of parameters in the Kalman filter is discussed by Sallas and Harville (1988) and Harvey (1981, pages 111-113).

## Declaration

public class com.imsl.stat.KalmanFilter
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *KalmanFilter*
  `public` **KalmanFilter**`(` `double[]` **b**`,` `double[]` **covb**`,` `int` **rank**`,` `double` **sumOfSquaress**`,` `double` **logDeterminant** `)`
  - **Description**
    Constructor for `KalmanFilter`.
  - **Parameters**
    * `b` – A `double` array containing the estimated state vector. `b` is the estimated state vector at time `k` given the observations through time `k`-1.
    * `covb` – A `double` array of size `b.length` by `b.length` such that `covb` * $\sigma^2$ is the mean squared error matrix for `b`.
    * `rank` – An `int` scalar containing the rank of the variance-covariance matrix for all the observations.
    * `sumOfSquaress` – A `double` scalar containing the generalized sum of squares.
    * `logDeterminant` – A `double` scalar containing the natural log of the product of the nonzero eigenvalues of P where `P` * $\sigma^2$ is the variance-covariance matrix of the observations.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the dimensions of `b`, and `covb` are not consistent.

## Methods

- *filter*
  `public final void` **filter**`( )`

  - **Description**
    Performs Kalman filtering and evaluates the likelihood function for the state-space model.

---

- *getCovB*
  `public double[]` **getCovB**`( )`

  - **Description**
    Returns the mean squared error matrix for `b` divided by sigma squared.
  - **Returns** – a `double` array of size `b.length` by `b.length` such that `covb` * $\sigma^2$ is the mean squared error matrix for `b`.

---

- *getCovV*
  `public double[][]` **getCovV**`( )`

  - **Description**
    Returns the variance-covariance matrix of $v$ divided by sigma squared.
  - **Returns** – a `double` matrix containing a `y.length by y.length` matrix such that `covv` * $\sigma^2$ is the variance-covariance matrix of the one-step-ahead prediction error, `getPredictionError`.

---

- *getLogDeterminant*
  `public double` **getLogDeterminant**`( )`

  - **Description**
    Returns the natural log of the product of the nonzero eigenvalues of $P$ where $P$ * $sigma^2$ is the variance-covariance matrix of the observations.
  - **Returns** – a `double` scalar containing the natural log of the product of the nonzero eigenvalues of P where `P` * $\sigma^2$ is the variance-covariance matrix of the observations. In the usual case when P is nonsingular, `logDeterminant` is the natural log of the determinant of P.

---

- *getPredictionError*
  `public double[]` **getPredictionError**`( )`

  - **Description**
    Returns the one-step-ahead prediction error.
  - **Returns** – a `double` array of size `y.length` containing the one-step-ahead prediction error.

---

- *getRank*
  `public int` **getRank**`( )`

– **Description**
Returns the rank of the variance-covariance matrix for all the observations.

– **Returns** – An `int` scalar containing the rank of the variance-covariance matrix for all the observations.

---

- *getStateVector*
`public double[] getStateVector( )`

  – **Description**
  Returns the estimated state vector at time k + 1 given the observations through time k.

  – **Returns** – a `double` array containing the estimated state vector at time k + 1 given the observations through time k.

---

- *getSumOfSquares*
`public double getSumOfSquares( )`

  – **Description**
  Returns the generalized sum of squares.

  – **Returns** – a `double` scalar containing the generalized sum of squares. The estimate of $\sigma^2$ is given by `sumOfSquares / rank`.

---

- *setQ*
`public void setQ( double[][] q )`

  – **Description**
  Sets the Q matrix.

  – **Parameters**
    * `q` – A `double` matrix containing the `b.length by b.length` matrix such that `q` * $\sigma^2$ is the variance-covariance matrix of the error vector in the state equation. Default: There is no error term in the state equation.

---

- *setTolerance*
`public void setTolerance( double tolerance )`

  – **Description**
  Sets the tolerance used in determining linear dependence.

  – **Parameters**
    * `tolerance` – A `double` scalar containing the tolerance used in determining linear dependence. Default: `tolerance` = 100.0*2.2204460492503131e-16.

---

- *setTransitionMatrix*
`public void setTransitionMatrix( double[][] t )`

– **Description**
Sets the transition matrix.

– **Parameters**

* `t` – A `double` matrix containing the `b.length by b.length` transition matrix in the state equation. Default: `t` = identity matrix

---

- *update*
`public void` **update**`( double[] y, double[][] z, double[][] r )`

– **Description**
Performs computation of the update equations.

– **Parameters**

* `y` – A `double` array containing the observations.
* `z` – A `double` matrix containing the `y.length by b.length` matrix relating the observations to the state vector in the observation equation.
* `r` – A `double` matrix containing the `y.length by y.length` matrix such that `r` * $\sigma^2$ is the variance-covariance matrix of errors in the observation equation. $\sigma^2$ is a positive unknown scalar. Only elements in the upper triangle of `r` are referenced.

### Example: Kalman Filter

KalmanFilter is used to compute the filtered estimates and one-step-ahead estimates for a scalar problem discussed by Harvey (1981, pages 116-117). The observation equation and state equation are given by

$$y_k = b_k + e_k$$

$$b_{k+1} = b_k + w_{k+1}$$

*k = 1, 2, 3, 4*

where the $e_k$s are identically and independently distributed normal with mean 0 and variance $\sigma^2$, the $w_k$s are identically and independently distributed normal with mean 0 and variance $4\sigma^2$, and $b_1$ is distributed normal with mean 4 and variance $16\sigma^2$. Two KalmanFilter objects are needed for each time point in order to compute the filtered estimate and the one-step-ahead estimate. The first object does not use the methods `SetTransitionMatrix` and `setQ` so that the prediction equations are skipped in the computations. The update equations are skipped in the computations in the second object.

This example also computes the one-step-ahead prediction errors. Harvey (1981, page 117) contains a misprint for the value $v_4$ that he gives as 1.197. The correct value of $v_4 = 1.003$ is computed by KalmanFilter.

```
import java.text.*;
import com.imsl.stat.*;
import java.text.MessageFormat;

public class KalmanFilterEx1 {
    static private final MessageFormat mf =
        new MessageFormat("{0}/{1}\t{2}\t{3}\t{4}\t{5}\t{6}\t{7}\t{8}");

    public static void main(String args[]) {
        int nobs = 4;
        int rank = 0;
        double logDeterminant = 0.0;
        double ss = 0.0;
        double[] b = {4};
        double[] covb = {16};
        double[][] q = {{4}};
        double[][] r = {{1}};
        double[][] t = {{1}};
        double[][] z = {{1}};
        double[] ydata = {4.4, 4.0, 3.5, 4.6};

        Object argFormat[] =
            {"k", "j", "b", "cov(b)", "rank", "ss", "ln(det)", "v", "cov(v)"};
        System.out.println(mf.format(argFormat));

        for (int i = 0;  i < nobs;  i++) {
            double y[] = {ydata[i]};
            KalmanFilter kalman =
            new KalmanFilter(b, covb, rank, ss, logDeterminant);
            kalman.update(y, z, r);
            kalman.filter();
            b = kalman.getStateVector();
            covb = kalman.getCovB();
            rank = kalman.getRank();
            ss = kalman.getSumOfSquares();
            logDeterminant = kalman.getLogDeterminant();
            double v[] = kalman.getPredictionError();
            double covv[][] = kalman.getCovV();
            argFormat[0] = new Integer(i);
```

```
            argFormat[1] = new Integer(i);
            argFormat[2] = new Double(b[0]);
            argFormat[3] = new Double(covb[0]);
            argFormat[4] = new Integer(rank);
            argFormat[5] = new Double(ss);
            argFormat[6] = new Double(logDeterminant);
            argFormat[7] = new Double(v[0]);
            argFormat[8] = new Double(covv[0][0]);
            System.out.println(mf.format(argFormat));

            kalman = new KalmanFilter(b, covb, rank, ss, logDeterminant);
            kalman.setTransitionMatrix(t);
            kalman.setQ(q);
            kalman.filter();
            b = kalman.getStateVector();
            covb = kalman.getCovB();
            rank = kalman.getRank();
            ss = kalman.getSumOfSquares();
            logDeterminant = kalman.getLogDeterminant();
            argFormat[0] = new Integer(i+1);
            argFormat[1] = new Integer(i);
            argFormat[2] = new Double(b[0]);
            argFormat[3] = new Double(covb[0]);
            argFormat[4] = new Integer(rank);
            argFormat[5] = new Double(ss);
            argFormat[6] = new Double(logDeterminant);
            argFormat[7] = new Double(v[0]);
            argFormat[8] = new Double(covv[0][0]);
            System.out.println(mf.format(argFormat));
        }
    }
}
```

## Output

```
k/j b cov(b) rank ss ln(det) v cov(v)
0/0 4.376 0.941 1 0.009 2.833 0.4 17
1/0 4.376 4.941 1 0.009 2.833 0.4 17
1/1 4.063 0.832 2 0.033 4.615 -0.376 5.941
2/1 4.063 4.832 2 0.033 4.615 -0.376 5.941
2/2 3.597 0.829 3 0.088 6.378 -0.563 5.832
```

```
3/2 3.597 4.829 3 0.088 6.378 -0.563 5.832
3/3 4.428 0.828 4 0.26 8.141 1.003 5.829
4/3 4.428 4.828 4 0.26 8.141 1.003 5.829
```

# Chapter 19

# Multivariate Analysis

Classes



## Usage Notes

### Cluster Analysis

`ClusterKMeans` performs a K-means cluster analysis. Basic K-means clustering attempts to find a clustering that minimizes the within-cluster sums-of-squares. In this method of clustering the data, matrix X is grouped so that each observation (row in X) is assigned to one of a fixed number, K, of clusters. The sum of the squared difference of each observation about its assigned cluster's mean is used as the criterion for assignment. In the basic algorithm, observations are transferred from one cluster or another when doing so decreases the within-cluster sums-of-squared differences. When no transfer occurs in a pass through the entire data set, the algorithm stops. `ClusterKMeans` is one implementation of the basic algorithm.

The usual course of events in K-means cluster analysis is to use `ClusterKMeans` to obtain the optimal clustering. The clustering is then evaluated by functions described in "Basic Statistics," and/or other chapters in this manual. Often, K-means clustering with more than one value of K is performed, and the value of K that best fits the data is used.

Clustering can be performed either on observations or variables. The discussion of the function `ClusterKMeans` assumes the clustering is to be performed on the observations, which correspond to the rows of the input data matrix. If variables, rather than observations, are to be clustered, the data matrix should first be transposed. In the documentation for `ClusterKMeans`, the words "observation" and "variable" are interchangeable.

**Principal Components**

The idea in principal components is to find a small number of linear combinations of the original variables that maximize the variance accounted for in the original data. This amounts to an eigensystem analysis of the covariance (or correlation) matrix. In addition to the eigensystem analysis, when the principal component model is used, `FactorAnalysis` computes standard errors for the eigenvalues. Correlations of the original variables with the principal component scores also are computed.

**Factor Analysis**

Factor analysis and principal component analysis, while quite different in assumptions, often serve the same ends. Unlike principal components in which linear combinations yielding the highest possible variances are obtained, factor analysis generally obtains linear combinations of the observed variables according to a model relating the observed variable to hypothesized underlying factors, plus a random error term called the unique error or uniqueness. In factor analysis, the unique errors associated with each variable are usually assumed to be independent of the factors. Additionally, in the common factor model, the unique errors are assumed to be mutually independent. The factor analysis model is expressed in the following equation:

$$x - \mu = \Lambda f + e$$

where $x$ is the $p$ vector of observed values, $\mu$ is the $p$ vector of variable means, $\Lambda$ is the $p \times k$ matrix of factor loadings, $f$ is the $k$ vector of hypothesized underlying random factors, $e$ is the $p$ vector of hypothesized unique random errors, $p$ is the number of variables in the observed variables, and $k$ is the number of factors.

Because much of the computation in factor analysis was originally done by hand or was expensive on early computers, quick (but dirty) algorithms that made the calculations possible were developed. One result is the many factor extraction methods available today. Generally speaking, in the exploratory or model building phase of a factor analysis, a method of factor extraction that is not computationally intensive (such as principal components, principal factor, or image analysis) is used. If desired, a computationally intensive method is then used to obtain the final factors.

**Discriminant Analysis**

The class `DiscriminantAnalysis` allows linear or quadratic discrimination and the use of either reclassification, split sample, or the leaving-out-one methods in order to evaluate the rule. Moreover, `DiscriminantAnalysis` can be executed in an online mode, that is, one or more observations can be added to the rule during each invocation of `DiscriminantAnalysis`.

The mean vectors for each group of observations and an estimate of the common covariance matrix for all groups are input to `DiscriminantAnalysis`. These estimates can be computed via routine `DiscriminantAnalysis`. Output from `DiscriminantAnalysis` are linear combinations of the observations, which at most separate the groups. These linear combinations may subsequently be used for discriminating between the groups. Their use in graphically displaying differences between the groups is possibly more important, however.

# *class* ClusterKMeans

Perform a $K$-means (centroid) cluster analysis.

`ClusterKMeans` is an implementation of Algorithm AS 136 by Hartigan and Wong (1979). It computes $K$-means (centroid) Euclidean metric clusters for an input matrix starting with initial estimates of the $K$ cluster means. It allows for missing values (coded as NaN, *not a number*) and for weights and frequencies.

Let $p$ denote the number of variables to be used in computing the Euclidean distance between observations. The idea in $K$-means cluster analysis is to find a clustering (or grouping) of the observations so as to minimize the total within-cluster sums of squares. In this case, the total sums of squares within each cluster is computed as the sum of the centered sum of squares over all nonmissing values of each variable. That is,

$$\phi = \sum_{i=1}^{K} \sum_{j=1}^{p} \sum_{m=1}^{n_i} f_{\nu_{im}} w_{\nu_{im}} \delta_{\nu_{im},j} \left(x_{\nu_{im},j} - \bar{x}_{ij}\right)^2$$

where $\nu_{im}$ denotes the row index of the $m$-th observation in the $i$-th cluster in the matrix $X$; $n_i$ is the number of rows of $X$ assigned to group $i$; $f$ denotes the frequency of the observation; w denotes its weight; $d$ is zero if the $j$-th variable on observation $\nu_{im}$ is missing, otherwise $\delta$ is one; and $\bar{x}_{ij}$ is the average of the nonmissing observations for variable $j$ in group $i$. This method sequentially processes each observation and reassigns it to another cluster if doing so results in a decrease in the total within-cluster sums of squares. See Hartigan and Wong (1979) or Hartigan (1975) for details.

## Declaration

public class com.imsl.stat.ClusterKMeans
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* ClusterKMeans.NoConvergenceException

Convergence did not occur within the maximum number of iterations.

#### Declaration

public static class com.imsl.stat.ClusterKMeans.NoConvergenceException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *ClusterKMeans.NoConvergenceException*
  `public` **ClusterKMeans.NoConvergenceException**`( java.lang.String` **message** `)`
- *ClusterKMeans.NoConvergenceException*
  `public` **ClusterKMeans.NoConvergenceException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* ClusterKMeans.ClusterNoPointsException

There is a cluster with no points

#### Declaration

public static class com.imsl.stat.ClusterKMeans.ClusterNoPointsException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ClusterKMeans.ClusterNoPointsException*
  `public` **ClusterKMeans.ClusterNoPointsException**`( java.lang.String` **message** `)`

- *ClusterKMeans.ClusterNoPointsException*
  `public` **ClusterKMeans.ClusterNoPointsException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* ClusterKMeans.NonnegativeFreqException

Frequencies must be nonnegative.

### Declaration

public static class com.imsl.stat.ClusterKMeans.NonnegativeFreqException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ClusterKMeans.NonnegativeFreqException*
  `public` **ClusterKMeans.NonnegativeFreqException**`( java.lang.String` **message** `)`

- *ClusterKMeans.NonnegativeFreqException*
  `public` **ClusterKMeans.NonnegativeFreqException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* ClusterKMeans.NonnegativeWeightException

Weights must be nonnegative.

### Declaration

public static class com.imsl.stat.ClusterKMeans.NonnegativeWeightException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *ClusterKMeans.NonnegativeWeightException*
  `public` **ClusterKMeans.NonnegativeWeightException**`(` `java.lang.String` **message** `)`

- *ClusterKMeans.NonnegativeWeightException*
  `public` **ClusterKMeans.NonnegativeWeightException**`(` `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

### Constructor

---

- *ClusterKMeans*
  `public` **ClusterKMeans**`(` `double[][]` **x**`,` `double[][]` **cs** `)`
  - **Description**
    Constructor for `ClusterKMeans`.
  - **Parameters**
    * `x` – A `double` matrix containing the observations to be clustered.
    * `cs` – A `double` matrix containing the cluster seeds, i.e. estimates for the cluster centers.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `x.length`, `x[0].length` are equal 0, or `cs.length` is less than 1.

### Methods

---

- *compute*
  `public final double[][]` **compute**`( ) throws`
  `com.imsl.stat.ClusterKMeans.NoConvergenceException,`
  `com.imsl.stat.ClusterKMeans.ClusterNoPointsException`
  - **Description**
    Computes the cluster means.
  - **Returns** – A `double` matrix containing computed result.
  - **Throws**
    * `com.imsl.stat.ClusterKMeans.NonnegativeFreqException` – is thrown if a frequency is negative.

* `com.imsl.stat.ClusterKMeans.NonnegativeWeightException` – is thrown if a weight is negative.
* `com.imsl.stat.ClusterKMeans.NoConvergenceException` – is thrown if convergence did not occur within the maximum number of iterations.
* `com.imsl.stat.ClusterKMeans.ClusterNoPointsException` – is thrown if the cluster seed yields a cluster with no points.

---

- *getClusterCounts*
  `public int[]` **getClusterCounts**`( )`
  - **Description**
    Returns the number of observations in each cluster.
  - **Returns** – An `int` array containing the number of observations in each cluster.

---

- *getClusterMembership*
  `public int[]` **getClusterMembership**`( )`
  - **Description**
    Returns the cluster membership for each observation.
  - **Returns** – An `int` array containing the cluster membership for each observation. Cluster membership 1 indicates the observation belongs to cluster 1, cluster membership 2 indicates the observation belongs to cluster 2, etc.

---

- *getClusterSSQ*
  `public double[]` **getClusterSSQ**`( )`
  - **Description**
    Returns the within sum of squares for each cluster.
  - **Returns** – A `double` array containing the within sum of squares for each cluster.

---

- *setFrequencies*
  `public void` **setFrequencies**`( double[]` **frequencies** `) throws com.imsl.stat.ClusterKMeans.NonnegativeFreqException`
  - **Description**
    Sets the frequency for each observation.
  - **Parameters**
    * `frequencies` – A `double` array of size `x.length` containing the frequency for each observation. Default: `frequencies[]` = 1.

---

- *setMaxIterations*
  `public void` **setMaxIterations**`( int` **iterations** `)`

– **Description**
Sets the maximum number of iterations.

– **Parameters**

* `iterations` – An `int` scalar specifying the maximum number of iterations. Default: `interations` = 30.

---

- *setWeights*
`public void` **setWeights**`( double[]` **weights** `) throws com.imsl.stat.ClusterKMeans.NonnegativeWeightException`

– **Description**
Sets the weight for each observation.

– **Parameters**

* `weights` – A `double` array of size `x.length` containing the weight for each observation. Default: `weights[]` = 1.

### Example: K-means Cluster Analysis

This example performs K-means cluster analysis on Fisher's iris data. The initial cluster seed for each iris type is an observation known to be in the iris type.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.*;

public class ClusterKMeansEx1 {
    public static void main(String argv[]) throws Exception {

        double[][] x = {
                        { 5.100, 3.500, 1.400, 0.200},
                        { 4.900, 3.000, 1.400, 0.200},
                        { 4.700, 3.200, 1.300, 0.200},
                        { 4.600, 3.100, 1.500, 0.200},
                        { 5.000, 3.600, 1.400, 0.200},
                        { 5.400, 3.900, 1.700, 0.400},
                        { 4.600, 3.400, 1.400, 0.300},
                        { 5.000, 3.400, 1.500, 0.200},
                        { 4.400, 2.900, 1.400, 0.200},
                        { 4.900, 3.100, 1.500, 0.100},
                        { 5.400, 3.700, 1.500, 0.200},
                        { 4.800, 3.400, 1.600, 0.200},
```

```
{ 4.800, 3.000, 1.400, 0.100},
{ 4.300, 3.000, 1.100, 0.100},
{ 5.800, 4.000, 1.200, 0.200},
{ 5.700, 4.400, 1.500, 0.400},
{ 5.400, 3.900, 1.300, 0.400},
{ 5.100, 3.500, 1.400, 0.300},
{ 5.700, 3.800, 1.700, 0.300},
{ 5.100, 3.800, 1.500, 0.300},
{ 5.400, 3.400, 1.700, 0.200},
{ 5.100, 3.700, 1.500, 0.400},
{ 4.600, 3.600, 1.000, 0.200},
{ 5.100, 3.300, 1.700, 0.500},
{ 4.800, 3.400, 1.900, 0.200},
{ 5.000, 3.000, 1.600, 0.200},
{ 5.000, 3.400, 1.600, 0.400},
{ 5.200, 3.500, 1.500, 0.200},
{ 5.200, 3.400, 1.400, 0.200},
{ 4.700, 3.200, 1.600, 0.200},
{ 4.800, 3.100, 1.600, 0.200},
{ 5.400, 3.400, 1.500, 0.400},
{ 5.200, 4.100, 1.500, 0.100},
{ 5.500, 4.200, 1.400, 0.200},
{ 4.900, 3.100, 1.500, 0.200},
{ 5.000, 3.200, 1.200, 0.200},
{ 5.500, 3.500, 1.300, 0.200},
{ 4.900, 3.600, 1.400, 0.100},
{ 4.400, 3.000, 1.300, 0.200},
{ 5.100, 3.400, 1.500, 0.200},
{ 5.000, 3.500, 1.300, 0.300},
{ 4.500, 2.300, 1.300, 0.300},
{ 4.400, 3.200, 1.300, 0.200},
{ 5.000, 3.500, 1.600, 0.600},
{ 5.100, 3.800, 1.900, 0.400},
{ 4.800, 3.000, 1.400, 0.300},
{ 5.100, 3.800, 1.600, 0.200},
{ 4.600, 3.200, 1.400, 0.200},
{ 5.300, 3.700, 1.500, 0.200},
{ 5.000, 3.300, 1.400, 0.200},
{ 7.000, 3.200, 4.700, 1.400},
{ 6.400, 3.200, 4.500, 1.500},
{ 6.900, 3.100, 4.900, 1.500},
{ 5.500, 2.300, 4.000, 1.300},
```

```
{ 6.500, 2.800, 4.600, 1.500},
{ 5.700, 2.800, 4.500, 1.300},
{ 6.300, 3.300, 4.700, 1.600},
{ 4.900, 2.400, 3.300, 1.000},
{ 6.600, 2.900, 4.600, 1.300},
{ 5.200, 2.700, 3.900, 1.400},
{ 5.000, 2.000, 3.500, 1.000},
{ 5.900, 3.000, 4.200, 1.500},
{ 6.000, 2.200, 4.000, 1.000},
{ 6.100, 2.900, 4.700, 1.400},
{ 5.600, 2.900, 3.600, 1.300},
{ 6.700, 3.100, 4.400, 1.400},
{ 5.600, 3.000, 4.500, 1.500},
{ 5.800, 2.700, 4.100, 1.000},
{ 6.200, 2.200, 4.500, 1.500},
{ 5.600, 2.500, 3.900, 1.100},
{ 5.900, 3.200, 4.800, 1.800},
{ 6.100, 2.800, 4.000, 1.300},
{ 6.300, 2.500, 4.900, 1.500},
{ 6.100, 2.800, 4.700, 1.200},
{ 6.400, 2.900, 4.300, 1.300},
{ 6.600, 3.000, 4.400, 1.400},
{ 6.800, 2.800, 4.800, 1.400},
{ 6.700, 3.000, 5.000, 1.700},
{ 6.000, 2.900, 4.500, 1.500},
{ 5.700, 2.600, 3.500, 1.000},
{ 5.500, 2.400, 3.800, 1.100},
{ 5.500, 2.400, 3.700, 1.000},
{ 5.800, 2.700, 3.900, 1.200},
{ 6.000, 2.700, 5.100, 1.600},
{ 5.400, 3.000, 4.500, 1.500},
{ 6.000, 3.400, 4.500, 1.600},
{ 6.700, 3.100, 4.700, 1.500},
{ 6.300, 2.300, 4.400, 1.300},
{ 5.600, 3.000, 4.100, 1.300},
{ 5.500, 2.500, 4.000, 1.300},
{ 5.500, 2.600, 4.400, 1.200},
{ 6.100, 3.000, 4.600, 1.400},
{ 5.800, 2.600, 4.000, 1.200},
{ 5.000, 2.300, 3.300, 1.000},
{ 5.600, 2.700, 4.200, 1.300},
{ 5.700, 3.000, 4.200, 1.200},
```

```
{ 5.700, 2.900, 4.200, 1.300},
{ 6.200, 2.900, 4.300, 1.300},
{ 5.100, 2.500, 3.000, 1.100},
{ 5.700, 2.800, 4.100, 1.300},
{ 6.300, 3.300, 6.000, 2.500},
{ 5.800, 2.700, 5.100, 1.900},
{ 7.100, 3.000, 5.900, 2.100},
{ 6.300, 2.900, 5.600, 1.800},
{ 6.500, 3.000, 5.800, 2.200},
{ 7.600, 3.000, 6.600, 2.100},
{ 4.900, 2.500, 4.500, 1.700},
{ 7.300, 2.900, 6.300, 1.800},
{ 6.700, 2.500, 5.800, 1.800},
{ 7.200, 3.600, 6.100, 2.500},
{ 6.500, 3.200, 5.100, 2.000},
{ 6.400, 2.700, 5.300, 1.900},
{ 6.800, 3.000, 5.500, 2.100},
{ 5.700, 2.500, 5.000, 2.000},
{ 5.800, 2.800, 5.100, 2.400},
{ 6.400, 3.200, 5.300, 2.300},
{ 6.500, 3.000, 5.500, 1.800},
{ 7.700, 3.800, 6.700, 2.200},
{ 7.700, 2.600, 6.900, 2.300},
{ 6.000, 2.200, 5.000, 1.500},
{ 6.900, 3.200, 5.700, 2.300},
{ 5.600, 2.800, 4.900, 2.000},
{ 7.700, 2.800, 6.700, 2.000},
{ 6.300, 2.700, 4.900, 1.800},
{ 6.700, 3.300, 5.700, 2.100},
{ 7.200, 3.200, 6.000, 1.800},
{ 6.200, 2.800, 4.800, 1.800},
{ 6.100, 3.000, 4.900, 1.800},
{ 6.400, 2.800, 5.600, 2.100},
{ 7.200, 3.000, 5.800, 1.600},
{ 7.400, 2.800, 6.100, 1.900},
{ 7.900, 3.800, 6.400, 2.000},
{ 6.400, 2.800, 5.600, 2.200},
{ 6.300, 2.800, 5.100, 1.500},
{ 6.100, 2.600, 5.600, 1.400},
{ 7.700, 3.000, 6.100, 2.300},
{ 6.300, 3.400, 5.600, 2.400},
{ 6.400, 3.100, 5.500, 1.800},
```

```
                    { 6.000, 3.000, 4.800, 1.800},
                    { 6.900, 3.100, 5.400, 2.100},
                    { 6.700, 3.100, 5.600, 2.400},
                    { 6.900, 3.100, 5.100, 2.300},
                    { 5.800, 2.700, 5.100, 1.900},
                    { 6.800, 3.200, 5.900, 2.300},
                    { 6.700, 3.300, 5.700, 2.500},
                    { 6.700, 3.000, 5.200, 2.300},
                    { 6.300, 2.500, 5.000, 1.900},
                    { 6.500, 3.000, 5.200, 2.000},
                    { 6.200, 3.400, 5.400, 2.300},
                    { 5.900, 3.000, 5.100, 1.800}};


        double[][] cs = {{ 5.100, 3.500, 1.400, 0.200},
                         { 7.000, 3.200, 4.700, 1.400},
                         { 6.300, 3.300, 6.000, 2.500}};


        ClusterKMeans kmean = new ClusterKMeans(x, cs);

        double[][] cm = kmean.compute();
        double[] wss = kmean.getClusterSSQ();
        int[] ic = kmean.getClusterMembership();
        int[] nc = kmean.getClusterCounts();

        PrintMatrix pm = new PrintMatrix ("Cluster Means");

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(4);
        pmf.setNumberFormat(nf);
        pm.print (pmf, cm);

        new PrintMatrix("Cluster Membership").print(ic);
        new PrintMatrix("Sum of Squares").print(wss);
        new PrintMatrix("Number of observations").print(nc);


    }
}
```

### Output

```
          Cluster Means
     0       1       2       3
0  5.0060  3.4280  1.4620  0.2460
1  5.9016  2.7484  4.3935  1.4339
2  6.8500  3.0737  5.7421  2.0711

Cluster Membership
     0
  0  1
  1  1
  2  1
  3  1
  4  1
  5  1
  6  1
  7  1
  8  1
  9  1
 10  1
 11  1
 12  1
 13  1
 14  1
 15  1
 16  1
 17  1
 18  1
 19  1
 20  1
 21  1
 22  1
 23  1
 24  1
```

```
25  1
26  1
27  1
28  1
29  1
30  1
31  1
32  1
33  1
34  1
35  1
36  1
37  1
38  1
39  1
40  1
41  1
42  1
43  1
44  1
45  1
46  1
47  1
48  1
49  1
50  2
51  2
52  3
53  2
54  2
55  2
56  2
57  2
58  2
59  2
60  2
61  2
62  2
63  2
64  2
65  2
66  2
```

```
 67   2
 68   2
 69   2
 70   2
 71   2
 72   2
 73   2
 74   2
 75   2
 76   2
 77   3
 78   2
 79   2
 80   2
 81   2
 82   2
 83   2
 84   2
 85   2
 86   2
 87   2
 88   2
 89   2
 90   2
 91   2
 92   2
 93   2
 94   2
 95   2
 96   2
 97   2
 98   2
 99   2
100   3
101   2
102   3
103   3
104   3
105   3
106   2
107   3
108   3
```

```
109   3
110   3
111   3
112   3
113   2
114   2
115   3
116   3
117   3
118   3
119   2
120   3
121   2
122   3
123   2
124   3
125   3
126   2
127   2
128   3
129   3
130   3
131   3
132   3
133   2
134   3
135   3
136   3
137   3
138   2
139   3
140   3
141   3
142   2
143   3
144   3
145   3
146   2
147   3
148   3
149   2
```

```
Sum of Squares
     0
0  15.151
1  39.821
2  23.879

Number of observations
   0
0  50
1  62
2  38
```

## *class* **Dissimilarities**

Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.

Class `Dissimilarities` computes an upper triangular matrix (excluding the diagonal) of dissimilarities (or similarities) between the columns or rows of a matrix. Nine different distance measures can be computed. For the first three measures, three different scaling options can be employed. The distance matrix computed is generally used as input to clustering or multidimensional scaling functions.

The following discussion assumes that the distance measure is being computed between the columns of the matrix. If distances between the rows of the matrix are desired, set `iRow` to 1 when calling the `Dissimilarities` constructor.

For `distanceMethod` = 0 to 2, each row of `x` is first scaled according to the value of `distanceScale`. The scaling parameters are obtained from the values in the row scaled as either the standard deviation of the row or the row range; the standard deviation is computed from the unbiased estimate of the variance. If `distanceScale` is 0, no scaling is performed, and the parameters in the following discussion are all 1.0. Once the scaling value (if any) has been computed, the distance between column $i$ and column $j$ is computed via the difference vector $z_k = \frac{(x_k - y_k)}{s_k}, i = 1, \ldots, ndstm$, where $x_k$ denotes the $k$-th element in the $i$-th column, and $y_k$ denotes the corresponding element in the $j$-th column. For given $z_i$, the metrics 0 to 2 are defined as:

| `distanceMethod` | **Metric** |
|---|---|
| 0 | Euclidean distance ($L_2$norm) |
| 1 | Sum of the absolute differences ($L_1$ norm) |
| 2 | Maximum difference ($L_\infty$ norm) |

Distance measures corresponding to `distanceMethod` = 3 to 8 do not allow for scaling.

| `distanceMethod` | **Metric** |
|---|---|
| 3 | Mahalanobis distance |
| 4 | Absolute value of the cosine of the angle between the vectors |
| 5 | Angle in radians (0, pi) between the lines through the origin defined by the vectors |
| 6 | Correlation coefficient |
| 7 | Absolute value of the correlation coefficient |
| 8 | Number of exact matches, where $x_i = y_i$. |

For the Mahalanobis distance, any variable used in computing the distance measure that is (numerically) linearly dependent upon the previous variables in the `indexArray` vector is omitted from the distance measure.

## Declaration

public class com.imsl.stat.Dissimilarities
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* Dissimilarities.ScaleFactorZeroException

The computations cannot continue because a scale factor is zero.

#### Declaration

public static class com.imsl.stat.Dissimilarities.ScaleFactorZeroException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *Dissimilarities.ScaleFactorZeroException*
  `public` **Dissimilarities.ScaleFactorZeroException**`( int` **index** `)`
  - **Description**
    Constructs a `ScaleFactorZeroException`.

– **Parameters**
  * `index` – An `int` which specifies the index of the scale factor array at which scale factor is zero.

### *class* Dissimilarities.ZeroNormException

The computations cannot continue because the Euclidean norm of the column is equal to zero.

#### Declaration

public static class com.imsl.stat.Dissimilarities.ZeroNormException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *Dissimilarities.ZeroNormException*
  `public` **Dissimilarities.ZeroNormException**`( int` **index** `)`
  – **Description**
    Constructs a `ZeroNormException`.
  – **Parameters**
    * `index` – An `int` which specifies the column index for which the norm has been found to be zero.

### *class* Dissimilarities.NoPositiveVarianceException

No variable has positive variance. The Mahalanobis distances cannot be computed.

#### Declaration

public static class com.imsl.stat.Dissimilarities.NoPositiveVarianceException
**extends** com.imsl.IMSLException (page 1240)

#### Constructor

---

- *Dissimilarities.NoPositiveVarianceException*
  `public` **Dissimilarities.NoPositiveVarianceException**`( )`
  - **Description**
    Constructs a `NoPositiveVarianceException`.

## Constructors

- *Dissimilarities*
  `public` **Dissimilarities**`( double[][] x, int` **distanceMethod**`, int` **distanceScale**`, int` **iRow** `) throws com.imsl.stat.Dissimilarities.ScaleFactorZeroException, com.imsl.stat.Dissimilarities.ZeroNormException, com.imsl.stat.Dissimilarities.NoPositiveVarianceException`
  - **Description**
    Constructor for `Dissimilarities`.
  - **Parameters**
    * `x` – A `double` matrix containing the data input matrix.
    * `distanceMethod` – An `int` identifying the method to be used in computing the dissimilarities or similarities. Acceptable values of `distanceMethod` are 1, 2, ..., 8. See above for a description of these methods.
    * `distanceScale` – An `int` containing the scaling option.

| `distanceScale` | **Method** |
|---|---|
| 0 | No scaling is performed. |
| 1 | Scale each column (row if `iRow=1`) by the standard deviation of the column (row). |
| 2 | Scale each column (row if `iRow=1`) by the range of the column (row). |

    * `iRow` – An `int` identifying whether distances are computed between rows or columns of `x`. If `iRow` = 1, distances are computed between the rows of `x`. Otherwise, distances between the columns of `x` are computed.
  - **Throws**
    * `java.lang.IllegalArgumentException` – thrown when the row lengths of input matrix a are not equal (i.e. the matrix edges are "jagged")
    * `com.imsl.stat.Dissimilarities.ScaleFactorZeroException` – thrown when computations cannot continue because a scale factor is zero
    * `com.imsl.stat.Dissimilarities.NoPositiveVarianceException` – thrown when no variable has positive variance

* com.imsl.stat.Dissimilarities.ZeroNormException – is thrown when the Euclidean norm of a column is equal to zero

---

- *Dissimilarities*

  public **Dissimilarities**( double[][] **x**, int **distanceMethod**, int **distanceScale**, int **iRow**, int[] **indexArray** ) throws com.imsl.stat.Dissimilarities.ScaleFactorZeroException, com.imsl.stat.Dissimilarities.ZeroNormException, com.imsl.stat.Dissimilarities.NoPositiveVarianceException

  - **Description**

    Constructor for Dissimilarities.
  - **Parameters**
    * x – A double matrix containing the data input matrix.
    * distanceMethod – An int identifying the method to be used in computing the dissimilarities or similarities. Acceptable values of distanceMethod are 1, 2, ..., 8. See above for a description of these methods.
    * distanceScale – An int containing the scaling option.

| distanceScale | **Method** |
|---|---|
| 0 | No scaling is performed |
| 1 | Scale each column (row if iRow=1) by the standard deviation of the column (row). |
| 2 | Scale each column (row if iRow=1) by the range of the column (row) |

    * iRow – An int identifying whether distances are computed between rows or columns of x. If iRow=1, distances are computed between the rows of x. Otherwise, distances between the columns of x are computed.
    * indexArray – An int array containing the indices of the rows (columns if iRow is 1) to be used in computing the distance measure.
  - **Throws**
    * java.lang.IllegalArgumentException – thrown when the row lengths of input matrix a are not equal (i.e. the matrix edges are "jagged")
    * com.imsl.stat.Dissimilarities.ScaleFactorZeroException – thrown when computations cannot continue because a scale factor is zero
    * com.imsl.stat.Dissimilarities.NoPositiveVarianceException – thrown when no variable has positive variance.
    * com.imsl.stat.Dissimilarities.ZeroNormException – is thrown when the Euclidean norm of a column is equal to zero

## Method

- *getDistanceMatrix*
  `public final double[][]` **getDistanceMatrix**`( )`
  - **Description**
    Returns the distance matrix.
  - **Returns** – A `double` matrix containing the distance matrix.

### Example: Dissimilarities

The following example illustrates the use of Dissimilarities for computing the Euclidean distance between the rows of a matrix:

```
import java.io.*;
import com.imsl.stat.*;
import com.imsl.math.*;

public class DissimilaritiesEx1 {
    public static void main(String argv[]) throws Exception {
        double[][] x = {
            { 1., 1.},
            { 1., 0.},
            { 1., -1.},
            { 1., 2.}};
        int distanceMethod = 0;
        int distanceScale = 0;
        int iRow = 1;

            Dissimilarities dist =
               new Dissimilarities(x, distanceMethod, distanceScale, iRow);
            double[][] distanceMatrix = dist.getDistanceMatrix();

            for (int i=0;i<distanceMatrix.length;i++){
                for (int j=0;j<distanceMatrix[0].length;j++)
                    System.out.print(distanceMatrix[i][j]+", ");
                System.out.println();
            }
    }
}
```

## Output

```
0.0, 1.0, 2.0, 1.0,
0.0, 0.0, 1.0, 2.0,
0.0, 0.0, 0.0, 3.0,
0.0, 0.0, 0.0, 0.0,
```

# *class* ClusterHierarchical

Performs a hierarchical cluster analysis from a distance matrix.

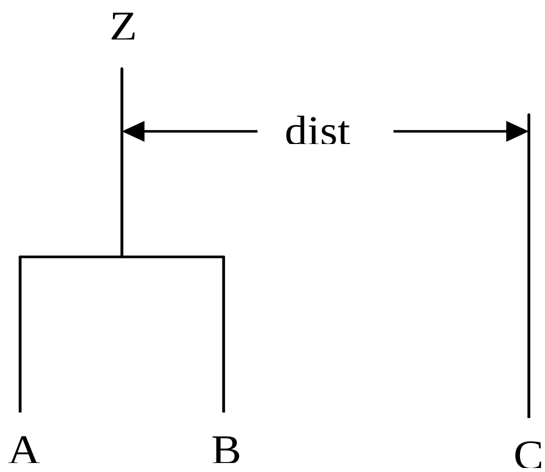
Class `ClusterHierarchical` conducts a hierarchical cluster analysis based upon a distance matrix, or by appropriate use of the argument `transform`, based upon a similarity matrix. Only the upper triangular part of the `dist` matrix is required as input.

Hierarchical clustering in `ClusterHierarchical` proceeds as follows:

Initially, each data point is considered to be a cluster, numbered 1 to n = *npt*, where *npt* is the number of rows in `dist`.

1. If the data matrix contains similarities, they are converted to distances by the method specified by the argument `transform`. Set $k = 1$.
2. A search is made of the distance matrix to find the two closest clusters. These clusters are merged to form a new cluster, numbered $n + k$. The cluster numbers of the two clusters joined at this stage are saved as *Right Sons* and *Left Sons*, and the distance measure between the two clusters is stored as *Cluster Level*.
3. Based upon the method of clustering, updating of the distance measure in the row and column of `dist` corresponding to the new cluster is performed.
4. Set $k = k + 1$. If $k$ is less than $n$, go to Step 2.

The five methods differ primarily in how the distance matrix is updated after two clusters have been joined. The argument `method` specifies how the distance of the cluster just merged with each of the remaining clusters will be updated. Class `ClusterHierarchical` allows five methods for computing the distances. To understand these measures, suppose in the following discussion that clusters $A$ and $B$ have just been joined to form cluster $Z$, and interest is in computing the distance of $Z$ with another cluster called $C$.


Z
dist
A
B
C

| `method` | **Description** |
|---|---|
| 0 | Single linkage (minimum distance). The distance from $Z$ to $C$ is the minimum of the distances ($A$ to $C$, $B$ to $C$). |
| 1 | Complete linkage (maximum distance). The distance from $Z$ to $C$ is the maximum of the distances ($A$ to $C$, $B$ to $C$). |
| 2 | Average-distance-within-clusters method. The distance from $Z$ to $C$ is the average distance of all objects that would be within the cluster formed by merging clusters $Z$ and $C$. This average may be computed according to formulas given by Anderberg (1973, page 139). |
| 3 | Average-distance-between-clusters method. The distance from $Z$ to $C$ is the average distance of objects within cluster $Z$ to objects within cluster $C$. This average may be computed according to methods given by Anderberg (1973, page 140). |
| 4 | Ward's method: Clusters are formed so as to minimize the increase in the within-cluster sums of squares. The distance between two clusters is the increase in these sums of squares if the two clusters were merged. A method for computing this distance from a squared Euclidean distance matrix is given by Anderberg (1973, pages 142-145). |

In general, single linkage will yield long thin clusters while complete linkage will yield clusters that are more spherical. Average linkage and Ward's linkage tend to yield clusters that are similar to those obtained with complete linkage.

Function Class `ClusterHierarchical` produces a unique representation of the binary cluster tree via the following three conventions; the fact that the tree is unique should aid in interpreting the clusters. First, when two clusters are joined and each cluster contains two or more data points, the cluster that was initially formed with the smallest level becomes the left son. Second, when a cluster containing more than one data point is joined with a cluster containing a single data point, the cluster with the single data point becomes the right son. Finally, when two clusters containing only one object are joined, the cluster with the smallest cluster number becomes the right son.

## Comments

1. The clusters corresponding to the original data points are numbered from 1 to *npt*, where *npt* is the number of rows in `dist`. The *npt* - 1 clusters formed by merging clusters are numbered *npt* + 1 to *npt* + (*npt* - 1 ).
2. Raw correlations, if used as similarities, should be made positive and transformed to a distance measure. One such transformation can be performed by setting argument `transform`, with `transform` = 2.
3. The user may cluster either variables or observations with `ClusterHierarchical` since a dissimilarity matrix, not the original data, is used. Class com.imsl.stat.Dissimilarities may be used to compute the matrix `dist` for either the variables or observations.

## Declaration

public class com.imsl.stat.ClusterHierarchical
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Constructor

- *ClusterHierarchical*
  `public` **ClusterHierarchical**`(` `double[][]` **dist**`,` `int` **method**`,` `int` **transform** `)`
  - **Description**
    Constructor for `ClusterHierarchical`.
  - **Parameters**
    - `dist` – A `double` symmetric matrix containing the distance (or similarity) matrix. On input, only the upper triangular part needs to be present. `ClusterHierarchical` saves the upper triangular part of `dist` in the lower triangle. On return, the upper triangular part of `dist` is restored, and the matrix is made symmetric.
    - `method` – An `int` identifying the clustering method to be used.

| method | Description |
|---|---|
| 0 | Single linkage (minimum distance). |
| 1 | Complete linkage (maximum distance). |
| 2 | Average distance within (average distance between objects within the merged cluster). |
| 3 | Average distance between (average distance between objects in the two clusters). |
| 4 | Ward's method (minimize the within-cluster sums of squares). For Ward's method, the elements of `dist` are assumed to be Euclidean distances. |

* `transform` – An `int` identifying the type of transformation applied to the measures in `dist`.

| transform | Description |
|---|---|
| 0 | No transformation is required. The elements of `dist` are distances. |
| 1 | Convert similarities to distances by multiplication by -1.0. |
| 2 | Convert similarities (usually correlations) to distances by taking the reciprocal of the absolute value. |

– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown when the row lengths of input matrix `a` are not equal (i.e. the matrix edges are "jagged")

## Methods

---

• *getClusterLeftSons*

`public final int[] `**getClusterLeftSons**`( )`

– **Description**
Returns the left sons of each merged cluster.

– **Returns** – An `int` array containing the left sons of each merged cluster.

• *getClusterLevel*

`public final double[] `**getClusterLevel**`( )`

– **Description**
Returns the level at which the clusters are joined.

– **Returns** – A `double` array containing the level at which the clusters are joined. Element [$k$-1] contains the distance (or similarity) level at which cluster $npt + k$ was formed. If the original data in `dist` was transformed, the inverse transformation is applied to the returned values.

---

• *getClusterMembership*
`public final int[]` **getClusterMembership**`( int` **nClusters** `)`

– **Description**
Returns the cluster membership of each observation.

– **Parameters**
  ∗ `nClusters` – An `int` which specifies the desired number of clusters.

– **Returns** – An `int` array containing the cluster membership of each observation.

---

• *getClusterRightSons*
`public final int[]` **getClusterRightSons**`( )`

– **Description**
Returns the right sons of each merged cluster.

– **Returns** – An `int` array containing the right sons of each merged cluster.

---

• *getObsPerCluster*
`public final int[]` **getObsPerCluster**`( int` **nClusters** `)`

– **Description**
Returns the number of observations in each cluster.

– **Parameters**
  ∗ `nClusters` – An `int` which specifies the desired number of clusters.

– **Returns** – An `int` array containing the number of observations in each cluster.

### Example 1: ClusterHierarchical

This example illustrates a typical usage of `ClusterHierarchical`. The Fisher iris data is clustered. First the distance between irises is computed using the class `Dissimilarities`. The resulting distance matrix is then clustered using `ClusterHierarchical`, and cluster memberships for 5 clusters are computed.

```
import java.io.*;
import com.imsl.stat.*;
import com.imsl.math.*;
```

```
public class ClusterHierarchicalEx1 {
    public static void main(String argv[]) throws Exception {
        double[][] irisData = {
            { 5.1, 3.5, 1.4, .2},
            { 4.9, 3.0, 1.4, .2},
            { 4.7, 3.2, 1.3, .2},
            { 4.6, 3.1, 1.5, .2},
            { 5.0, 3.6, 1.4, .2},
            { 5.4, 3.9, 1.7, .4},
            { 4.6, 3.4, 1.4, .3},
            { 5.0, 3.4, 1.5, .2},
            { 4.4, 2.9, 1.4, .2},
            { 4.9, 3.1, 1.5, .1},
            { 5.4, 3.7, 1.5, .2},
            { 4.8, 3.4, 1.6, .2},
            { 4.8, 3.0, 1.4, .1},
            { 4.3, 3.0, 1.1, .1},
            { 5.8, 4.0, 1.2, .2},
            { 5.7, 4.4, 1.5, .4},
            { 5.4, 3.9, 1.3, .4},
            { 5.1, 3.5, 1.4, .3},
            { 5.7, 3.8, 1.7, .3},
            { 5.1, 3.8, 1.5, .3},
            { 5.4, 3.4, 1.7, .2},
            { 5.1, 3.7, 1.5, .4},
            { 4.6, 3.6, 1.0, .2},
            { 5.1, 3.3, 1.7, .5},
            { 4.8, 3.4, 1.9, .2},
            { 5.0, 3.0, 1.6, .2},
            { 5.0, 3.4, 1.6, .4},
            { 5.2, 3.5, 1.5, .2},
            { 5.2, 3.4, 1.4, .2},
            { 4.7, 3.2, 1.6, .2},
            { 4.8, 3.1, 1.6, .2},
            { 5.4, 3.4, 1.5, .4},
            { 5.2, 4.1, 1.5, .1},
            { 5.5, 4.2, 1.4, .2},
            { 4.9, 3.1, 1.5, .2},
            { 5.0, 3.2, 1.2, .2},
            { 5.5, 3.5, 1.3, .2},
            { 4.9, 3.6, 1.4, .1},
```

```
{ 4.4, 3.0, 1.3, .2},
{ 5.1, 3.4, 1.5, .2},
{ 5.0, 3.5, 1.3, .3},
{ 4.5, 2.3, 1.3, .3},
{ 4.4, 3.2, 1.3, .2},
{ 5.0, 3.5, 1.6, .6},
{ 5.1, 3.8, 1.9, .4},
{ 4.8, 3.0, 1.4, .3},
{ 5.1, 3.8, 1.6, .2},
{ 4.6, 3.2, 1.4, .2},
{ 5.3, 3.7, 1.5, .2},
{ 5.0, 3.3, 1.4, .2},
{ 7.0, 3.2, 4.7, 1.4},
{ 6.4, 3.2, 4.5, 1.5},
{ 6.9, 3.1, 4.9, 1.5},
{ 5.5, 2.3, 4.0, 1.3},
{ 6.5, 2.8, 4.6, 1.5},
{ 5.7, 2.8, 4.5, 1.3},
{ 6.3, 3.3, 4.7, 1.6},
{ 4.9, 2.4, 3.3, 1.0},
{ 6.6, 2.9, 4.6, 1.3},
{ 5.2, 2.7, 3.9, 1.4},
{ 5.0, 2.0, 3.5, 1.0},
{ 5.9, 3.0, 4.2, 1.5},
{ 6.0, 2.2, 4.0, 1.0},
{ 6.1, 2.9, 4.7, 1.4},
{ 5.6, 2.9, 3.6, 1.3},
{ 6.7, 3.1, 4.4, 1.4},
{ 5.6, 3.0, 4.5, 1.5},
{ 5.8, 2.7, 4.1, 1.0},
{ 6.2, 2.2, 4.5, 1.5},
{ 5.6, 2.5, 3.9, 1.1},
{ 5.9, 3.2, 4.8, 1.8},
{ 6.1, 2.8, 4.0, 1.3},
{ 6.3, 2.5, 4.9, 1.5},
{ 6.1, 2.8, 4.7, 1.2},
{ 6.4, 2.9, 4.3, 1.3},
{ 6.6, 3.0, 4.4, 1.4},
{ 6.8, 2.8, 4.8, 1.4},
{ 6.7, 3.0, 5.0, 1.7},
{ 6.0, 2.9, 4.5, 1.5},
{ 5.7, 2.6, 3.5, 1.0},
```

```
{ 5.5, 2.4, 3.8, 1.1},
{ 5.5, 2.4, 3.7, 1.0},
{ 5.8, 2.7, 3.9, 1.2},
{ 6.0, 2.7, 5.1, 1.6},
{ 5.4, 3.0, 4.5, 1.5},
{ 6.0, 3.4, 4.5, 1.6},
{ 6.7, 3.1, 4.7, 1.5},
{ 6.3, 2.3, 4.4, 1.3},
{ 5.6, 3.0, 4.1, 1.3},
{ 5.5, 2.5, 4.0, 1.3},
{ 5.5, 2.6, 4.4, 1.2},
{ 6.1, 3.0, 4.6, 1.4},
{ 5.8, 2.6, 4.0, 1.2},
{ 5.0, 2.3, 3.3, 1.0},
{ 5.6, 2.7, 4.2, 1.3},
{ 5.7, 3.0, 4.2, 1.2},
{ 5.7, 2.9, 4.2, 1.3},
{ 6.2, 2.9, 4.3, 1.3},
{ 5.1, 2.5, 3.0, 1.1},
{ 5.7, 2.8, 4.1, 1.3},
{ 6.3, 3.3, 6.0, 2.5},
{ 5.8, 2.7, 5.1, 1.9},
{ 7.1, 3.0, 5.9, 2.1},
{ 6.3, 2.9, 5.6, 1.8},
{ 6.5, 3.0, 5.8, 2.2},
{ 7.6, 3.0, 6.6, 2.1},
{ 4.9, 2.5, 4.5, 1.7},
{ 7.3, 2.9, 6.3, 1.8},
{ 6.7, 2.5, 5.8, 1.8},
{ 7.2, 3.6, 6.1, 2.5},
{ 6.5, 3.2, 5.1, 2.0},
{ 6.4, 2.7, 5.3, 1.9},
{ 6.8, 3.0, 5.5, 2.1},
{ 5.7, 2.5, 5.0, 2.0},
{ 5.8, 2.8, 5.1, 2.4},
{ 6.4, 3.2, 5.3, 2.3},
{ 6.5, 3.0, 5.5, 1.8},
{ 7.7, 3.8, 6.7, 2.2},
{ 7.7, 2.6, 6.9, 2.3},
{ 6.0, 2.2, 5.0, 1.5},
{ 6.9, 3.2, 5.7, 2.3},
{ 5.6, 2.8, 4.9, 2.0},
```

```
{ 7.7, 2.8, 6.7, 2.0},
{ 6.3, 2.7, 4.9, 1.8},
{ 6.7, 3.3, 5.7, 2.1},
{ 7.2, 3.2, 6.0, 1.8},
{ 6.2, 2.8, 4.8, 1.8},
{ 6.1, 3.0, 4.9, 1.8},
{ 6.4, 2.8, 5.6, 2.1},
{ 7.2, 3.0, 5.8, 1.6},
{ 7.4, 2.8, 6.1, 1.9},
{ 7.9, 3.8, 6.4, 2.0},
{ 6.4, 2.8, 5.6, 2.2},
{ 6.3, 2.8, 5.1, 1.5},
{ 6.1, 2.6, 5.6, 1.4},
{ 7.7, 3.0, 6.1, 2.3},
{ 6.3, 3.4, 5.6, 2.4},
{ 6.4, 3.1, 5.5, 1.8},
{ 6.0, 3.0, 4.8, 1.8},
{ 6.9, 3.1, 5.4, 2.1},
{ 6.7, 3.1, 5.6, 2.4},
{ 6.9, 3.1, 5.1, 2.3},
{ 5.8, 2.7, 5.1, 1.9},
{ 6.8, 3.2, 5.9, 2.3},
{ 6.7, 3.3, 5.7, 2.5},
{ 6.7, 3.0, 5.2, 2.3},
{ 6.3, 2.5, 5.0, 1.9},
{ 6.5, 3.0, 5.2, 2.0},
{ 6.2, 3.4, 5.4, 2.3},
{ 5.9, 3.0, 5.1, 1.8}};

Dissimilarities dist = new Dissimilarities(irisData, 0, 1, 1);
double[][] distanceMatrix = dist.getDistanceMatrix();
ClusterHierarchical clink = new ClusterHierarchical(
    dist.getDistanceMatrix(),2,0);

int nClusters = 5;
int[] iclus = clink.getClusterMembership(nClusters);
int[] nclus = clink.getObsPerCluster(nClusters);
System.out.println("Cluster Membership");
for (int i=0;i<15;i++){
    for (int j=0;j<10;j++)
        System.out.print(iclus[i*10+j]+" ");
    System.out.println();
```

```
        }

        System.out.println("Observations Per Cluster");
        for (int i=0;i<nClusters;i++)
            System.out.print(nclus[i]+" ");
        System.out.println();
    }
}
```

### Output

```
Cluster Membership
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
5 5 5 5 5 5 5 5 5 5
3 3 3 4 3 4 3 4 3 4
4 3 4 3 4 3 4 4 4 4
3 3 3 3 3 3 3 3 3 4
4 4 4 3 4 3 3 4 4 4
4 3 4 4 4 4 4 3 4 4
2 3 2 3 2 1 4 1 3 2
2 3 2 3 3 2 3 2 1 4
2 3 1 3 2 1 3 3 3 1
1 2 3 3 3 1 2 3 3 2
2 2 3 2 2 2 3 3 2 3
Observations Per Cluster
8 19 44 29 50
```

## *class* **FactorAnalysis**

Performs Principal Component Analysis or Factor Analysis on a covariance or correlation matrix.

Class `FactorAnalysis` computes principal components or initial factor loading estimates for a variance-covariance or correlation matrix using exploratory factor analysis models.

Models available are the principal component model for factor analysis and the common factor model with additions to the common factor model in alpha factor analysis and image analysis. Methods of estimation include principal components, principal factor, image analysis, unweighted least squares, generalized least squares, and maximum likelihood.

For the principal component model there are methods to compute the characteristic roots, characteristic vectors, standard errors for the characteristic roots, and the correlations of the principal component scores with the original variables. Principal components obtained from correlation matrices are the same as principal components obtained from standardized (to unit variance) variables.

The principal component scores are the elements of the vector $y = \Gamma^T x$ where $\Gamma$ is the matrix whose columns are the characteristic vectors (eigenvectors) of the sample covariance (or correlation) matrix and $x$ is the vector of observed (or standardized) random variables. The variances of the principal component scores are the characteristic roots (eigenvalues) of the covariance (correlation) matrix.

Asymptotic variances for the characteristic roots were first obtained by Girshick (1939) and are given more recently by Kendall, Stuart, and Ord (1983, page 331). These variances are computed either for variance-covariance matrices or for correlation matrices.

The correlations of the principal components with the observed (or standardized) variables are the same as the unrotated factor loadings obtained for the principal components model for factor analysis when a correlation matrix is input.

In the factor analysis model used for factor extraction, the basic model is given as $\Sigma = \Lambda\Lambda^T + \Psi$ where $\Sigma$ is the $p \times p$ population covariance matrix. $\Lambda$ is the $p \times k$ matrix of factor loadings relating the factors $f$ to the observed variables $x$, and $\Psi$ is the $p \times p$ matrix of covariances of the unique errors $e$. Here, $p$ represents the number of variables and $k$ is the number of factors. The relationship between the factors, the unique errors, and the observed variables is given as $x = \Lambda f + e$ where, in addition, it is assumed that the expected values of $e$, $f$, and $x$ are zero. (The sample means can be subtracted from $x$ if the expected value of $x$ is not zero.) It is also assumed that each factor has unit variance, the factors are independent of each other, and that the factors and the unique errors are mutually independent. In the common factor model, the elements of the vector of unique errors $e$ are also assumed to be independent of one another so that the matrix $\Psi$ is diagonal. This is not the case in the principal component model in which the errors may be correlated.

Further differences between the various methods concern the criterion that is optimized and the amount of computer effort required to obtain estimates. Generally speaking, the least-squares and maximum likelihood methods, which use iterative algorithms, require the most computer time with the principal factor, principal component, and the image methods requiring much less time since the algorithms in these methods are not iterative. The algorithm in alpha factor analysis is also iterative, but the estimates in this method generally require somewhat less computer effort than the least-squares and maximum likelihood estimates. In all algorithms one eigensystem analysis is required on each iteration.

## Declaration

public class com.imsl.stat.FactorAnalysis
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* FactorAnalysis.RankException

Rank of covariance matrix error.

#### Declaration

public static class com.imsl.stat.FactorAnalysis.RankException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *FactorAnalysis.RankException*
  `public` **FactorAnalysis.RankException**`( java.lang.String` **message** `)`
- *FactorAnalysis.RankException*
  `public` **FactorAnalysis.RankException**`( java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

### *class* FactorAnalysis.NotPositiveSemiDefiniteException

Covariance matrix not positive semi-definite.

#### Declaration

public static class com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *FactorAnalysis.NotPositiveSemiDefiniteException*
  `public` **FactorAnalysis.NotPositiveSemiDefiniteException**`(`
  `java.lang.String` **message** `)`

- *FactorAnalysis.NotPositiveSemiDefiniteException*
  `public` **FactorAnalysis.NotPositiveSemiDefiniteException**`(`
  `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

## *class* FactorAnalysis.NotSemiDefiniteException

Hessian matrix not semi-definite.

### Declaration

public static class com.imsl.stat.FactorAnalysis.NotSemiDefiniteException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *FactorAnalysis.NotSemiDefiniteException*
  `public` **FactorAnalysis.NotSemiDefiniteException**`(` `java.lang.String`
  **message** `)`

- *FactorAnalysis.NotSemiDefiniteException*
  `public` **FactorAnalysis.NotSemiDefiniteException**`(` `java.lang.String` **key**`,`
  `java.lang.Object[]` **arguments** `)`

## *class* FactorAnalysis.NotPositiveDefiniteException

Covariance matrix not positive definite.

### Declaration

public static class com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

- *FactorAnalysis.NotPositiveDefiniteException*
  `public` **FactorAnalysis.NotPositiveDefiniteException**`(` `java.lang.String` **message** `)`
- *FactorAnalysis.NotPositiveDefiniteException*
  `public` **FactorAnalysis.NotPositiveDefiniteException**`(` `java.lang.String` **key**, `java.lang.Object[]` **arguments** `)`

## *class* FactorAnalysis.SingularException

Covariance matrix singular error.

### Declaration

public static class com.imsl.stat.FactorAnalysis.SingularException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

- *FactorAnalysis.SingularException*
  `public` **FactorAnalysis.SingularException**`(` `java.lang.String` **message** `)`
- *FactorAnalysis.SingularException*
  `public` **FactorAnalysis.SingularException**`(` `java.lang.String` **key**, `java.lang.Object[]` **arguments** `)`

## *class* FactorAnalysis.BadVarianceException

Bad variance error.

### Declaration

public static class com.imsl.stat.FactorAnalysis.BadVarianceException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *FactorAnalysis.BadVarianceException*
  `public` **FactorAnalysis.BadVarianceException**`( java.lang.String` **message** `)`

---

- *FactorAnalysis.BadVarianceException*
  `public` **FactorAnalysis.BadVarianceException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* **FactorAnalysis.EigenvalueException**

Eigenvalue error.

### Declaration

public static class com.imsl.stat.FactorAnalysis.EigenvalueException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *FactorAnalysis.EigenvalueException*
  `public` **FactorAnalysis.EigenvalueException**`( java.lang.String` **message** `)`

---

- *FactorAnalysis.EigenvalueException*
  `public` **FactorAnalysis.EigenvalueException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

## *class* **FactorAnalysis.NonPositiveEigenvalueException**

Non positive eigenvalue error.

### Declaration

public static class com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *FactorAnalysis.NonPositiveEigenvalueException*
  `public` **FactorAnalysis.NonPositiveEigenvalueException**`(`
  `java.lang.String` **message** `)`

- *FactorAnalysis.NonPositiveEigenvalueException*
  `public` **FactorAnalysis.NonPositiveEigenvalueException**`(`
  `java.lang.String` **key**`,` `java.lang.Object[]` **arguments** `)`

## *class* FactorAnalysis.NoDegreesOfFreedomException

No degrees of freedom error.

### Declaration

public static class com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

---

- *FactorAnalysis.NoDegreesOfFreedomException*
  `public` **FactorAnalysis.NoDegreesOfFreedomException**`(` `java.lang.String`
  **message** `)`

- *FactorAnalysis.NoDegreesOfFreedomException*
  `public` **FactorAnalysis.NoDegreesOfFreedomException**`(` `java.lang.String`
  **key**`,` `java.lang.Object[]` **arguments** `)`

## Fields

---

- public static final `int` **VARIANCE_COVARIANCE_MATRIX**
  - Indicates variance-covariance matrix.
- public static final `int` **CORRELATION_MATRIX**
  - Indicates correlation matrix.

- public static final `int` **PRINCIPAL_COMPONENT_MODEL**
  - Indicates principal component model.
- public static final `int` **PRINCIPAL_FACTOR_MODEL**
  - Indicates principal factor model.
- public static final `int` **UNWEIGHTED_LEAST_SQUARES**
  - Indicates unweighted least squares method.
- public static final `int` **GENERALIZED_LEAST_SQUARES**
  - Indicates generalized least squares method.
- public static final `int` **MAXIMUM_LIKELIHOOD**
  - Indicates maximum likelihood method.
- public static final `int` **IMAGE_FACTOR_ANALYSIS**
  - Indicates image factor analysis.
- public static final `int` **ALPHA_FACTOR_ANALYSIS**
  - Indicates alpha factor analysis.

## Constructor

---

- *FactorAnalysis*
  `public` **FactorAnalysis**`( double[][]` **cov**`, int` **matrixType**`, int` **nf** `)`
  - **Description**
    Constructor for `FactorAnalysis`.
  - **Parameters**
    * `cov` – A `double` matrix containing the covariance or correlation matrix.
    * `matrixType` – An `int` scalar indicating the type of matrix that is input. Uses class member `VARIANCE_COVARIANCE_MATRIX`, `CORRELATION_MATRIX` for `matrixType`.
    * `nf` – An `int` scalar indicating the number of factors in the model. If `nf` is not known in advance, several different values of `nf` should be used, and the most reasonable value kept in the final solution. Since, in practice, the non-iterative methods often lead to solutions which differ little from the iterative methods, it is usually suggested that a non-iterative method be used in the initial stages of the factor analysis, and that the iterative methods be used once issues such as the number of factors have been resolved.

– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if `x.length`, and `x[0].length` are equal to 0.

## Methods

---

- *getCorrelations*

  `public double[][]` **getCorrelations**`( ) throws com.imsl.stat.FactorAnalysis.RankException, com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException, com.imsl.stat.FactorAnalysis.NotSemiDefiniteException, com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException, com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException, com.imsl.stat.FactorAnalysis.SingularException, com.imsl.stat.FactorAnalysis.BadVarianceException, com.imsl.stat.FactorAnalysis.EigenvalueException, com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException`

  – **Description**
    Returns the correlations of the principal components.
  – **Returns** – An `double` matrix containing the correlations of the principal components with the observed/standardized variables. If a covariance matrix is input to the constructor, then the correlations are with the observed variables. Otherwise, the correlations are with the standardized (to a variance of 1.0) variables. Only valid for the principal components model.

---

- *getFactorLoadings*

  `public double[][]` **getFactorLoadings**`( ) throws com.imsl.stat.FactorAnalysis.RankException, com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException, com.imsl.stat.FactorAnalysis.NotSemiDefiniteException, com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException, com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException, com.imsl.stat.FactorAnalysis.SingularException, com.imsl.stat.FactorAnalysis.BadVarianceException, com.imsl.stat.FactorAnalysis.EigenvalueException, com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException`

  – **Description**
    Returns the unrotated factor loadings.
  – **Returns** – A `double` matrix containing the unrotated factor loadings.

- *getParameterUpdates*
  public double[] **getParameterUpdates**( ) throws
  com.imsl.stat.FactorAnalysis.RankException,
  com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,
  com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,
  com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,
  com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,
  com.imsl.stat.FactorAnalysis.SingularException,
  com.imsl.stat.FactorAnalysis.BadVarianceException,
  com.imsl.stat.FactorAnalysis.EigenvalueException,
  com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException

  - **Description**
    Returns the parameter updates.
  - **Returns** – A `double` array containing the parameter updates when convergence was reached (or the iterations terminated). The parameter updates are only meaningful for the common factor model. The parameter updates are set to 0.0 for the principal component model.

- *getPercents*
  public double[] **getPercents**( ) throws
  com.imsl.stat.FactorAnalysis.RankException,
  com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,
  com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,
  com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,
  com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,
  com.imsl.stat.FactorAnalysis.SingularException,
  com.imsl.stat.FactorAnalysis.BadVarianceException,
  com.imsl.stat.FactorAnalysis.EigenvalueException,
  com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException

  - **Description**
    Returns the cumulative percent of the total variance explained by each principal component. Valid for the principal component model.
  - **Returns** – An `double` array containing the total variance explained by each principal component.

- *getStandardErrors*
  public double[] **getStandardErrors**( ) throws
  com.imsl.stat.FactorAnalysis.RankException,
  com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,
  com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,

com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,
com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,
com.imsl.stat.FactorAnalysis.SingularException,
com.imsl.stat.FactorAnalysis.BadVarianceException,
com.imsl.stat.FactorAnalysis.EigenvalueException,
com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException

- **Description**
  Returns the estimated asymptotic standard errors of the eigenvalues.
- **Returns** – An `double` array containing the estimated asymptotic standard errors of the eigenvalues.

---

- *getStatistics*
  `public double[]` **getStatistics**`( ) throws`
  `com.imsl.stat.FactorAnalysis.RankException,`
  `com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,`
  `com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,`
  `com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,`
  `com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,`
  `com.imsl.stat.FactorAnalysis.SingularException,`
  `com.imsl.stat.FactorAnalysis.BadVarianceException,`
  `com.imsl.stat.FactorAnalysis.EigenvalueException,`
  `com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException`

  - **Description**
    Returns statistics.
  - **Returns** – A `double` array (Stat) containing output statistics. Stat is not defined and is set to NaN when the method used to obtain the estimates, is the principal component method, principal factor method, image factor analysis method, or alpha analysis method.

| *i* | *Stat[i]* |
|---|---|
| 0 | Value of the function minimum. |
| 1 | Tucker reliability coefficient. |
| 2 | Chi-squared test statistic for testing that the number of factors in the model are adequate for the data. |
| 3 | Degrees of freedom in chi-squared. This is computed as $((nvar - nf)^2 - nvar - nf)/2$where `nvar` is the number of variables and `nf` is the number of factors in the model. |
| 4 | Probability of a greater chi-squared statistic. |
| 5 | Number of iterations. |

---

- *getValues*

public double[] **getValues**( ) throws
com.imsl.stat.FactorAnalysis.RankException,
com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,
com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,
com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,
com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,
com.imsl.stat.FactorAnalysis.SingularException,
com.imsl.stat.FactorAnalysis.BadVarianceException,
com.imsl.stat.FactorAnalysis.EigenvalueException,
com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException

  - **Description**
    Returns the eigenvalues.
  - **Returns** – A `double` array containing the eigenvalues of the matrix from which the factors were extracted ordered from largest to smallest. If Alpha Factor analysis is used, then the first `nf` positions of the array contain the Alpha coefficients. Here, `nf` is the number of factors in the model. If the algorithm fails to converge for a particular eigenvalue, that eigenvalue is set to NaN. Note that the eigenvalues are usually not the eigenvalues of the input matrix cov. They are the eigenvalues of the input matrix cov when the `principal component` method is used.

---

- *getVariances*

public double[] **getVariances**( ) throws
com.imsl.stat.FactorAnalysis.RankException,
com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,

```
com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,
com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,
com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,
com.imsl.stat.FactorAnalysis.SingularException,
com.imsl.stat.FactorAnalysis.BadVarianceException,
com.imsl.stat.FactorAnalysis.EigenvalueException,
com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException
```

- **Description**
  Gets the unique variances.
- **Returns** – A `double` array of length nvar containing the unique variances, where `nvar` is the number of variables.

---

- *getVectors*
  `public double[][]` **getVectors**`( ) throws`
  ```
  com.imsl.stat.FactorAnalysis.RankException,
  com.imsl.stat.FactorAnalysis.NoDegreesOfFreedomException,
  com.imsl.stat.FactorAnalysis.NotSemiDefiniteException,
  com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException,
  com.imsl.stat.FactorAnalysis.NotPositiveDefiniteException,
  com.imsl.stat.FactorAnalysis.SingularException,
  com.imsl.stat.FactorAnalysis.BadVarianceException,
  com.imsl.stat.FactorAnalysis.EigenvalueException,
  com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException
  ```

  - **Description**
    Returns the eigenvectors.
  - **Returns** – A `double` matrix containing the eigenvectors of the matrix from which the factors were extracted. The j-th column of the eigenvector matrix corresponds to the j-th eigenvalue. The eigenvectors are normalized to each have Euclidean length equal to one. Also, the sign of each vector is set so that the largest component in magnitude (the first of the largest if there are ties) is made positive. Note that the eigenvectors are usually not the eigenvectors of the input matrix cov. They are the eigenvectors of the input matrix cov when the `principal component` method is used.

---

- *setConvergenceCriterion1*
  `public void` **setConvergenceCriterion1**`( double` **eps** `)`

  - **Description**
    Sets the convergence criterion used to terminate the iterations.
  - **Parameters**

* eps – A double used to terminate the iterations. For the least squares and and maximum likelihood methods convergence is assumed when the relative change in the criterion is less than eps. For alpha factor analysis, convergence is assumed when the maximum change (relative to the variance) of a uniqueness is less than eps. eps is not referenced for the other estimation methods. If this member function is not called, eps is set to 0.0001.

---

- *setConvergenceCriterion2*
`public void` **setConvergenceCriterion2**`( double` **epse** `)`
  - **Description**
    Sets the convergence criterion used to switch to exact second derivatives.
  - **Parameters**
    * epse – A double used to switch to exact second derivatives. When the largest relative change in the unique standard deviation vector is less than epse exact second derivative vectors are used. If this member function is not called, epse is set to 0.1. Not referenced for principal component, principal factor, image factor, or alpha factor methods.

---

- *setDegreesOfFreedom*
`public void` **setDegreesOfFreedom**`( int` **ndf** `)`
  - **Description**
    Sets the number of degrees of freedom.
  - **Parameters**
    * ndf – An int value specifying the number of degrees of freedom in the input matrix. If this member function is not called 100 degrees of freedom are assumed.

---

- *setFactorLoadingEstimationMethod*
`public void` **setFactorLoadingEstimationMethod**`( int` **methodType** `)`
  - **Description**
    Sets the factor loading estimation method.
  - **Parameters**
    * methodType – An int scalar indicating the method to be applied for obtaining the factor loadings. Use class member PRINCIPAL_COMPONENT_MODEL, PRINCIPAL_FACTOR_MODEL, UNWEIGHTED_LEAST_SQUARES, GENERALIZED_LEAST_SQUARES, MAXIMUM_LIKELIHOOD, IMAGE_FACTOR_ANALYSIS, or ALPHA_FACTOR_ANALYSIS for methodType. If this member function is not called, the PRINCIPAL_COMPONENT_MODEL is used.

For the principal component and principal factor methods, the factor loading estimates are computed as

$$\hat{\Gamma}\hat{\Delta}^{-1/2}$$

where $\Gamma$ and the diagonal matrix $\Delta$ are the eigenvalues and eigenvectors of a matrix. In the principal component model, the eigensystem analysis is performed on the sample covariance (correlation) matrix $S$ while in the principal factor model the matrix $(S - \Psi)$ is used. If the unique error variances $\Psi$ are not known in the principal factor model, then they are estimated. This is achieved by calling the member function `setVarianceEstimationMethod` and setting `init` to 0. If the principal component model is used, the error variances are set to 0.0 automatically. The basic idea in the principal component method is to find factors that maximize the variance in the original data that is explained by the factors. Because this method allows the unique errors to be correlated, some factor analysts insist that the principal component method is not a factor analytic method. Usually however, the estimates obtained via the principal component model and other models in factor analysis will be quite similar. It should be noted that both the principal component and the principal factor methods give different results when the correlation matrix is used in place of the covariance matrix. Indeed, any rescaling of the sample covariance matrix can lead to different estimates with either of these methods. A further difficulty with the principal factor method is the problem of estimating the unique error variances. Theoretically, these must be known in advance and passed in through member function `setVariances`. In practice, the estimates of these parameters produced by calling the member function `setVarianceEstimationMethod` and setting `init` to 0 are often used. In either case, the resulting adjusted covariance (correlation) matrix

$$(S - \hat{\Psi})$$

may not yield the `nf` positive eigenvalues required for `nf` factors to be obtained. If this occurs, the user must either lower the number of factors to be estimated or give new unique error variance values.

For the least-squares and maximum likelihood methods an iterative algorithm is used to obtain the estimates (see Joreskog 1977). As with the principal factor model, the user may either input the initial unique error variances or allow the algorithm to compute initial estimates. Unlike the principal factor method, the code then optimizes the criterion function with respect to both $\Psi$ and $\Gamma$. (In the principal factor method, $\Psi$ is assumed to be known. Given $\Psi$, estimates for $\Lambda$ may be obtained.)

The major differences between the estimation methods described in this member function are in the criterion function that is optimized. Let $S$

denote the sample covariance (correlation) matrix, and let $\Sigma$ denote the covariance matrix that is to be estimated by the factor model. In the unweighted least-squares method, also called the iterated principal factor method or the minres method (see Harman 1976, page 177), the function minimized is the sum of the squared differences between $S$ and $\Sigma$. This is written as $\Phi_u l = .5 trace((S - \Sigma)^2)$.

Generalized least-squares and maximum likelihood estimates are asymptotically equivalent methods. Maximum likelihood estimates maximize the (normal theory) likelihood $\{\Phi_m l = trace(\Sigma^{-1}S) - log(|\Sigma^{-1}S|)\}$. while generalized least squares optimizes the function $\Phi_g s = trace(\Sigma S^{-1} - I)^2$.

In all three methods, a two-stage optimization procedure is used. This proceeds by first solving the likelihood equations for $\Lambda$ in terms of $\Psi$ and substituting the solution into the likelihood. This gives a criterion $\Phi(\Psi, \Lambda(\Psi))$, which is optimized with respect to $\Psi$. In the second stage, the estimates

$$\hat{\Lambda}$$

are obtained from the estimates for $\Psi$.

The generalized least-squares and the maximum likelihood methods allow for the computation of a statistic for testing that `nf` common factors are adequate to fit the model. This is a chi-squared test that all remaining parameters associated with additional factors are zero. If the probability of a larger chi-squared is small (see `stat[4]` under `getStatistics`) so that the null hypothesis is rejected, then additional factors are needed (although these factors may not be of any practical importance). Failure to reject does not legitimize the model. The statistic `stat[2]` is a likelihood ratio statistic in maximum likelihood estimates. As such, it asymptotically follows a chi-squared distribution with degrees of freedom given in `stat[3]`.

The Tucker and Lewis (1973) reliability coefficient, $\rho$, is returned in `stat[1]` when the maximum likelihood or generalized least-squares methods are used. This coefficient is an estimate of the ratio of explained to the total variation in the data. It is computed as follows:

$$\rho = \frac{mM_o - mM_k}{mM_o - 1}$$

$$m = d - \frac{2p+5}{6} - \frac{2k}{6}$$

$$M_o = \frac{-ln(|S|)}{p(p-1)/2}$$

$$M_k = \frac{\Phi}{((p-k)^2 - p - k)/2}$$

where $|S|$ is the determinant of `cov`, $p$ is the number of variables, $k$ is the number of factors, $\Phi$ is the optimized criterion, and $d$ is the number of degrees of freedom.

The term "image analysis" is used here to denote the noniterative image method of Kaiser (1963). It is not the image factor analysis discussed by Harman (1976, page 226). The image method (as well as the alpha factor analysis method) begins with the notion that only a finite number from an infinite number of possible variables have been measured. The image factor pattern is calculated under the assumption that the ratio of the number of factors to the number of observed variables is near zero so that a very good estimate for the unique error variances (for standardized variables) is given as one minus the squared multiple correlation of the variable under consideration with all variables in the covariance matrix.

First, the matrix $D^2 = (diag(S^{-1}))^{-1}$ is computed where the operator "diag" results in a matrix consisting of the diagonal elements of its argument, and $S$ is the sample covariance (correlation) matrix. Then, the eigenvalues $\Lambda$ and eigenvectors $\Gamma$ of the matrix $D^{-1}SD^{-1}$ are computed. Finally, the unrotated image factor pattern matrix is computed as $A = D\Gamma[(\Lambda - I)^2\Lambda^{-1}]^{1/2}$.

The alpha factor analysis method of Kaiser and Caffrey (1965) finds factor-loading estimates to maximize the correlation between the factors and the complete universe of variables of interest. The basic idea in this method is as follows: only a finite number of variables out of a much larger set of possible variables is observed. The population factors are linearly related to this larger set while the observed factors are linearly related to the observed variables. Let $f$ denote the factors obtainable from a finite set of observed random variables, and let $\xi$ denote the factors obtainable from the universe of observable variables. Then, the alpha method attempts to find factor-loading estimates so as to maximize the correlation between $f$ and $\xi$. In order to obtain these estimates, the iterative algorithm of Kaiser and Caffrey (1965) is used.

---

- *setMaxIterations*

  `public void` **`setMaxIterations(`** `int` **`maxit`** `)`

  - **Description**

    Sets the maximum number of iterations in the iterative procedure.

  - **Parameters**

    * `maxit` – An `int` used as the maximum number of iterations allowed during the iterative portion of the algorithm. If this member function is not called, `maxit` is set to 60. Not referenced for factor loading methods principal component, principal factor, or image factor methods.

---

- *setMaxStep*
  `public void` **setMaxStep**`( int` **maxstp** `)`
  - **Description**
    Sets the maximum number of step halvings allowed during an iteration.
  - **Parameters**
    * `maxstp` – An `int` used as the maximum number of step halvings allowed during an iteration. If this member function is not called, `maxstp` is set to 8. Not referenced for principal component, principal factor, image factor, or alpha factor methods.

---

- *setVarianceEstimationMethod*
  `public void` **setVarianceEstimationMethod**`( int` **init** `)`
  - **Description**
    Sets the variance estimation method.
  - **Parameters**
    * `init` – An `int` used to designate the method to be applied for obtaining the initial estimates
      of the unique variances. If this member function is not called, `init` is set to 1.

| *init* | *Method* |
|---|---|
| 0 | Initial estimates are taken as the constant 1-nf/(2*nvar) divided by the diagonal elements of the inverse of input matrix `cov`, where `nvar` is the number of variables. |
| 1 | Initial estimates are input by the user in vector `uniq (setVariances)`. |

Note that when the factor loading estimation method is PRINCIPAL_COMPONENT_MODEL, the initial estimates in `uniq` are reset to 0.0.

---

- *setVariances*
  `public void` **setVariances**`( double[]` **uniq** `)`
  - **Description**
    Sets the variances.
  - **Parameters**
    * `uniq` – A `double` array of length nvar containing the unique variances, where `nvar` is the number of variables. If this member function is not called, the elements of uniq are set to 0.0. If the iterative methods fail for the unique variances used, new initial estimates should be tried. These

may be obtained by use of another factoring method (use the final estimates from the new method as initial estimates in the old method). Another alternative is to call member function `setVarianceEstimationMethod` and set the input argument to 0. This will cause the initial unique variances to be estimated by the code.

### Example: Principal Components

This example illustrates the use of the FactorAnalysis class for a nine-variable matrix. The PRINCIPAL_COMPONENT_MODEL is selected and the input matrix type selected is a CORRELATION_MATRIX.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class FactorAnalysisEx1 {
    public static void main(String args[]) throws Exception {
                double[][] corr = {
            {1.0, 0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639},
            {0.523, 1.0, 0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645},
            {0.395, 0.479, 1.0, 0.355, 0.27, 0.254, 0.452, 0.219, 0.504},
            {0.471, 0.506, 0.355, 1.0, 0.691, 0.791, 0.443, 0.285, 0.505},
            {0.346, 0.418, 0.27, 0.691, 1.0, 0.679, 0.383, 0.149, 0.409},
            {0.426, 0.462, 0.254, 0.791, 0.679, 1.0, 0.372, 0.314, 0.472},
            {0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0, 0.385, 0.68},
            {0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0, 0.47},
            {0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68, 0.47, 1.0}
        };
        FactorAnalysis pc = new FactorAnalysis(corr, FactorAnalysis.CORRELATION_MATRIX, 9);
        pc.setFactorLoadingEstimationMethod(pc.PRINCIPAL_COMPONENT_MODEL);
        pc.setDegreesOfFreedom(100);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(4);
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        new PrintMatrix("Eigenvalues").print(pmf, pc.getValues());
        new PrintMatrix("Percents").print(pmf, pc.getPercents());
        new PrintMatrix("Standard Errors").print(pmf, pc.getStandardErrors());
        new PrintMatrix("Eigenvectors").print(pmf, pc.getVectors());
```

```
        new PrintMatrix("Unrotated Factor Loadings").print(pmf, pc.getFactorLoadings());
    }
}
```

### Output

```
Eigenvalues
     0
0  4.6769
1  1.2640
2  0.8444
3  0.5550
4  0.4471
5  0.4291
6  0.3102
7  0.2770
8  0.1962

 Percents
     0
0  0.5197
1  0.6601
2  0.7539
3  0.8156
4  0.8653
5  0.9130
6  0.9474
7  0.9782
8  1.0000

Standard Errors
     0
0  0.6498
1  0.1771
2  0.0986
3  0.0879
4  0.0882
5  0.0890
6  0.0944
7  0.0994
8  0.1113
```

Eigenvectors

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.3462 | -0.2354 | 0.1386 | -0.3317 | -0.1088 | 0.7974 | 0.1735 | -0.1240 | -0.0488 |
| 1 | 0.3526 | -0.1108 | -0.2795 | -0.2161 | 0.7664 | -0.2002 | 0.1386 | -0.3032 | -0.0079 |
| 2 | 0.2754 | -0.2697 | -0.5585 | 0.6939 | -0.1531 | 0.1511 | 0.0099 | -0.0406 | -0.0997 |
| 3 | 0.3664 | 0.4031 | 0.0406 | 0.1196 | 0.0017 | 0.1152 | -0.4022 | -0.1178 | 0.7060 |
| 4 | 0.3144 | 0.5022 | -0.0733 | -0.0207 | -0.2804 | -0.1796 | 0.7295 | 0.0075 | 0.0046 |
| 5 | 0.3455 | 0.4553 | 0.1825 | 0.1114 | 0.1202 | 0.0696 | -0.3742 | 0.0925 | -0.6780 |
| 6 | 0.3487 | -0.2714 | -0.0725 | -0.3545 | -0.5242 | -0.4355 | -0.2854 | -0.3408 | -0.1089 |
| 7 | 0.2407 | -0.3159 | 0.7383 | 0.4329 | 0.0861 | -0.1969 | 0.1862 | -0.1623 | 0.0505 |
| 8 | 0.3847 | -0.2533 | -0.0078 | -0.1468 | 0.0459 | -0.1498 | -0.0251 | 0.8521 | 0.1225 |

Unrotated Factor Loadings

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.7487 | -0.2646 | 0.1274 | -0.2471 | -0.0728 | 0.5224 | 0.0966 | -0.0652 | -0.0216 |
| 1 | 0.7625 | -0.1245 | -0.2568 | -0.1610 | 0.5124 | -0.1312 | 0.0772 | -0.1596 | -0.0035 |
| 2 | 0.5956 | -0.3032 | -0.5133 | 0.5170 | -0.1024 | 0.0990 | 0.0055 | -0.0214 | -0.0442 |
| 3 | 0.7923 | 0.4532 | 0.0373 | 0.0891 | 0.0012 | 0.0755 | -0.2240 | -0.0620 | 0.3127 |
| 4 | 0.6799 | 0.5646 | -0.0674 | -0.0154 | -0.1875 | -0.1177 | 0.4063 | 0.0039 | 0.0021 |
| 5 | 0.7472 | 0.5119 | 0.1677 | 0.0830 | 0.0804 | 0.0456 | -0.2084 | 0.0487 | -0.3003 |
| 6 | 0.7542 | -0.3051 | -0.0666 | -0.2641 | -0.3505 | -0.2853 | -0.1589 | -0.1794 | -0.0482 |
| 7 | 0.5206 | -0.3552 | 0.6784 | 0.3225 | 0.0576 | -0.1290 | 0.1037 | -0.0854 | 0.0224 |
| 8 | 0.8319 | -0.2848 | -0.0071 | -0.1094 | 0.0307 | -0.0981 | -0.0140 | 0.4485 | 0.0543 |

## Example: Factor Analysis

This example illustrates the use of the FactorAnalysis class. The following data were originally analyzed by Emmett(1949). There are 211 observations on 9 variables. Following Lawley and Maxwell (1971), three factors will be obtained by the method of maximum likelihood.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class FactorAnalysisEx2 {
    public static void main(String args[]) throws Exception {
```

```
            double[][] cov = {
        {1.0, 0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639},
        {0.523, 1.0, 0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645},
        {0.395, 0.479, 1.0, 0.355, 0.27, 0.254, 0.452, 0.219, 0.504},
        {0.471, 0.506, 0.355, 1.0, 0.691, 0.791, 0.443, 0.285, 0.505},
        {0.346, 0.418, 0.27, 0.691, 1.0, 0.679, 0.383, 0.149, 0.409},
        {0.426, 0.462, 0.254, 0.791, 0.679, 1.0, 0.372, 0.314, 0.472},
        {0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.0, 0.385, 0.68},
        {0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.0, 0.47},
        {0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.68, 0.47, 1.0}
    };
    FactorAnalysis fl =
     new FactorAnalysis(cov, FactorAnalysis.VARIANCE_COVARIANCE_MATRIX, 3);
    fl.setConvergenceCriterion1(.000001);
    fl.setConvergenceCriterion2(.01);
    fl.setFactorLoadingEstimationMethod(fl.MAXIMUM_LIKELIHOOD);
    fl.setVarianceEstimationMethod(0);
    fl.setMaxStep(10);
    fl.setDegreesOfFreedom(210);
    NumberFormat nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(4);
    PrintMatrixFormat pmf = new PrintMatrixFormat();
    pmf.setNumberFormat(nf);
    new PrintMatrix("Unique Error Variances").print
               (pmf, fl.getVariances());
    new PrintMatrix("Unrotated Factor Loadings").print
               (pmf, fl.getFactorLoadings());
    new PrintMatrix("Eigenvalues").print(pmf, fl.getValues());
    new PrintMatrix("Statistics").print(pmf, fl.getStatistics());

  }
}
```

## Output

```
Unique Error Variances
     0
0  0.4505
1  0.4271
2  0.6166
```

```
3  0.2123
4  0.3805
5  0.1769
6  0.3995
7  0.4615
8  0.2309

  Unrotated Factor Loadings
      0        1        2
0  0.6642  -0.3209   0.0735
1  0.6888  -0.2471  -0.1933
2  0.4926  -0.3022  -0.2224
3  0.8372   0.2924  -0.0354
4  0.7050   0.3148  -0.1528
5  0.8187   0.3767   0.1045
6  0.6615  -0.3960  -0.0777
7  0.4579  -0.2955   0.4913
8  0.7657  -0.4274  -0.0117

Eigenvalues
      0
0  0.0626
1  0.2295
2  0.5413
3  0.8650
4  0.8937
5  0.9736
6  1.0802
7  1.1172
8  1.1401

 Statistics
       0
0   0.0350
1   1.0000
2   7.1494
3  12.0000
4   0.8476
5   5.0000
```

# *class* DiscriminantAnalysis

Performs a linear or a quadratic discriminant function analysis among several known groups and the use of either reclassification, split sample, or the leaving-out-one methods in order to evaluate the rule.

Class `DiscriminantAnalysis` performs discriminant function analysis using either linear or quadratic discrimination. The output from `DiscriminantAnalysis` includes a measure of distance between the groups, a table summarizing the classification results, a matrix containing the posterior probabilities of group membership for each observation, and the within-sample means and covariance matrices. The linear discriminant function coefficients are also computed.

All observations are input during one call to `DiscriminantAnalysis`, a method of operation that has the advantage of simplicity.

The first step in the algorithm is the initialization step. The variables `means`, `classication table`, and `covariances` are initialized to zero, and other program parameters are set. The next step begins by adding all observations in `x` to the means and the factorizations of the covariance matrices. It continues by computing some statistics of interest if requested: the linear discriminant functions, the prior probabilities, the log of the determinant of each of the covariance matrices, a test statistic for testing that all of the within-group covariance matrices are equal, and a matrix of Mahalanobis distances between the groups. The matrix of Mahalanobis distances is computed via the pooled covariance matrix when linear discrimination is specified, the row covariance matrix is used when the discrimination is quadratic. Covariance matrices are defined as follows. Let $N_i$ denote the sum of the frequencies of the observations in group $i$, and let $M_i$ denote the number of observations in group $i$. Then, if $S_i$ denotes the within-group $i$ covariance matrix,

$$S_i = \frac{1}{N_i - 1} \sum_{j=1}^{M_i} w_j f_j (x_j - \overline{x})(x_j - \overline{x})^T$$

where $w_j$ is the weight of the $j$-th observation in group $i$, $f_j$ is its frequency, $x_j$ is the $j$-th observation column vector (in group $i$), and $\overline{x}$ denotes the mean vector of the observations in group i. The mean vectors are computed as

$$\overline{x} = \frac{1}{W_i} \sum_{j=1}^{M_i} w_j f_j x_j$$

where

$$W_i = \sum_{j=1}^{M_i} w_j f_j$$

Given the means and the covariance matrices, the linear discriminant function for group $i$ is computed as:

$$z_i = ln(p_i) - 0.5\overline{x_i}^T S_p^{-1}\overline{x_i} + x^T S_p^{-1}\overline{x_i}$$

where $ln(pi)$ is the natural log of the prior probability for the $i$-th group, $x$ is the observation to be classified, and $S_p$ denotes the pooled covariance matrix.

Let S denote either the pooled covariance matrix or one of the within-group covariance matrices $S_i$. ($S$ will be the pooled covariance matrix in linear discrimination, and $S_i$ otherwise.) The Mahalanobis distance between group $i$ and group $j$ is computed as:

$$D_{ij}^2 = (\overline{x_i} - \overline{x_j})^T S^{-1}(\overline{x_i} - \overline{x_j})$$

Finally, the asymptotic chi-squared test for the equality of covariance matrices is computed as follows (Morrison 1976, page 252):

$$\gamma = C^{-1}\sum_{i=1}^{k} n_i\{ln(|S_p|) - ln(|S_i|)\}$$

where $n_i$ is the number of degrees of freedom in the $i$-th sample covariance matrix, $k$ is the number of groups, and

$$C^{-1} = 1 - \frac{2p^2+3p-1}{6(p+1)(k-1)}\left(\sum_{i=1}^{k}\frac{1}{n_i} - \frac{1}{\Sigma_j n_j}\right)$$

where $p$ is the number of variables.

The estimated posterior probability of each observation $x$ belonging to group $i$ is computed using the prior probabilities and the sample mean vectors and estimated covariance matrices under a multivariate normal assumption. Under quadratic discrimination, the within-group covariance matrices are used to compute the estimated posterior probabilities. The estimated posterior probability of an observation $x$ belonging to group $i$ is

$$\hat{q}_i(x) = \frac{e^{-\frac{1}{2}D_i^2(x)}}{\sum_{j=1}^{k} e^{-\frac{1}{2}D_j^2(x)}}$$

where

$$D_i^2(x) = \begin{cases} (x-\overline{x_i})^T S_i^{-1}(x-\overline{x_i}) + ln\,|S_i| - 2ln(p_i) & LINEAR\ or\ QUADRATIC \\ (x-\overline{x_i})^T S_p^{-1}(x-\overline{x_i}) - 2ln(p_i) & LINEAR,\ POOLED \end{cases}$$

For the leaving-out-one method of classification, the sample mean vector and sample covariance matrices in the formula for

$$D_i^2(x)$$

are adjusted so as to remove the observation $x$ from their computation. For linear discrimination, the linear discriminant function coefficients are actually used to compute the same posterior probabilities.

Using the posterior probabilities, each observation in $X$ is classified into a group; the result is tabulated in the matrix returned by `getClassTable` and saved in the vector returned by `getClassMembership`. The clasification table is not altered at this stage if `X[i][groupIndex]` contains a group number that is out of range. If the reclassification method is specified, then all observations with no missing values in the `nVariables` classification variables are classified. When the leaving-out-one method is used, observations with invalid group numbers, weights, frequencies or classification variables are not classified. Regardless of the frequency, a 1 is added (or subtracted) from the classification table for each row of `X` that is classified and contains a valid group number. When the leaving-out-one method is used, adjustment is made to the posterior probabilities to remove the effect of the observation in the classification rule. In this adjustment, each observation is presumed to have a weight of `weights[i]`, and a frequency of 1.0. See Lachenbruch (1975, page 36) for the required adjustment.

Finally, upon completion, the covariance matrices are computed from their LU factorizations.

## Declaration

public class com.imsl.stat.DiscriminantAnalysis
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

## Inner Classes

### *class* DiscriminantAnalysis.SumOfWeightsNegException

The sum of the weights have become negative.

#### Declaration

public static class com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *DiscriminantAnalysis.SumOfWeightsNegException*
  `public` **DiscriminantAnalysis.SumOfWeightsNegException**`( java.lang.String` **message** `)`

- *DiscriminantAnalysis.SumOfWeightsNegException*
  `public` **DiscriminantAnalysis.SumOfWeightsNegException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* DiscriminantAnalysis.EmptyGroupException

There are no observations in a group. Cannot compute statistics.

#### Declaration

public static class com.imsl.stat.DiscriminantAnalysis.EmptyGroupException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *DiscriminantAnalysis.EmptyGroupException*
  `public` **DiscriminantAnalysis.EmptyGroupException**`( java.lang.String` **message** `)`

- *DiscriminantAnalysis.EmptyGroupException*
  `public` **DiscriminantAnalysis.EmptyGroupException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### *class* DiscriminantAnalysis.CovarianceSingularException

The variance-Covariance matrix is singular.

#### Declaration

public static class com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException
**extends** com.imsl.IMSLException (page 1240)

#### Constructors

- *DiscriminantAnalysis.CovarianceSingularException*
  `public` **DiscriminantAnalysis.CovarianceSingularException**`(`
  `java.lang.String` **message** `)`

- *DiscriminantAnalysis.CovarianceSingularException*
  `public` **DiscriminantAnalysis.CovarianceSingularException**`(`
  `java.lang.String` **key**, `java.lang.Object[]` **arguments** `)`

## Fields

- public static final `int` **LINEAR**
  - Indicates a linear discrimination method.
- public static final `int` **QUADRATIC**
  - Indicates a quadratic discrimination method.
- public static final `int` **POOLED**
  - Indicates Pooled covariances computed.
- public static final `int` **POOLED_GROUP**
  - Indicates Pooled, group covariances computed.
- public static final `int` **RECLASSIFICATION**
  - Indicates reclassification as the classicfication method.
- public static final `int` **LEAVE_OUT_ONE**
  - Indicates leave-out-one as the Classicfication Method.
- public static final `int` **PRIOR_PROPORTIONAL**
  - Indicates prior probability type is to be prior proportional.
- public static final `int` **PRIOR_EQUAL**
  - Indicates prior probability type is to be prior equal.

## Constructor

- *DiscriminantAnalysis*
  `public` **DiscriminantAnalysis**`( int` **nVariables**`, int` **nGroups** `)`
  - **Description**
    Constructor for `DiscriminantAnalysis`.
  - **Parameters**
    * `nVariables` – An `int` representing the number of variables to be used in the discrimination.
    * `nGroups` – An `int` representing the number of groups in the data.

## Methods

- *getClassMembership*
  `public int[]` **getClassMembership**`( )`
  - **Description**
    Returns the group number to which the observation was classified.
  - **Returns** – An `int` array containing the group to which the observation was classified. If an observation has an invalid group number, frequency, or weight when the leaving-out-one method has been specified, then the observation is not classified and the corresponding elements of the array are set to zero.

---

- *getClassTable*
  `public double[][]` **getClassTable**`( )`
  - **Description**
    Returns the classification table.
  - **Returns** – A *nGroups* × *nGroups* `double` array containing the classification table. Each observation that is classified and has a group number equal to 1.0, 2.0, ..., *nGroups* is entered into the table. The rows of the table correspond to the known group membership. The columns refer to the group to which the observation was classified.

---

- *getCoefficients*
  `public double[][]` **getCoefficients**`( )`
  - **Description**
    Returns the linear discriminant function coefficients.

– **Returns** – A `double` array containing the linear discriminant function coefficients. The first column of the array contains the constant term, and the remaining columns contain the variable coefficients. The $i$-th row of the returned array corresponds to group $i$. The coefficients are always computed as linear discriminant function coefficients even when quadratic discrimination is specified.

---

- *getCovariance*
`public double[][][]` **getCovariance**`( )`

  – **Description**
  Returns the array of covariances.

  – **Returns** – A $nVariables \times nVariables \times g$ `double` array containing the covariances. Here, $g = nGroups + 1$ unless pooled only covariance matrices are computed, in which case $g$=1. When pooled only covariance matrices are computed, the within-group covariance matrices are not computed. The pooled covariance matrix is always computed and is returned as the $g$-th covariance matrix.

---

- *getGroupCounts*
`public int[]` **getGroupCounts**`( )`

  – **Description**
  Returns the group counts.

  – **Returns** – An `int` array of length `nGroups` containing the number of observations in each group.

---

- *getMahalanobis*
`public double[][]` **getMahalanobis**`( )`

  – **Description**
  Returns the Mahalanobis distances between the group means.

  – **Returns** – A $nGroups \times nGroups$ `double` array containing the Mahalanobis distances between the group means. For linear discrimination, the Mahalanobis distance

$$D_{ij}^2$$

  between group means $i$ and $j$ is computed using the within covariance matrix for group $i$ in place of the pooled covariance matrix.

---

- *getMeans*
`public double[][]` **getMeans**`( )`

  – **Description**
  Returns the variable means.

– **Returns** – A `double` array containing the variable means. The $i$-th row of the returned array contains the group $i$ variable means.

---

- *getNRowsMissing*
  `public int` **getNRowsMissing**`( )`
  – **Description**
  Returns the number of rows of data encountered containing missing values (NaN).
  – **Returns** – A `int` representing the number of rows of data encountered containing missing values (NaN) for the classification, group, weight, and/or frequency variables. If a row of data contains a missing value (NaN) for any of these variables, that row is excluded from the computations.

---

- *getPrior*
  `public double[]` **getPrior**`( )`
  – **Description**
  Returns the prior probabilities.
  – **Returns** – A `double` vector of length `nGroups` containing the prior probabilities for each group.

---

- *getProbability*
  `public double[][]` **getProbability**`( )`
  – **Description**
  Returns the posterior probabilities for each observation.
  – **Returns** – A $x.length \times nGroups$ `double` array containing the posterior probabilities for each observation.

---

- *getStatistics*
  `public double[]` **getStatistics**`( )`
  – **Description**
  Returns statistics.
  – **Returns** – A `double` array (stat) containing output statistics.

| *I* | *STAT[I]* |
|---|---|
| 0 | Sum of the degrees of freedom for the within-covariance matrices. |
| 1 | Chi-squared statistic. |
| 2 | The degrees of freedom in the chi-squared statistic. |
| 3 | Probability of a greater chi-squared, respectively, of a test of the homogeneity of the within-covariance matrices. (Not computed when the pooled only covariance matrix is computed). |
| 4 thru 4+`nGroups` | Log of the determinant of each group's covariance matrix. (Not computed when the pooled only covariance matrix is computed) and of the pooled covariance matrix. |
| Last `nGroups + 1` elements | Sum of the weights within each group. |
| Last element | Sum of the weights in all groups. |

- *setClassificationMethod*
  `public void` **setClassificationMethod**`( int` **method** `)`
  - **Description**
    Sets the classification method.
  - **Parameters**
    * `method` – A `int` scalar indicating the method of classification. Use class member `RECLASSIFICATION` or `LEAVE_OUT_ONE`. If this member function is not called, the `RECLASSIFICATION` method is used.

- *setCovarianceComputation*
  `public void` **setCovarianceComputation**`( int` **type** `)`
  - **Description**
    Sets the type of covariance matrices to be computed.
  - **Parameters**
    * `type` – An `int` scalar indicating the type of covariance matrices to be computed. Use class member `POOLED` or `POOLED_GROUP`. If this member function is not called, the `POOLED_GROUP` type is used.

- *setDiscriminationMethod*
  `public void` **setDiscriminationMethod**`( int` **method** `)`

– **Description**
Sets the discrimination method.

– **Parameters**

  * `method` – An `int` scalar indicating the method of discrimination. Use class member `LINEAR` or `QUADRATIC`. If this member function is not called, the `LINEAR` method is used.

---

• *setPrior*

`public void` **setPrior**`( double[]` **prior** `)`

– **Description**
Sets the prior probabilities.

– **Parameters**

  * `prior` – A `double` vector of length `nGroups` containing the prior probabilities for each group. The elements of `prior` should sum to 1.0. If this member function is not called, the elements of `prior` are set so as to be equal if `PRIOR_EQUAL` is set or they are set to be proportional to the sample size in each group if `PRIOR_PROPORTIONAL` is set.

---

• *setPrior*

`public void` **setPrior**`( int` **type** `)`

– **Description**
Sets the type of prior probabilities to be computed.

– **Parameters**

  * `type` – An `int` scalar indicating the type of prior probabilities to be computed. Use class member `PRIOR_EQUAL` or `PRIOR_PROPORTIONAL`. If this member function is not called, the `PRIOR_EQUAL` type is used.

---

• *update*

```
public void update( double[][] x ) throws
com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException,
com.imsl.stat.DiscriminantAnalysis.EmptyGroupException,
com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException
```

– **Description**
Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

– **Parameters**

  * `x` – a `double` matrix containing the observations. The first `nVariables` columns correspond to the variables, and the last column (column `nVariables`) contains the group numbers. The groups must be numbered 1,2, ..., `nGroups`.

* *update*

  `public void` **update**`( double[][] x, double[]` **frequencies**`, double[]` **weights** `) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException`

  – **Description**

    Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.

  – **Parameters**

    * `x` – A `double` matrix containing the observations. The first `nVariables` columns correspond to the variables, and the last column (column `nVariables`) contains the group numbers. The groups must be numbered 1,2, ..., `nGroups`.
    * `frequencies` – A `double` array containing the associated frequencies.
    * `weights` – A `double` array containing the associated weights.

* *update*

  `public void` **update**`( double[][] x, int` **groupIndex** `) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException`

  – **Description**

    Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

  – **Parameters**

    * `x` – A `double` matrix containing the observations. The first `nVariables` columns correspond to the variables, excluding the `groupIndex` column.
    * `groupIndex` – An `int` containing the column index of `x` in which the group numbers are stored. The groups must be numbered 1,2, ..., `nGroups`.

* *update*

  `public void` **update**`( double[][] x, int[]` **varIndex** `) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException`

  – **Description**

    Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

– **Parameters**

  * `x` – A `double` matrix containing the observations. The columns indicated in `varIndex` correspond to the variables, and the last column (column `nVariables`) contains the group numbers. The groups must be numbered 1,2, ..., `nGroups`.
  * `varIndex` – An `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis.

---

• *update*

`public void` **update**`( double[][] x, int[]` **varIndex**`, double[]` **frequencies**`, double[]` **weights** `) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException`

– **Description**

Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.

– **Parameters**

  * `x` – A `double` matrix containing the observations. The columns indicated in `varIndex` correspond to the variables, and the last column (column `nVariables`) contains the group numbers. The groups must be numbered 1,2, ..., `nGroups`.
  * `varIndex` – An `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis.
  * `frequencies` – A `double` array containing the associated frequencies.
  * `weights` – A `double` array containing the associated weights.

---

• *update*

`public void` **update**`( double[][] x, int` **groupIndex**`, double[]` **frequencies**`, double[]` **weights** `) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException`

– **Description**

Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.

– **Parameters**

  * `x` – A `double` matrix containing the observations. The first `nVariables` columns correspond to the variables, excluding the `groupIndex` column.

* groupIndex – An int containing the column index of x in which the group numbers are stored. The groups must be numbered 1,2, ..., nGroups.
* frequencies – A double array containing the associated frequencies.
* weights – A double array containing the associated weights.

---

- *update*

public void **update**( double[][] **x**, int **groupIndex**, int[] **varIndex** ) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException

  - **Description**

    Processes a set of observations and performs a linear or quadratic discriminant function analysis among the several known groups.

  - **Parameters**
    * x – A double matrix containing the observations. The columns indicated in varIndex correspond to the variables, and groupIndex column contains the group numbers.
    * groupIndex – An int containing the column index of x in which the group numbers are stored. The groups must be numbered 1,2, ..., nGroups.
    * varIndex – An int array containing the column indices in x that correspond to the variables to be used in the analysis.

---

- *update*

public void **update**( double[][] **x**, int **groupIndex**, int[] **varIndex**, double[] **frequencies**, double[] **weights** ) throws com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException, com.imsl.stat.DiscriminantAnalysis.EmptyGroupException, com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException

  - **Description**

    Processes a set of observations and associated frequencies and weights then performs a linear or quadratic discriminant function analysis among the several known groups.

  - **Parameters**
    * x – A double matrix containing the observations. The columns indicated in varIndex correspond to the variables, and groupIndex column contains the group numbers.
    * groupIndex – An int containing the column index of x in which the group numbers are stored. The groups must be numbered 1,2, ..., nGroups.
    * varIndex – An int array containing the column indices in x that correspond to the variables to be used in the analysis.
    * frequencies – A double array containing the associated frequencies.
    * weights – A double array containing the associated weights.

### Example: Discriminant Analysis

This example uses linear discrimination with equal prior probabilities on Fisher's (1936) iris data. This example illustrates the use of the DiscriminantAnalysis class.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;

public class DiscriminantAnalysisEx1 {
    public static void main(String args[]) throws Exception {
        double[][] xorig = {
            {1.0, 5.1, 3.5, 1.4, .2},
            {1.0, 4.9, 3.0, 1.4, .2},
            {1.0, 4.7, 3.2, 1.3, .2},
            {1.0, 4.6, 3.1, 1.5, .2},
            {1.0, 5.0, 3.6, 1.4, .2},
            {1.0, 5.4, 3.9, 1.7, .4},
            {1.0, 4.6, 3.4, 1.4, .3},
            {1.0, 5.0, 3.4, 1.5, .2},
            {1.0, 4.4, 2.9, 1.4, .2},
            {1.0, 4.9, 3.1, 1.5, .1},
            {1.0, 5.4, 3.7, 1.5, .2},
            {1.0, 4.8, 3.4, 1.6, .2},
            {1.0, 4.8, 3.0, 1.4, .1},
            {1.0, 4.3, 3.0, 1.1, .1},
            {1.0, 5.8, 4.0, 1.2, .2},
            {1.0, 5.7, 4.4, 1.5, .4},
            {1.0, 5.4, 3.9, 1.3, .4},
            {1.0, 5.1, 3.5, 1.4, .3},
            {1.0, 5.7, 3.8, 1.7, .3},
            {1.0, 5.1, 3.8, 1.5, .3},
            {1.0, 5.4, 3.4, 1.7, .2},
            {1.0, 5.1, 3.7, 1.5, .4},
            {1.0, 4.6, 3.6, 1.0, .2},
            {1.0, 5.1, 3.3, 1.7, .5},
            {1.0, 4.8, 3.4, 1.9, .2},
            {1.0, 5.0, 3.0, 1.6, .2},
            {1.0, 5.0, 3.4, 1.6, .4},
            {1.0, 5.2, 3.5, 1.5, .2},
            {1.0, 5.2, 3.4, 1.4, .2},
            {1.0, 4.7, 3.2, 1.6, .2},
```

```
{1.0, 4.8, 3.1, 1.6, .2},
{1.0, 5.4, 3.4, 1.5, .4},
{1.0, 5.2, 4.1, 1.5, .1},
{1.0, 5.5, 4.2, 1.4, .2},
{1.0, 4.9, 3.1, 1.5, .2},
{1.0, 5.0, 3.2, 1.2, .2},
{1.0, 5.5, 3.5, 1.3, .2},
{1.0, 4.9, 3.6, 1.4, .1},
{1.0, 4.4, 3.0, 1.3, .2},
{1.0, 5.1, 3.4, 1.5, .2},
{1.0, 5.0, 3.5, 1.3, .3},
{1.0, 4.5, 2.3, 1.3, .3},
{1.0, 4.4, 3.2, 1.3, .2},
{1.0, 5.0, 3.5, 1.6, .6},
{1.0, 5.1, 3.8, 1.9, .4},
{1.0, 4.8, 3.0, 1.4, .3},
{1.0, 5.1, 3.8, 1.6, .2},
{1.0, 4.6, 3.2, 1.4, .2},
{1.0, 5.3, 3.7, 1.5, .2},
{1.0, 5.0, 3.3, 1.4, .2},
{2.0, 7.0, 3.2, 4.7, 1.4},
{2.0, 6.4, 3.2, 4.5, 1.5},
{2.0, 6.9, 3.1, 4.9, 1.5},
{2.0, 5.5, 2.3, 4.0, 1.3},
{2.0, 6.5, 2.8, 4.6, 1.5},
{2.0, 5.7, 2.8, 4.5, 1.3},
{2.0, 6.3, 3.3, 4.7, 1.6},
{2.0, 4.9, 2.4, 3.3, 1.0},
{2.0, 6.6, 2.9, 4.6, 1.3},
{2.0, 5.2, 2.7, 3.9, 1.4},
{2.0, 5.0, 2.0, 3.5, 1.0},
{2.0, 5.9, 3.0, 4.2, 1.5},
{2.0, 6.0, 2.2, 4.0, 1.0},
{2.0, 6.1, 2.9, 4.7, 1.4},
{2.0, 5.6, 2.9, 3.6, 1.3},
{2.0, 6.7, 3.1, 4.4, 1.4},
{2.0, 5.6, 3.0, 4.5, 1.5},
{2.0, 5.8, 2.7, 4.1, 1.0},
{2.0, 6.2, 2.2, 4.5, 1.5},
{2.0, 5.6, 2.5, 3.9, 1.1},
{2.0, 5.9, 3.2, 4.8, 1.8},
{2.0, 6.1, 2.8, 4.0, 1.3},
```

```
{2.0, 6.3, 2.5, 4.9, 1.5},
{2.0, 6.1, 2.8, 4.7, 1.2},
{2.0, 6.4, 2.9, 4.3, 1.3},
{2.0, 6.6, 3.0, 4.4, 1.4},
{2.0, 6.8, 2.8, 4.8, 1.4},
{2.0, 6.7, 3.0, 5.0, 1.7},
{2.0, 6.0, 2.9, 4.5, 1.5},
{2.0, 5.7, 2.6, 3.5, 1.0},
{2.0, 5.5, 2.4, 3.8, 1.1},
{2.0, 5.5, 2.4, 3.7, 1.0},
{2.0, 5.8, 2.7, 3.9, 1.2},
{2.0, 6.0, 2.7, 5.1, 1.6},
{2.0, 5.4, 3.0, 4.5, 1.5},
{2.0, 6.0, 3.4, 4.5, 1.6},
{2.0, 6.7, 3.1, 4.7, 1.5},
{2.0, 6.3, 2.3, 4.4, 1.3},
{2.0, 5.6, 3.0, 4.1, 1.3},
{2.0, 5.5, 2.5, 4.0, 1.3},
{2.0, 5.5, 2.6, 4.4, 1.2},
{2.0, 6.1, 3.0, 4.6, 1.4},
{2.0, 5.8, 2.6, 4.0, 1.2},
{2.0, 5.0, 2.3, 3.3, 1.0},
{2.0, 5.6, 2.7, 4.2, 1.3},
{2.0, 5.7, 3.0, 4.2, 1.2},
{2.0, 5.7, 2.9, 4.2, 1.3},
{2.0, 6.2, 2.9, 4.3, 1.3},
{2.0, 5.1, 2.5, 3.0, 1.1},
{2.0, 5.7, 2.8, 4.1, 1.3},
{3.0, 6.3, 3.3, 6.0, 2.5},
{3.0, 5.8, 2.7, 5.1, 1.9},
{3.0, 7.1, 3.0, 5.9, 2.1},
{3.0, 6.3, 2.9, 5.6, 1.8},
{3.0, 6.5, 3.0, 5.8, 2.2},
{3.0, 7.6, 3.0, 6.6, 2.1},
{3.0, 4.9, 2.5, 4.5, 1.7},
{3.0, 7.3, 2.9, 6.3, 1.8},
{3.0, 6.7, 2.5, 5.8, 1.8},
{3.0, 7.2, 3.6, 6.1, 2.5},
{3.0, 6.5, 3.2, 5.1, 2.0},
{3.0, 6.4, 2.7, 5.3, 1.9},
{3.0, 6.8, 3.0, 5.5, 2.1},
{3.0, 5.7, 2.5, 5.0, 2.0},
```

```
{3.0, 5.8, 2.8, 5.1, 2.4},
{3.0, 6.4, 3.2, 5.3, 2.3},
{3.0, 6.5, 3.0, 5.5, 1.8},
{3.0, 7.7, 3.8, 6.7, 2.2},
{3.0, 7.7, 2.6, 6.9, 2.3},
{3.0, 6.0, 2.2, 5.0, 1.5},
{3.0, 6.9, 3.2, 5.7, 2.3},
{3.0, 5.6, 2.8, 4.9, 2.0},
{3.0, 7.7, 2.8, 6.7, 2.0},
{3.0, 6.3, 2.7, 4.9, 1.8},
{3.0, 6.7, 3.3, 5.7, 2.1},
{3.0, 7.2, 3.2, 6.0, 1.8},
{3.0, 6.2, 2.8, 4.8, 1.8},
{3.0, 6.1, 3.0, 4.9, 1.8},
{3.0, 6.4, 2.8, 5.6, 2.1},
{3.0, 7.2, 3.0, 5.8, 1.6},
{3.0, 7.4, 2.8, 6.1, 1.9},
{3.0, 7.9, 3.8, 6.4, 2.0},
{3.0, 6.4, 2.8, 5.6, 2.2},
{3.0, 6.3, 2.8, 5.1, 1.5},
{3.0, 6.1, 2.6, 5.6, 1.4},
{3.0, 7.7, 3.0, 6.1, 2.3},
{3.0, 6.3, 3.4, 5.6, 2.4},
{3.0, 6.4, 3.1, 5.5, 1.8},
{3.0, 6.0, 3.0, 4.8, 1.8},
{3.0, 6.9, 3.1, 5.4, 2.1},
{3.0, 6.7, 3.1, 5.6, 2.4},
{3.0, 6.9, 3.1, 5.1, 2.3},
{3.0, 5.8, 2.7, 5.1, 1.9},
{3.0, 6.8, 3.2, 5.9, 2.3},
{3.0, 6.7, 3.3, 5.7, 2.5},
{3.0, 6.7, 3.0, 5.2, 2.3},
{3.0, 6.3, 2.5, 5.0, 1.9},
{3.0, 6.5, 3.0, 5.2, 2.0},
{3.0, 6.2, 3.4, 5.4, 2.3},
{3.0, 5.9, 3.0, 5.1, 1.8}};
int i, j, jj, k;
int ipermu[] = {2, 3, 4, 5, 1};
double temp;
double x[][];

x = new double[xorig.length][xorig[0].length];
```

```
        for (i = 0; i< xorig.length; i++) {
            for (j = 1; j < xorig[0].length; j++) {
                x[i][j-1] = xorig[i][j];
            }
        }
        for (i = 0; i< xorig.length; i++) {
            x[i][4] = xorig[i][0];
        }

        int nvar = x[0].length -1;

        DiscriminantAnalysis da = new DiscriminantAnalysis(nvar, 3);
        da.setCovarianceComputation(da.POOLED);
        da.setClassificationMethod(da.RECLASSIFICATION);
        da.update(x);
        new PrintMatrix("Xmean are:  ").print(da.getMeans());
        new PrintMatrix("Coef:  ").print(da.getCoefficients());
        new PrintMatrix("Counts: ").print(da.getGroupCounts());
        new PrintMatrix("Stats: ").print(da.getStatistics());
        new PrintMatrix("ClassMembership: ").print(da.getClassMembership());
        new PrintMatrix("ClassTable: ").print(da.getClassTable());
        double cov[][][] = da.getCovariance();
        for (i= 0; i<cov.length;i++) {
             new PrintMatrix("Covariance Matrix "+i+" : ").print(cov[i]);
        }
        new PrintMatrix("Prior : ").print(da.getPrior());
        new PrintMatrix("PROB: ").print(da.getProbability());
        new PrintMatrix("MAHALANOBIS: ").print(da.getMahalanobis());
        System.out.println("nrmiss = " + da.getNRowsMissing());
    }

}
```

## Output

```
         Xmean are:
     0      1      2      3
0  5.006  3.428  1.462  0.246
1  5.936  2.77   4.26   1.326
2  6.588  2.974  5.552  2.026
```

```
                   Coef:
      0        1        2        3        4
0   -86.308  23.544  23.588  -16.431  -17.398
1   -72.853  15.698   7.073    5.211    6.434
2  -104.368  12.446   3.685   12.767   21.079

Counts:
   0
0  50
1  50
2  50

   Stats:
       0
 0  147
 1    ?
 2    ?
 3    ?
 4    ?
 5    ?
 6    ?
 7   -9.959
 8   50
 9   50
10   50
11  150

ClassMembership:
     0
  0  1
  1  1
  2  1
  3  1
  4  1
  5  1
  6  1
  7  1
  8  1
  9  1
 10  1
 11  1
```

```
12   1
13   1
14   1
15   1
16   1
17   1
18   1
19   1
20   1
21   1
22   1
23   1
24   1
25   1
26   1
27   1
28   1
29   1
30   1
31   1
32   1
33   1
34   1
35   1
36   1
37   1
38   1
39   1
40   1
41   1
42   1
43   1
44   1
45   1
46   1
47   1
48   1
49   1
50   2
51   2
52   2
53   2
```

```
54  2
55  2
56  2
57  2
58  2
59  2
60  2
61  2
62  2
63  2
64  2
65  2
66  2
67  2
68  2
69  2
70  3
71  2
72  2
73  2
74  2
75  2
76  2
77  2
78  2
79  2
80  2
81  2
82  2
83  3
84  2
85  2
86  2
87  2
88  2
89  2
90  2
91  2
92  2
93  2
94  2
95  2
```

| | |
|---|---|
| 96 | 2 |
| 97 | 2 |
| 98 | 2 |
| 99 | 2 |
| 100 | 3 |
| 101 | 3 |
| 102 | 3 |
| 103 | 3 |
| 104 | 3 |
| 105 | 3 |
| 106 | 3 |
| 107 | 3 |
| 108 | 3 |
| 109 | 3 |
| 110 | 3 |
| 111 | 3 |
| 112 | 3 |
| 113 | 3 |
| 114 | 3 |
| 115 | 3 |
| 116 | 3 |
| 117 | 3 |
| 118 | 3 |
| 119 | 3 |
| 120 | 3 |
| 121 | 3 |
| 122 | 3 |
| 123 | 3 |
| 124 | 3 |
| 125 | 3 |
| 126 | 3 |
| 127 | 3 |
| 128 | 3 |
| 129 | 3 |
| 130 | 3 |
| 131 | 3 |
| 132 | 3 |
| 133 | 2 |
| 134 | 3 |
| 135 | 3 |
| 136 | 3 |
| 137 | 3 |

```
138  3
139  3
140  3
141  3
142  3
143  3
144  3
145  3
146  3
147  3
148  3
149  3

 ClassTable:
   0   1   2
0  50   0   0
1   0  48   2
2   0   1  49

    Covariance Matrix 0 :
     0      1      2      3
0  0.265  0.093  0.168  0.038
1  0.093  0.115  0.055  0.033
2  0.168  0.055  0.185  0.043
3  0.038  0.033  0.043  0.042

 Prior :
     0
0  0.333
1  0.333
2  0.333

        PROB:
     0    1      2
  0  1  0      0
  1  1  0      0
  2  1  0      0
  3  1  0      0
  4  1  0      0
  5  1  0      0
  6  1  0      0
  7  1  0      0
```

| | | | |
|---|---|---|---|
| 8 | 1 | 0 | 0 |
| 9 | 1 | 0 | 0 |
| 10 | 1 | 0 | 0 |
| 11 | 1 | 0 | 0 |
| 12 | 1 | 0 | 0 |
| 13 | 1 | 0 | 0 |
| 14 | 1 | 0 | 0 |
| 15 | 1 | 0 | 0 |
| 16 | 1 | 0 | 0 |
| 17 | 1 | 0 | 0 |
| 18 | 1 | 0 | 0 |
| 19 | 1 | 0 | 0 |
| 20 | 1 | 0 | 0 |
| 21 | 1 | 0 | 0 |
| 22 | 1 | 0 | 0 |
| 23 | 1 | 0 | 0 |
| 24 | 1 | 0 | 0 |
| 25 | 1 | 0 | 0 |
| 26 | 1 | 0 | 0 |
| 27 | 1 | 0 | 0 |
| 28 | 1 | 0 | 0 |
| 29 | 1 | 0 | 0 |
| 30 | 1 | 0 | 0 |
| 31 | 1 | 0 | 0 |
| 32 | 1 | 0 | 0 |
| 33 | 1 | 0 | 0 |
| 34 | 1 | 0 | 0 |
| 35 | 1 | 0 | 0 |
| 36 | 1 | 0 | 0 |
| 37 | 1 | 0 | 0 |
| 38 | 1 | 0 | 0 |
| 39 | 1 | 0 | 0 |
| 40 | 1 | 0 | 0 |
| 41 | 1 | 0 | 0 |
| 42 | 1 | 0 | 0 |
| 43 | 1 | 0 | 0 |
| 44 | 1 | 0 | 0 |
| 45 | 1 | 0 | 0 |
| 46 | 1 | 0 | 0 |
| 47 | 1 | 0 | 0 |
| 48 | 1 | 0 | 0 |
| 49 | 1 | 0 | 0 |

| | | | |
|---|---|---|---|
| 50 | 0 | 1 | 0 |
| 51 | 0 | 0.999 | 0.001 |
| 52 | 0 | 0.996 | 0.004 |
| 53 | 0 | 1 | 0 |
| 54 | 0 | 0.996 | 0.004 |
| 55 | 0 | 0.999 | 0.001 |
| 56 | 0 | 0.986 | 0.014 |
| 57 | 0 | 1 | 0 |
| 58 | 0 | 1 | 0 |
| 59 | 0 | 1 | 0 |
| 60 | 0 | 1 | 0 |
| 61 | 0 | 0.999 | 0.001 |
| 62 | 0 | 1 | 0 |
| 63 | 0 | 0.994 | 0.006 |
| 64 | 0 | 1 | 0 |
| 65 | 0 | 1 | 0 |
| 66 | 0 | 0.981 | 0.019 |
| 67 | 0 | 1 | 0 |
| 68 | 0 | 0.96 | 0.04 |
| 69 | 0 | 1 | 0 |
| 70 | 0 | 0.253 | 0.747 |
| 71 | 0 | 1 | 0 |
| 72 | 0 | 0.816 | 0.184 |
| 73 | 0 | 1 | 0 |
| 74 | 0 | 1 | 0 |
| 75 | 0 | 1 | 0 |
| 76 | 0 | 0.998 | 0.002 |
| 77 | 0 | 0.689 | 0.311 |
| 78 | 0 | 0.993 | 0.007 |
| 79 | 0 | 1 | 0 |
| 80 | 0 | 1 | 0 |
| 81 | 0 | 1 | 0 |
| 82 | 0 | 1 | 0 |
| 83 | 0 | 0.143 | 0.857 |
| 84 | 0 | 0.964 | 0.036 |
| 85 | 0 | 0.994 | 0.006 |
| 86 | 0 | 0.998 | 0.002 |
| 87 | 0 | 0.999 | 0.001 |
| 88 | 0 | 1 | 0 |
| 89 | 0 | 1 | 0 |
| 90 | 0 | 0.999 | 0.001 |
| 91 | 0 | 0.998 | 0.002 |

| | | | |
|---|---|---|---|
| 92 | 0 | 1 | 0 |
| 93 | 0 | 1 | 0 |
| 94 | 0 | 1 | 0 |
| 95 | 0 | 1 | 0 |
| 96 | 0 | 1 | 0 |
| 97 | 0 | 1 | 0 |
| 98 | 0 | 1 | 0 |
| 99 | 0 | 1 | 0 |
| 100 | 0 | 0 | 1 |
| 101 | 0 | 0.001 | 0.999 |
| 102 | 0 | 0 | 1 |
| 103 | 0 | 0.001 | 0.999 |
| 104 | 0 | 0 | 1 |
| 105 | 0 | 0 | 1 |
| 106 | 0 | 0.049 | 0.951 |
| 107 | 0 | 0 | 1 |
| 108 | 0 | 0 | 1 |
| 109 | 0 | 0 | 1 |
| 110 | 0 | 0.013 | 0.987 |
| 111 | 0 | 0.002 | 0.998 |
| 112 | 0 | 0 | 1 |
| 113 | 0 | 0 | 1 |
| 114 | 0 | 0 | 1 |
| 115 | 0 | 0 | 1 |
| 116 | 0 | 0.006 | 0.994 |
| 117 | 0 | 0 | 1 |
| 118 | 0 | 0 | 1 |
| 119 | 0 | 0.221 | 0.779 |
| 120 | 0 | 0 | 1 |
| 121 | 0 | 0.001 | 0.999 |
| 122 | 0 | 0 | 1 |
| 123 | 0 | 0.097 | 0.903 |
| 124 | 0 | 0 | 1 |
| 125 | 0 | 0.003 | 0.997 |
| 126 | 0 | 0.188 | 0.812 |
| 127 | 0 | 0.134 | 0.866 |
| 128 | 0 | 0 | 1 |
| 129 | 0 | 0.104 | 0.896 |
| 130 | 0 | 0 | 1 |
| 131 | 0 | 0.001 | 0.999 |
| 132 | 0 | 0 | 1 |
| 133 | 0 | 0.729 | 0.271 |

```
134  0  0.066  0.934
135  0  0       1
136  0  0       1
137  0  0.006  0.994
138  0  0.193  0.807
139  0  0.001  0.999
140  0  0       1
141  0  0       1
142  0  0.001  0.999
143  0  0       1
144  0  0       1
145  0  0       1
146  0  0.006  0.994
147  0  0.003  0.997
148  0  0       1
149  0  0.018  0.982

        MAHALANOBIS:
      0       1        2
0    0      89.864  179.385
1   89.864   0       17.201
2  179.385  17.201    0

nrmiss = 0
```

# Chapter 20

# Probability Distribution Functions and Inverses

## Classes



## Usage Notes

Definitions and discussions of the terms basic to this chapter can be found in Johnson and Kotz (1969, 1970a, 1970b). These are also good references for the specific distributions.

In order to keep the calling sequences simple, whenever possible, the methods/classes described in this chapter are written for standard forms of statistical distributions. Hence, the number of parameters for any given distribution may be fewer than the number often associated with the distribution. Also, the methods relating to the normal distribution, `Cdf.normal` and `Cdf.inverseNormal`, are for a normal distribution with mean equal to zero and variance equal to one. For other means and variances, it is very easy for the user to standardize the variables by subtracting the mean and dividing by the square root of the variance.

The *distribution function* for the (real, single-valued) random variable $X$ is the function $F$

defined for all real $x$ by

$$F(x) = \text{Prob}(X \leq x)$$

where Prob($\cdot$) denotes the probability of an event. The distribution function is often called the *cumulative distribution function* (CDF).

For distributions with finite ranges, such as the beta distribution, the CDF is 0 for values less than the left endpoint and 1 for values greater than the right endpoint. The methods in the `Cdf` classes described in this chapter return the correct values for the distribution functions when values outside of the range of the random variable are input, but warning error conditions are set in these cases.

## Discrete Random Variables

For discrete distributions, the function giving the probability that the random variable takes on specific values is called the *probability function*, defined by

$$p(x) = \text{Prob}(X = x)$$

The CDF for a discrete random variable is

$$F(x) = \sum_{A} p(k)$$

where $A$ is set such that $k \leq x$. Since the distribution function is a step function, its inverse does not exist uniquely.

## Continuous Distributions

For continuous distributions, a probability function, as defined above, would not be useful because the probability of any given point is 0. For such distributions, the useful analog is the *probability density function* (PDF). The integral of the PDF is the probability over the interval, if the continuous random variable $X$ has PDF $f$, then

$$\text{Prob}(a \leq X \leq b) = \int_a^b f(x)\,dx$$

The relationship between the CDF and the PDF is

$$F(x) = \int_{-\infty}^{x} f(t)\,dt$$

For (absolutely) continuous distributions, the value of F(`x`) uniquely determines `x` within the support of the distribution. The “inverse” methods in the `Cdf` class compute the inverses of the distribution functions, that is, given F(`x`), they compute, `x`. The inverses are defined only over the open interval (0,1).

### Additional Comments

Whenever a probability close to 1.0 results from a call to a distribution function or is to be input to an inverse function, it is often impossible to achieve good accuracy because of the nature of the representation of numeric values. In this case, it may be better to work with the complementary distribution function (one minus the distribution function). If the distribution is symmetric about some point (as the normal distribution, for example) or is reflective about some point (as the beta distribution, for example), the complementary distribution function has a simple relationship with the distribution function. For example, to evaluate the standard normal distribution at 4.0, using the `normal` method in the `Cdf` class directly, the result to six places is 0.999968. Only two of those digits are really useful, however. A more useful result may be 1.000000 minus this value, which can be obtained to six places as 3.16712e-05 by evaluating `normal` at -4.0. For the normal distribution, the two values are related by $\Phi(x) = 1 - \Phi(-x)$, where $\Phi(\cdot)$ is the normal distribution function. Another example is the beta distribution with parameters 2 and 10. This distribution is skewed to the right, so evaluating `beta` at 0.7, 0.999953 is obtained. A more precise result is obtained by evaluating `beta` with parameters 10 and 2 at 0.3. This yields 4.72392e-5.

Many of the algorithms used by the classes in this chapter are discussed by Abramowitz and Stegun (1964). The algorithms make use of various expansions and recursive relationships and often use different methods in different regions.

Cumulative distribution functions are defined for all real arguments. However, if the input to one of the distribution functions in this chapter is outside the range of the random variable, an error is issued.

## *class* Cdf

Cumulative distribution functions.

### Declaration

public final class com.imsl.stat.Cdf
**extends** java.lang.Object

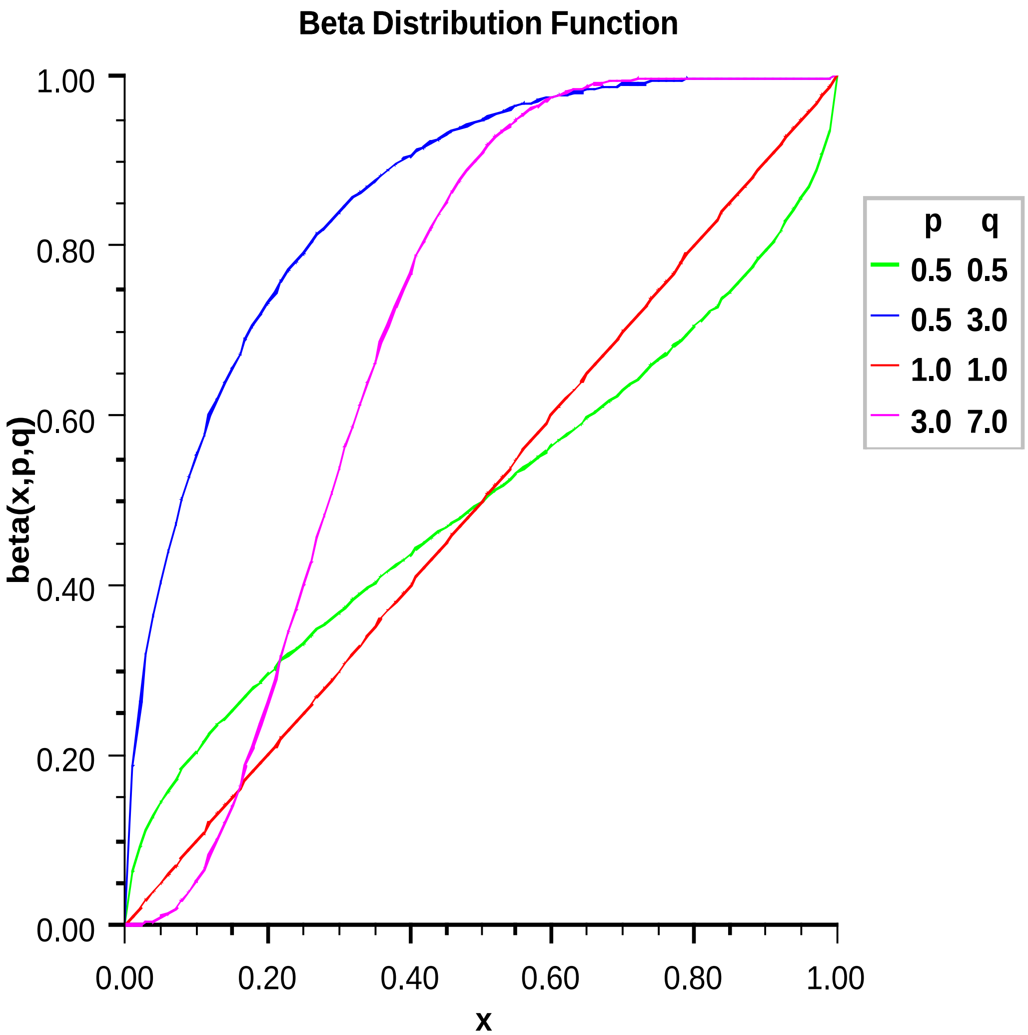
## Methods

- *beta*
  `public static double` **beta**`( double` **x**`, double` **pin**`, double` **qin** `)`

  – **Description**
  Evaluates the beta probability distribution function.
  Method `beta` evaluates the distribution function of a beta random variable with parameters `pin` and `qin`. This function is sometimes called the *incomplete beta ratio* and, with $p = pin$ and $q = qin$, is denoted by $I_x(p, q)$. It is given by

$$I_x\left(p,\, q\right) = \frac{\Gamma\left(p\right)\Gamma\left(q\right)}{\Gamma\left(p+q\right)}\int_0^x t^{p-1}\left(1-t\right)^{q-1}dt$$

  where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function $I_x(p, q)$ is the probability that the random variable takes a value less than or equal to $x$.
  The integral in the expression above is called the *incomplete beta function* and is denoted by $\beta_x(p, q)$. The constant in the expression is the reciprocal of the *beta function* (the incomplete function evaluated at one) and is denoted by $\beta_x(p, q)$.
  `beta` uses the method of Bosten and Battiste (1974).




– **Parameters**
  * `x` – a `double`, the argument at which the function is to be evaluated.
  * `pin` – a `double`, the first beta distribution parameter.
  * `qin` – a `double`, the second beta distribution parameter.
– **Returns** – a `double`, the probability that a beta random variable takes on a value less than or equal to x.

---

• *binomial*

`public static double` **`binomial`**`( int` **`k`**`, int` **`n`**`, double` **`p`** `)`

– **Description**
Evaluates the binomial distribution function.
Method `binomial` evaluates the distribution function of a binomial random variable with parameters $n$ and $p$. It does this by summing probabilities of the random variable taking on the specific values in its range. These probabilities are computed by the recursive relationship

$$\Pr\left(X=j\right)=\frac{\left(n+1-j\right)p}{j\left(1-p\right)}\Pr\left(X=j-1\right)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0, if $k$ is not greater than $n$ times $p$, and are computed backward from $n$, otherwise. The smallest positive machine number, $\varepsilon$, is used as the starting value for summing the probabilities, which are rescaled by $(1-p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is done. For the special case of $p=0$, `binomial` is set to 1; and for the case $p=1$, `binomial` is set to 1 if $k=n$ and to 0 otherwise.

– **Parameters**
  * `k` – the `int` argument for which the binomial distribution function is to be evaluated.
  * `n` – the `int` number of Bernoulli trials.
  * `p` – a `double` scalar value representing the probability of success on each trial.

– **Returns** – a `double` scalar value representing the probability that a binomial random variable takes a value less than or equal to k. This value is the probability that k or fewer successes occur in n independent Bernoulli trials, each of which has a p probability of success.

---

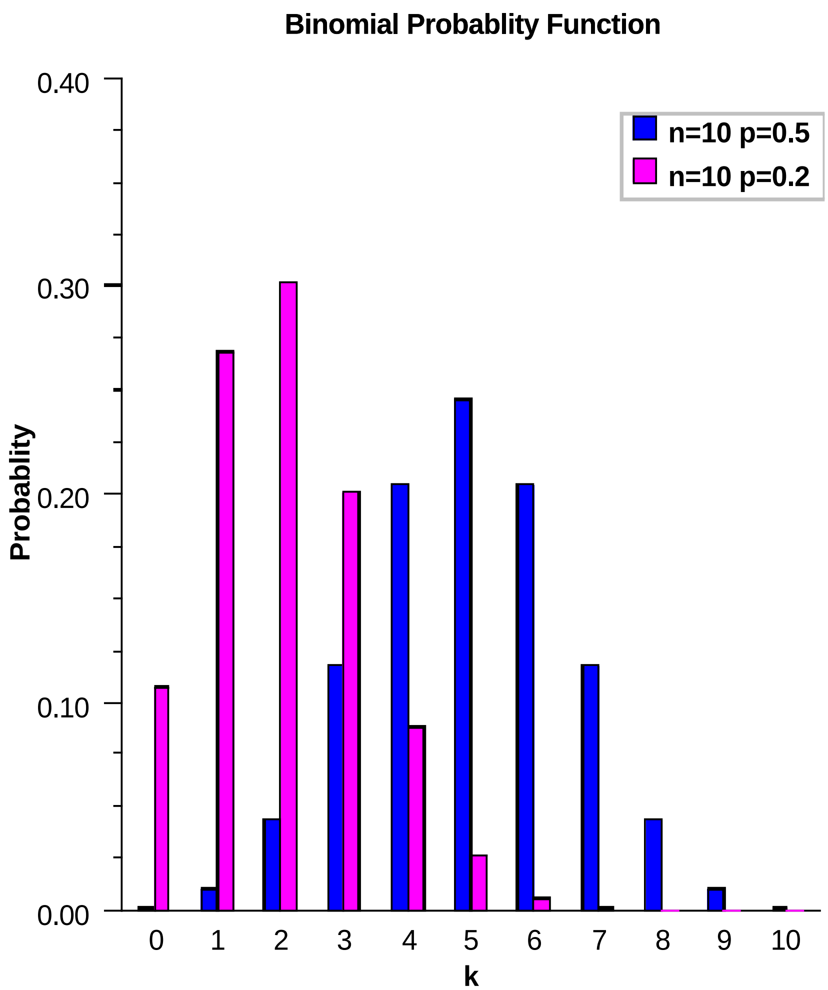
• *binomialProb*
`public static double` **binomialProb**`( int` **k**`, int` **n**`, double` **p** `)`

– **Description**
Evaluates the binomial probability function.
Method `binomialProb` evaluates the probability that a binomial random variable with parameters $n$ and $p$ takes on the value $k$. It does this by computing probabilities of the random variable taking on the values in its range less than (or the values greater than) $k$. These probabilities are computed by the recursive relationship

$$\Pr\left(X=j\right)=\frac{\left(n+1-j\right)p}{j\left(1-p\right)}\Pr\left(X=j-1\right)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0, if $k$ is not greater than $n\times p$, and are computed backward from $n$, otherwise. The smallest positive machine number, $\varepsilon$, is used as the starting

value for computing the probabilities, which are rescaled by $(1-p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is done.
For the special case of $p = 0$, `binomialProb` is set to 0 if $k$ is greater than 0 and to 1 otherwise; and for the case $p = 1$, `binomialProb` is set to 0 if $k$ is less than $n$ and to 1 otherwise.




– **Parameters**
  * `k` – the `int` argument for which the binomial distribution function is to be evaluated.
  * `n` – the `int` number of Bernoulli trials.

* p – a `double` scalar value representing the probability of success on each trial.

- **Returns** – a `double` scalar value representing the probability that a binomial random variable takes a value equal to k.

---

- *chi*

`public static double` **chi**`( double` **chsq**`, double` **df** `)`

  - **Description**

    Evaluates the chi-squared distribution function.

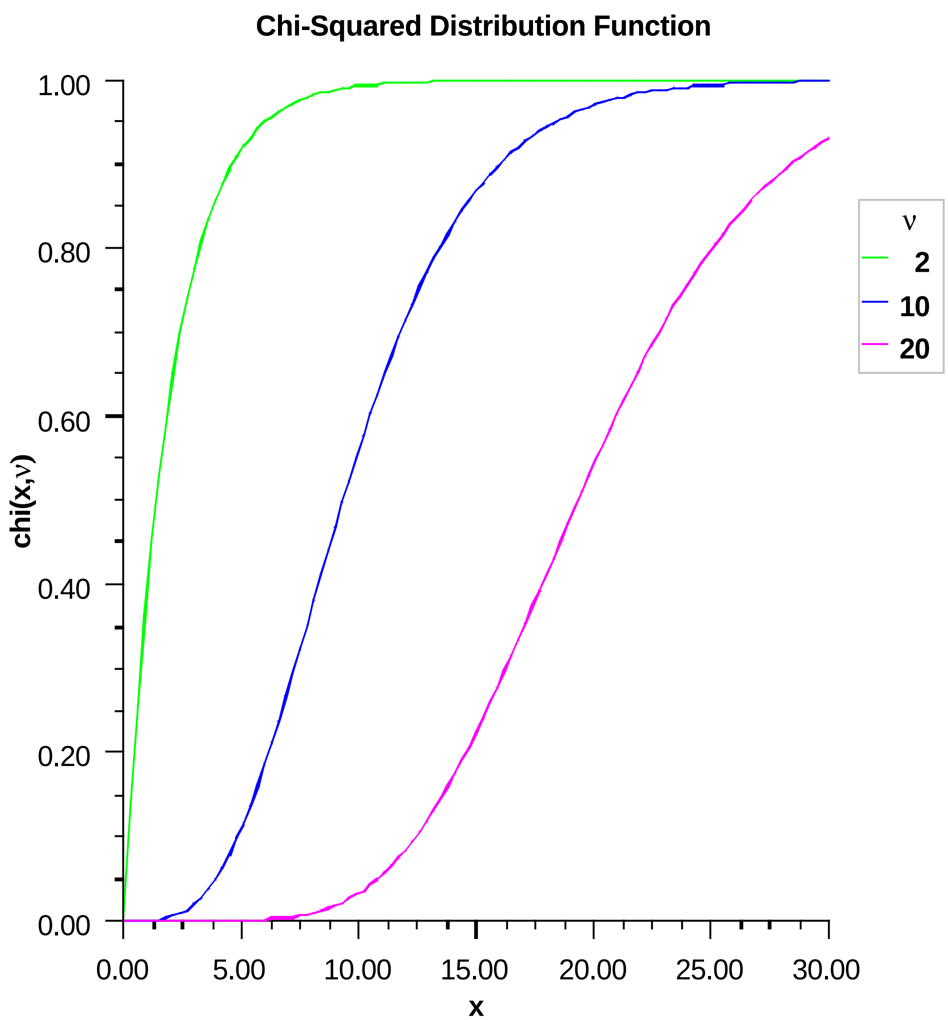
    Method `chi` evaluates the distribution function, $F$, of a chi-squared random variable with `df` degrees of freedom, that is, with $v = \mathrm{df}$, and $x = \mathrm{chsq}$,

$$F\left(x\right) = \frac{1}{2^{\nu/2}\Gamma\left(\nu/2\right)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

    where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point $x$ is the probability that the random variable takes a value less than or equal to $x$.

    For $v > 65$, `chi` uses the Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.17) to the normal distribution, and method `normal` is used to evaluate the normal distribution function.

    For $v \leq 65$, `chi` uses series expansions to evaluate the distribution function. If $x < \max(v/2, 26)$, `chi` uses the series 6.5.29 in Abramowitz and Stegun (1964), otherwise, it uses the asymptotic expansion 6.5.32 in Abramowitz and Stegun.




- **Parameters**
  - `chsq` – a `double` scalar value representing the argument at which the function is to be evaluated.
  - `df` – a `double` scalar value representing the number of degrees of freedom. This must be at least 0.5.
- **Returns** – a `double` scalar value representing the probability that a chi-squared random variable takes a value less than or equal to chsq.

---

- *F*

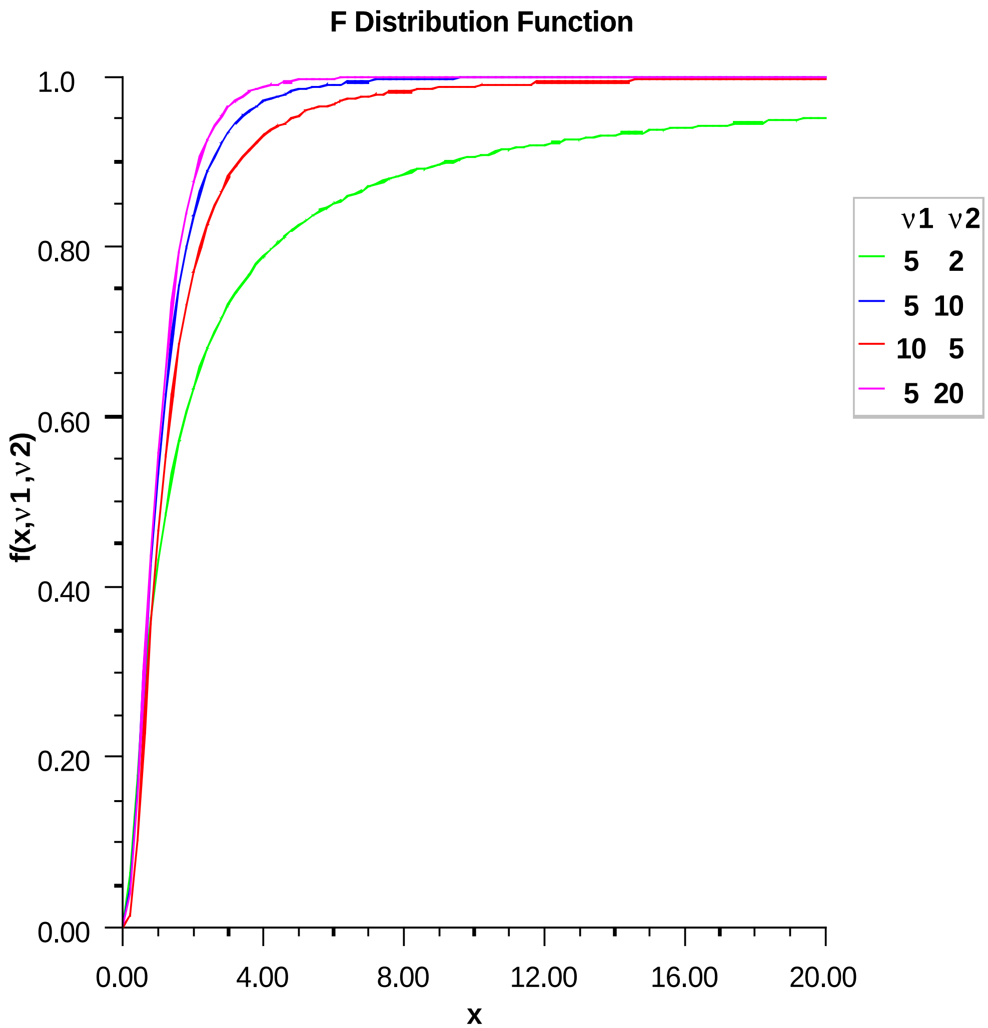
`public static double` **F**`( double` **x**`, double` **dfn**`, double` **dfd** `)`

- **Description**

  Evaluates the F distribution function.

  `F` evaluates the distribution function of a Snedecor's F random variable with `dfn` numerator degrees of freedom and `dfd` denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable and then using the function `beta`. If $X$ is an $F$ variate with $v_1$ and $v_2$ degrees of freedom and $Y = v_1X/(v_2 + v_1X)$, then $Y$ is a beta variate with parameters $p = v_1/2$ and $q = v_2/2$. `F` also uses a relationship between $F$ random variables that can be expressed as follows:

$$\mathrm{F}(X, dfn, dfd) = 1 - \mathrm{F}(1/X, dfd, dfn)$$




- **Parameters**
  - x – a double, the argument at which the function is to be evaluated.
  - dfn – a double, the numerator degrees of freedom. It must be positive.
  - dfd – a double, the denominator degrees of freedom. It must be positive.
- **Returns** – a double, the probability that an F random variable takes on a value less than or equal to x.

---

- *gamma*

```
public static double gamma( double x, double a )
```

– **Description**

Evaluates the gamma distribution function.

Method `gamma` evaluates the distribution function, $F$, of a gamma random variable with shape parameter $a$; that is,

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

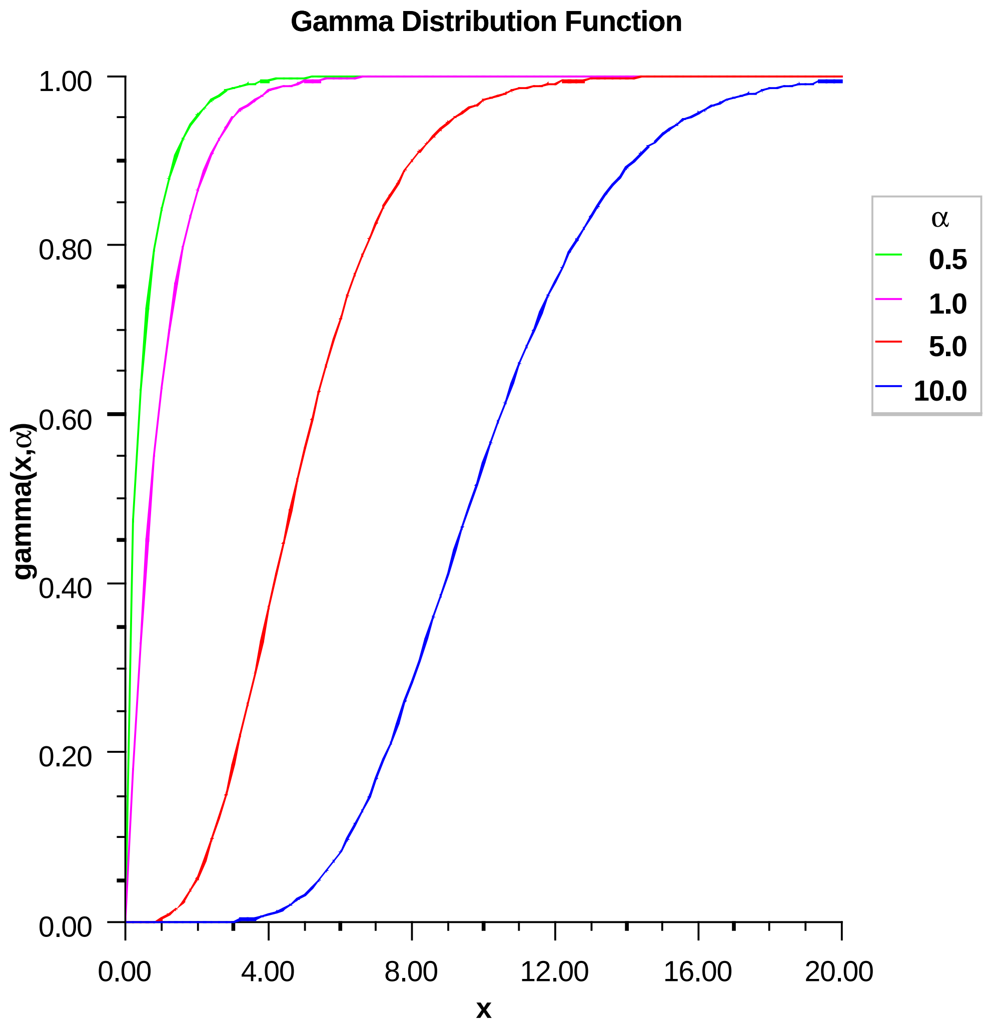
where $\Gamma(\cdot)$ is the gamma function. (The gamma function is the integral from 0 to $\infty$ of the same integrand as above). The value of the distribution function at the point $x$ is the probability that the random variable takes a value less than or equal to $x$.

The gamma distribution is often defined as a two-parameter distribution with a scale parameter $b$ (which must be positive), or even as a three-parameter distribution in which the third parameter $c$ is a location parameter. In the most general case, the probability density function over $(c, \infty)$ is

$$f(t) = \frac{1}{b^a \Gamma(a)} e^{-(t-c)/b} (x-c)^{a-1}$$

If $T$ is such a random variable with parameters $a$, $b$, and $c$, the probability that $T \leq t_0$ can be obtained from `gamma` by setting $X = (t_0 - c)/b$.

If $X$ is less than $a$ or if $X$ is less than or equal to 1.0, `gamma` uses a series expansion. Otherwise, a continued fraction expansion is used. (See Abramowitz and Stegun, 1964.)




- **Parameters**
  - `x` – a `double` scalar value representing the argument at which the function is to be evaluated.
  - `a` – a `double` scalar value representing the shape parameter. This must be positive.
- **Returns** – a `double` scalar value representing the probability that a gamma random variable takes on a value less than or equal to x.

---

- *hypergeometric*

public static double **hypergeometric**( int **k**, int **sampleSize**, int **defectivesInLot**, int **lotSize** )

- **Description**
  Evaluates the hypergeometric distribution function.
  Method `hypergeometric` evaluates the distribution function of a hypergeometric random variable with parameters $n$, $l$, and $m$. The hypergeometric random variable $X$ can be thought of as the number of items of a given type in a random sample of size $n$ that is drawn without replacement from a population of size $l$ containing $m$ items of this type. The probability function is

$$\Pr(X = j) = \frac{\binom{m}{j}\binom{l-m}{n-j}}{\binom{l}{n}} \text{for } j = i,\ i+1,\ i+2,\ \ldots,\ \min(n, m)$$

  where $i = \max(0, n - l + m)$.
  If $k$ is greater than or equal to $i$ and less than or equal to $\min(n, m)$, `hypergeometric` sums the terms in this expression for $j$ going from $i$ up to $k$. Otherwise, `hypergeometric` returns 0 or 1, as appropriate. So, as to avoid rounding in the accumulation, `hypergeometric` performs the summation differently depending on whether or not $k$ is greater than the mode of the distribution, which is the greatest integer less than or equal to $(m + 1)(n + 1)/(l + 2)$.
- **Parameters**
  * `k` – an `int`, the argument at which the function is to be evaluated.
  * `sampleSize` – an `int`, the sample size, `n`.
  * `defectivesInLot` – an `int`, the number of defectives in the lot, `m`.
  * `lotSize` – an `int`, the lot size, `l`.
- **Returns** – a `double`, the probability that a hypergeometric random variable takes a value less than or equal to k.

---

- *hypergeometricProb*
  public static double **hypergeometricProb**( int **k**, int **sampleSize**, int **defectivesInLot**, int **lotSize** )

  - **Description**
    Evaluates the hypergeometric probability function.
    Method `hypergeometricProb` evaluates the probability function of a hypergeometric random variable with parameters $n$, $l$, and $m$. The hypergeometric random variable $X$ can be thought of as the number of items of a given type in a random sample of size $n$ that is drawn without replacement from a population of size $l$ containing $m$ items of this type. The probability function is:

$$\Pr(X = k) = \frac{\binom{m}{k}\binom{l-m}{n-k}}{\binom{l}{n}} \text{for } k = i,\ i+1,\ i+2\ \ldots,\ \min(n, m)$$

where $i = max(0,\ n - l + m)$. `hypergeometricProb` evaluates the expression using log gamma functions.

- **Parameters**
  * `k` – an `int`, the argument at which the function is to be evaluated.
  * `sampleSize` – an `int`, the sample size, `n`.
  * `defectivesInLot` – an `int`, the number of defectives in the lot, `m`.
  * `lotSize` – an `int`, the lot size, `l`.
- **Returns** – a `double`, the probability that a hypergeometric random variable takes on a value equal to `k`.

---

- *inverseBeta*

`public static double` **inverseBeta**`( double` **p**`, double` **pin**`, double` **qin** `)`

- **Description**

  Evaluates the inverse of the beta probability distribution function.

  Method `inverseBeta` evaluates the inverse distribution function of a beta random variable with parameters `pin` and `qin`, that is, with $P = p$, $p = pin$, and $q = qin$, it determines $x$ (equal to `inverseBeta (p, pin, qin)`), such that

$$P = \frac{\Gamma(p)\,\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1}(1-t)^{q-1}\,dt$$

  where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to $x$ is $P$.
- **Parameters**
  * `p` – a `double`, the probability for which the inverse of the beta CDF is to be evaluated.
  * `pin` – a `double`, the first beta distribution parameter.
  * `qin` – a `double`, the second beta distribution parameter.
- **Returns** – a `double`, the probability that a beta random variable takes a value less than or equal to this value is p.

---

- *inverseChi*

`public static double` **inverseChi**`( double` **p**`, double` **df** `)`

- **Description**

  Evaluates the inverse of the chi-squared distribution function.

  Method `inverseChi` evaluates the inverse distribution function of a chi-squared random variable with `df` degrees of freedom, that is, with $P = p$ and $v = df$, it determines $x$ (equal to `inverseChi(p, df)`), such that

$$P = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^x e^{-t/2} t^{\nu/2-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to $x$ is $P$.

For $v < 40$, `inverseChi` uses bisection, if $v \geq 2$ or $P > 0.98$, or regula falsi to find the point at which the chi-squared distribution function is equal to $P$. The distribution function is evaluated using `chi`.

For $40 \leq v < 100$, a modified Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.18) to the normal distribution is used, and `inverseNormal` is used to evaluate the inverse of the normal distribution function. For $v \geq 100$, the ordinary Wilson-Hilferty approximation (Abramowitz and Stegun 1964, equation 26.4.17) is used.

– **Parameters**
  * `p` – a `double` scalar value representing the probability for which the inverse chi-squared function is to be evaluated.
  * `df` – a `double` scalar value representing the number of degrees of freedom. This must be at least 0.5.

– **Returns** – a `double` scalar value representing the probability that a chi-squared random variable takes a value less than or equal to this value is p.

---

- *inverseF*

`public static double` **inverseF**`( double` **p**`, double` **dfn**`, double` **dfd** `)`

– **Description**

Returns inverse of the F probability distribution function.

Method `inverseF` evaluates the inverse distribution function of a Snedecor's $F$ random variable with `dfn` numerator degrees of freedom and `dfd` denominator degrees of freedom. The function is evaluated by making a transformation to a beta random variable and then using `inverseBeta`. If $X$ is an $F$ variate with $v_1$ and $v_2$ degrees of freedom and $Y = v_1X/(v_2 + v_1X)$, then $Y$ is a beta variate with parameters $p = v_1/2$ and $q = v_2/2$. If $P \leq 0.5$, `inverseF` uses this relationship directly, otherwise, it also uses a relationship between $X$ random variables that can be expressed as follows, using `f`, which is the $F$ cumulative distribution function:

$$\mathrm{F}(X, dfn, dfd) = 1 - \mathrm{F}(1/X, dfd, dfn)$$

– **Parameters**
  * `p` – a `double`, the probability for which the inverse of the F distribution function is to be evaluated. Argument p must be in the open interval (0.0, 1.0).
  * `dfn` – a `double`, the numerator degrees of freedom. It must be positive.

* `dfd` – a `double`, the denominator degrees of freedom. It must be positive.

– **Returns** – a `double`, the probability that an F random variable takes a value less than or equal to this value is p.

---

- *inverseGamma*

`public static double` **inverseGamma**`( double p, double a )`

– **Description**

Evaluates the inverse of the gamma distribution function.

Method `inverseGamma` evaluates the inverse distribution function of a gamma random variable with shape parameter $a$, that is, it determines $x = \text{inverseGamma}(p, a))$, such that

$$P = \frac{1}{\Gamma(a)} \int_o^x e^{-t} t^{a-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to $x$ is $P$. See the documentation for routine `gamma` for further discussion of the gamma distribution.

`inverseGamma` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using method `gamma`.

– **Parameters**

* `p` – a `double` scalar value representing the probability at which the function is to be evaluated.
* `a` – a `double` scalar value representing the shape parameter. This must be positive.

– **Returns** – a `double` scalar value representing the probability that a gamma random variable takes a value less than or equal to this value is p.

---

- *inverseNormal*

`public static double` **inverseNormal**`( double p )`

– **Description**

Evaluates the inverse of the normal (Gaussian) distribution function.

Method `inverseNormal` evaluates the inverse of the distribution function, $\Phi$, of a standard normal (Gaussian) random variable, that is, `inverseNormal`(p) $= \Phi - 1(p)$, where

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt$$

The value of the distribution function at the point $x$ is the probability that the random variable takes a value less than or equal to $x$. The standard normal distribution has a mean of 0 and a variance of 1.

– **Parameters**

* p – a double scalar value representing the probability at which the function is to be evaluated.

– **Returns** – a double scalar value representing the probability that a standard normal random variable takes a value less than or equal to this value is p.

---

- *inverseStudentsT*
`public static double` **inverseStudentsT**`( double` **p**`, double` **df** `)`

– **Description**
Returns inverse of the Student's t distribution function.
`inverseStudentsT` evaluates the inverse distribution function of a Student's $t$ random variable with `df` degrees of freedom. Let $v = df$. If $v$ equals 1 or 2, the inverse can be obtained in closed form, if $v$ is between 1 and 2, the relationship of a $t$ to a beta random variable is exploited and `inverseBeta` is used to evaluate the inverse; otherwise the algorithm of Hill (1970) is used. For small values of $v$ greater than 2, Hill's algorithm inverts an integrated expansion in $1/(1+t^2/v)$ of the $t$ density. For larger values, an asymptotic inverse Cornish-Fisher type expansion about normal deviates is used.

– **Parameters**
* p – a double scalar value representing the probability for which the inverse Student's t function is to be evaluated.
* df – a double scalar value representing the number of degrees of freedom. This must be at least one.

– **Returns** – a double scalar value representing the probability that a Student's t random variable takes a value less than or equal to this value is p.

---

- *normal*
`public static double` **normal**`( double` **x** `)`

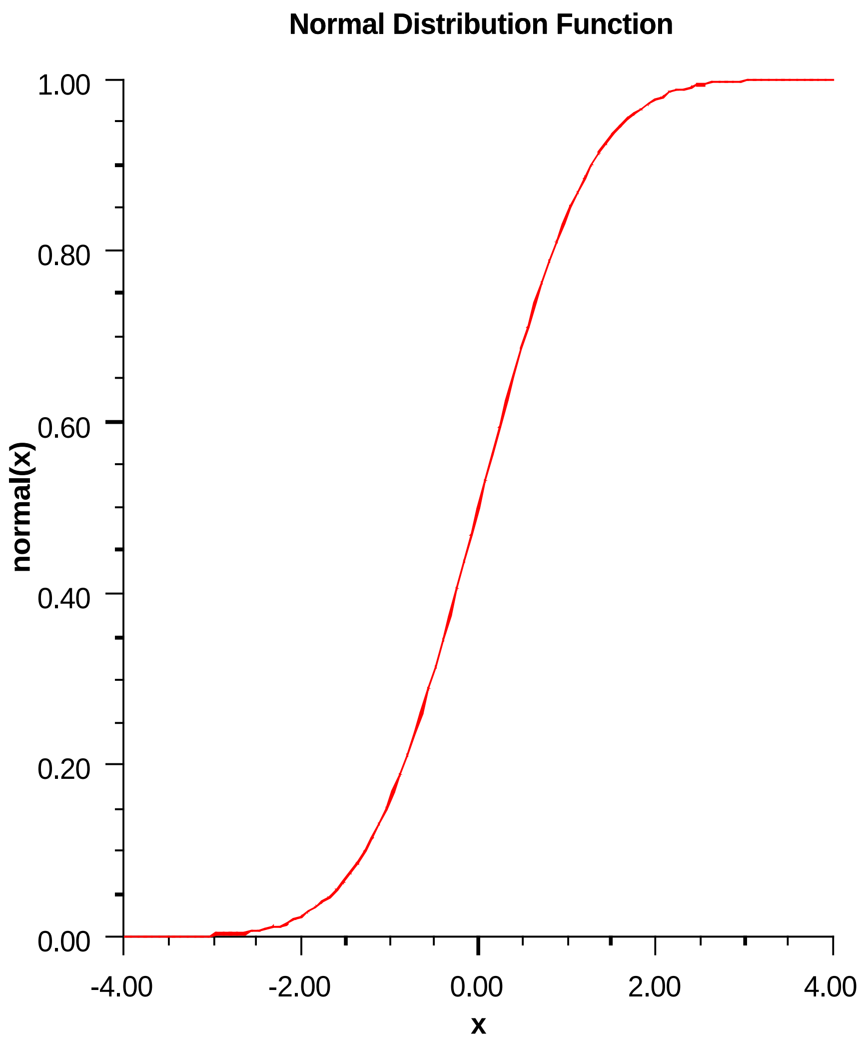
– **Description**
Evaluates the normal (Gaussian) distribution function.
Method `normal` evaluates the distribution function, $\Phi$, of a standard normal (Gaussian) random variable, that is,

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

The value of the distribution function at the point $x$ is the probability that the random variable takes a value less than or equal to $x$.
The standard normal distribution (for which `normal` is the distribution function) has mean of 0 and variance of 1. The probability that a normal random variable with mean $\mu$ and variance $\sigma^2$ is less than $y$ is given by `normal` evaluated at $(y - \mu)/\sigma$.

$\Phi(x)$ is evaluated by use of the complementary error function, erfc. The relationship is:

$$\Phi(x) = \text{erfc}(-x/\sqrt{2.0})/2$$




– **Parameters**

∗ `x` – a `double` scalar value representing the argument at which the function is to be evaluated.

– **Returns** – a `double` scalar value representing the probability that a normal variable takes a value less than or equal to x.

---

• *poisson*
`public static double` **poisson**`( int` **k**`, double` **theta** `)`

– **Description**
Evaluates the Poisson distribution function.
`poisson` evaluates the distribution function of a Poisson random variable with parameter `theta`. `theta`, which is the mean of the Poisson random variable, must be positive. The probability function (with $\theta = theta$) is

$$f(x) = e^{-\theta}\theta^x/x! \;\; for\, x = 0,\, 1,\, 2,\, \ldots$$

The individual terms are calculated from the tails of the distribution to the mode of the distribution and summed. `poisson` uses the recursive relationship

$$f\left(x+1\right) = f\left(x\right)\left(\theta/\left(x+1\right)\right), \;\; for\, x = 0,\, 1,\, 2,\, \ldots k-1$$

with $f(0) = e^{-\theta}$.

– **Parameters**
  * `k` – the `int` argument for which the Poisson distribution function is to be evaluated.
  * `theta` – a `double` scalar value representing the mean of the Poisson distribution.

– **Returns** – a `double` scalar value representing the probability that a Poisson random variable takes a value less than or equal to $k$.

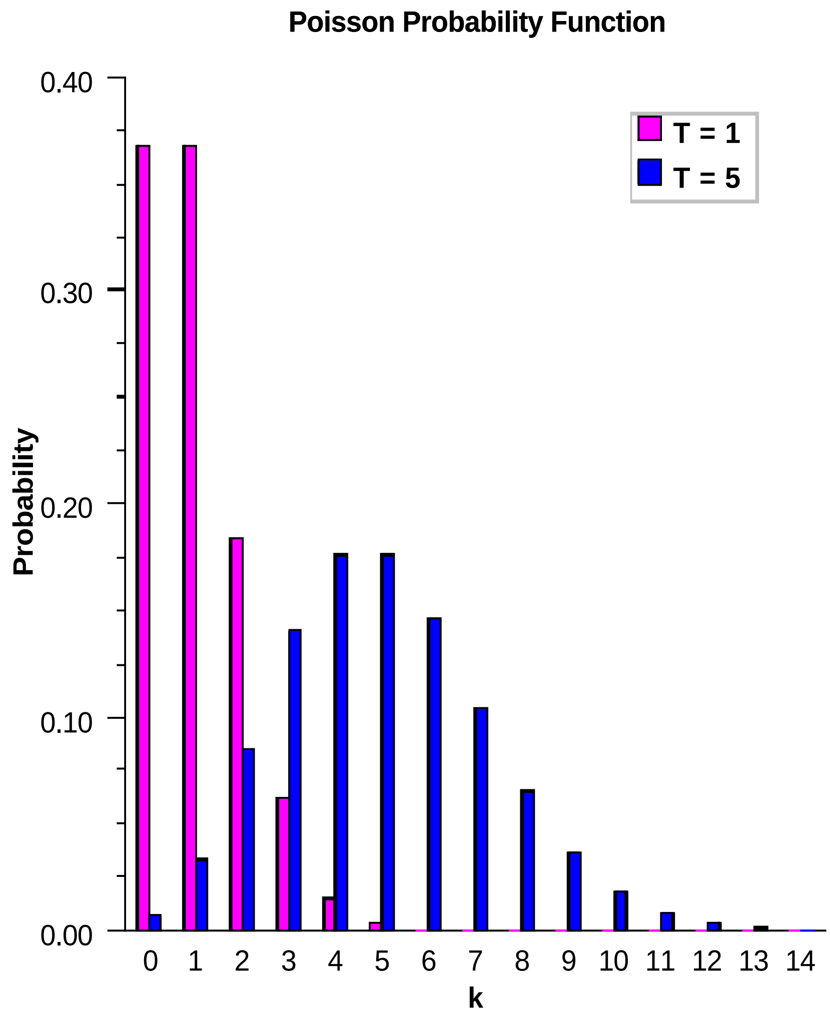
---

• *poissonProb*
`public static double` **poissonProb**`( int` **k**`, double` **theta** `)`

– **Description**
Evaluates the Poisson probability function.
Method `poissonProb` evaluates the probability function of a Poisson random variable with parameter `theta`. `theta`, which is the mean of the Poisson random variable, must be positive. The probability function (with $\theta = theta$) is

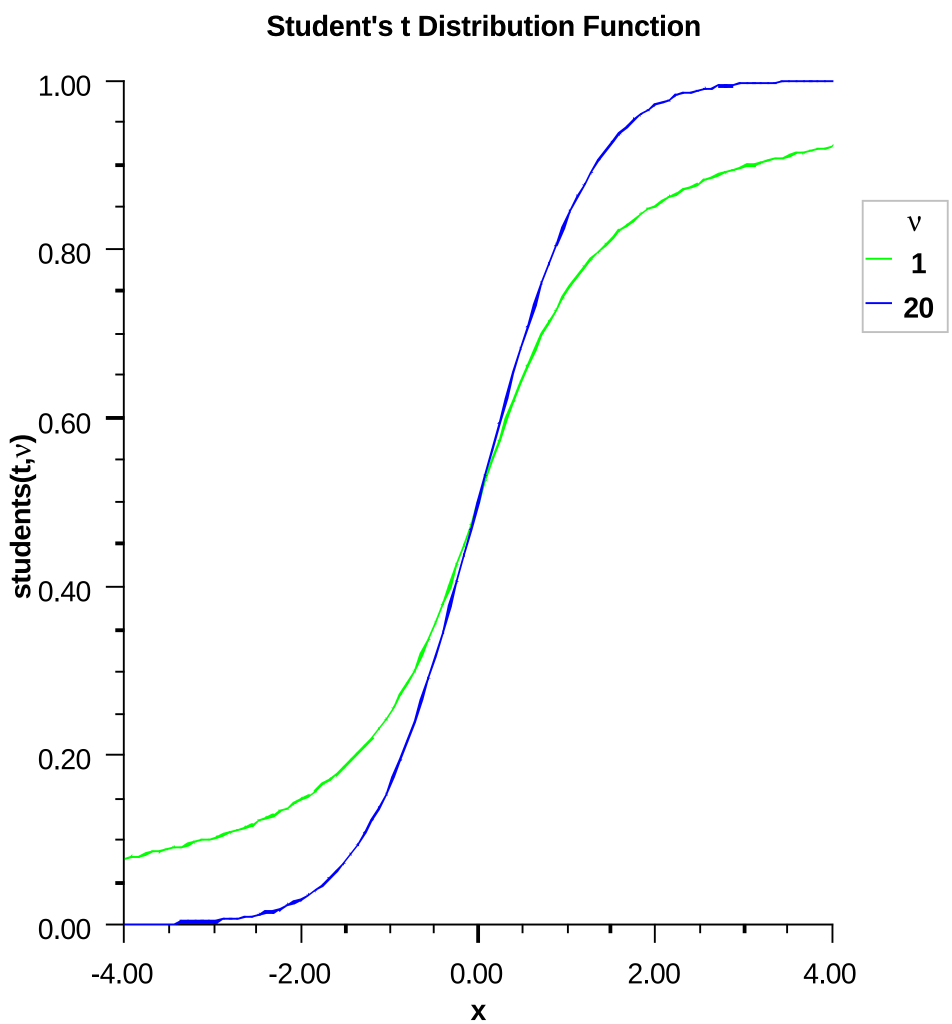
$$f(x) = e^{-\theta}\;\theta^k/k!, \quad for\, k = 0,\; 1,\; 2,\; \ldots$$

`poissonProb` evaluates this function directly, taking logarithms and using the log gamma function.




– **Parameters**
  * `k` – the `int` argument for which the Poisson probability function is to be evaluated.
  * `theta` – a `double` scalar value representing the mean of the Poisson distribution.
– **Returns** – a `double` scalar value representing the probability that a Poisson random variable takes a value equal to $k$.

- *studentsT*
  `public static double` **studentsT**`( double` **t**`, double` **df** `)`
  - **Description**
    Evaluates the Student's t distribution function.
    Method `studentsT` evaluates the distribution function of a Student's $t$ random variable with `df` degrees of freedom. If the square of $t$ is greater than or equal to `df`, the relationship of a $t$ to an $f$ random variable (and subsequently, to a beta random variable) is exploited, and routine `beta` is used. Otherwise, the method described by Hill (1970) is used. If `df` is not an integer, if `df` is greater than 19, or if `df` is greater than 200, a Cornish-Fisher expansion is used to evaluate the distribution function. If `df` is less than 20 and |t| is less than 2.0, a trigonometric series (see Abramowitz and Stegun 1964, equations 26.7.3 and 26.7.4, with some rearrangement) is used. For the remaining cases, a series given by Hill (1970) that converges well for large values of `t` is used.




– **Parameters**

  ∗ `t` – a `double` scalar value representing the argument at which the function is to be evaluated

  ∗ `df` – a `double` scalar value representing the number of degrees of freedom. This must be at least one.

– **Returns** – a `double` scalar value representing the probability that a Student's t random variable takes a value less than or equal to t

---

- *Weibull*

public static double **Weibull**( double **x**, double **gamma**, double **alpha** )

– **Description**
  Evaluates the Weibull distribution function.
– **Parameters**
  * `x` – a `double` scalar value representing the argument at which the function is to be evaluated. It must be non-negative.
  * `gamma` – a `double` scalar value representing the shape parameter.
  * `alpha` – a `double` scalar value representing the scale parameter.
– **Returns** – a `double` scalar value representing the probability that a Weibull random variable takes a value less than or equal to x

## Example: The Cumulative Distribution Functions

Various cumulative distribution functions are exercised. Their use in this example typifies the manner in which other functions in the Cdf class would be used.

```
import com.imsl.stat.*;

public class CdfEx1 {
    public static void main(String args[]) {
        double x, prob, result;
        int p, q, k, n;
        // Beta
        x =.5;
        p = 12;
        q = 12;
        result = Cdf.beta(x, p, q);
        System.out.println("beta(.5, 12, 12) is "+result);

        // Inverse Beta
        x =.5;
        p = 12;
        q = 12;
        result = Cdf.inverseBeta(x, p, q);
        System.out.println("inversebeta(.5, 12, 12) is "+result);

        // binomial
        k = 3;
        n = 5;
        prob = .95;
        result = Cdf.binomial(k, n, prob);
```

```
        System.out.println("binomial(3, 5, .95) is "+result);

        // Chi
        x = .15;
        n = 2;
        result = Cdf.chi(x, n);
        System.out.println("chi(.15, 2) is "+result);

        // Inverse Chi
        prob = .99;
        n = 2;
        result = Cdf.inverseChi(prob, n);
        System.out.println("inverseChi(.99, 2) is "+result);
    }
}
```

### Output

```
beta(.5, 12, 12) is 0.5000000000000016
inversebeta(.5, 12, 12) is 0.4999999999999991
binomial(3, 5, .95) is 0.02259250000000004
chi(.15, 2) is 0.07225651367144711
inverseChi(.99, 2) is 9.210340371976306
```

## *interface* **CdfFunction**

Public interface for the user-supplied cumulative distribution function to be used by InverseCdf and ChiSquaredTest.

### Declaration

public interface com.imsl.stat.CdfFunction

### Method

---

- *cdf*
  `double` **`cdf`**`( double` **`p`** `)`

– **Description**
  Public interface for the user-supplied cumulative distribution function to be used by InverseCdf.
– **Parameters**
  * `p` – a `double` scalar value representing the point at which the inverse CDF is desired.
– **Returns** – a `double` scalar value representing the probability that a random variable for this CDF takes a value less than or equal to this value is p.

## *class* **InverseCdf**

Inverse of user-supplied cumulative distribution function.

Class `InverseCdf` evaluates the inverse of a continuous, strictly monotone function. Its most obvious use is in evaluating inverses of continuous distribution functions that can be defined by a user-supplied function, which implements the `InverseCdf` interface. The inverse is computed using regula falsi and/or bisection, possibly with the Illinois modification (see Dahlquist and Bjorck 1974). A maximum of 100 iterations are performed.

### Declaration

public class com.imsl.stat.InverseCdf
**extends** java.lang.Object
**implements** java.io.Serializable

### Inner Class

#### *class* **InverseCdf.DidNotConvergeException**

The iteration did not converge

#### Declaration

public static class com.imsl.stat.InverseCdf.DidNotConvergeException
**extends** com.imsl.IMSLException (page 1240)

### Constructors

- *InverseCdf.DidNotConvergeException*
  `public` **InverseCdf.DidNotConvergeException**`( java.lang.String` **message** `)`

- *InverseCdf.DidNotConvergeException*
  `public` **InverseCdf.DidNotConvergeException**`( java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`

### Constructor

- *InverseCdf*
  `public` **InverseCdf**`( CdfFunction` **cdf** `)`
  - **Description**
    Constructor for the inverse of a user-supplied cummulative distribution function.
  - **Parameters**
    * `cdf` – is a `CdfFunction` object that contains the user-supplied function to be inverted. The cdf function must be continuous and strictly monotone.

### Methods

- *eval*
  `public double` **eval**`( double` **p**`, double` **guess** `) throws com.imsl.stat.InverseCdf.DidNotConvergeException`
  - **Description**
    Evaluates the inverse CDF function.
  - **Parameters**
    * `p` – a `double` scalar value representing the point at which the inverse CDF is desired
    * `guess` – a `double` scalar value representing an initial estimate of the inverse at p
  - **Returns** – a `double` scalar value representing the inverse of the CDF at the point p. Cdf(inverseCdf) is “close” to p.

- *setTolerance*
  `public void` **setTolerance**`( double` **tolerance** `)`

  - **Description**
    Sets the tolerance to be used as the convergence criterion.
  - **Parameters**
    * `tolerance` – a `double` scalar value representing the convergence criterion. When the relative change from one iteration to the next is less than tolerance, convergence is assumed. The default value for tolerance is 0.0001.

### Example: Inverse of a User-Supplied Cumulative Distribution Function

In this example, InverseCdf is used to compute the point such that the probability is 0.9 that a standard normal random variable is less than or equal to the computed point.

```
import com.imsl.stat.*;

public class InverseCdfEx1 implements CdfFunction {
    public double cdf(double x) {
        return Cdf.normal(x);
    }

    public static void main(String args[]) throws
    InverseCdf.DidNotConvergeException {
        double x1, p;

        p = 0.9;;
        InverseCdfEx1 invcdf = new InverseCdfEx1();
        InverseCdf inv = new InverseCdf(invcdf);
        inv.setTolerance(1.0e-10);
        x1 = inv.eval(p, 0.0);
        System.out.println("The 90th percentile of a standard normal is "+x1);
    }
}
```

### Output

```
The 90th percentile of a standard normal is 1.2815515655446006
```

# Chapter 21

# Random Number Generation

**Classes**



## *class* Random

Generate uniform and non-uniform random number distributions.

The non-uniform distributions are generated from a uniform distribution. By default, this class uses the uniform distribution generated by the base class java.util.Random. If the multiplier is set in this class then a multiplicative congruential method is used. The form of the generator is

$$x_i \equiv cx_{i-1} \mathrm{mod}(2^{31} - 1)$$

Each $x_i$ is then scaled into the unit interval (0,1). If the multiplier, $c$, is a primitive root modulo $2^{31} - 1$ (which is a prime), then the generator will have a maximal period of $2^{31} - 2$. There are several other considerations, however. See Knuth (1981) for a good general discussion. Possible values for $c$ are 16807, 397204094, and 950706376. The selection is made by the method setMultiplier. Evidence suggests that the performance of

950706376 is best among these three choices (Fishman and Moore 1982).

The generation of uniform (0,1) numbers is done by the method nextDouble.

## Declaration

```
public class com.imsl.stat.Random
extends java.util.Random
implements java.io.Serializable, java.lang.Cloneable
```

## Constructors

- *Random*
  `public Random( )`
  - **Description**
    Constructor for the Random number generator class.

- *Random*
  `public Random( long seed )`
  - **Description**
    Constructor for the Random number generator class with supplied seed.
  - **Parameters**
    * `seed` – a `long` which represents the random number generator seed

## Methods

- *next*
  `protected synchronized int next( int bits )`
  - **Description**
    Generates the next pseudorandom number. If the `multiplier` is set then the multiplicative congruential method is used. Otherwise, `super.next(bits)` is used.
  - **Parameters**
    * `bits` – is the number of random bits required.
  - **Returns** – the next pseudorandom value from this random number generator's sequence.

- *nextBeta*

  `public double` **nextBeta**`( double` **p**`, double` **q** `)`

  – **Description**

    Generate a pseudorandom number from a beta distribution.

    Method `nextBeta` generates pseudorandom numbers from a beta distribution with parameters $p$ and $q$, both of which must be positive. The probability density function is

    $$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1}(1-x)^{q-1} \quad for\, 0 \le x \le 1$$

    where $\Gamma(\cdot)$ is the gamma function.

    The algorithm used depends on the values of $p$ and $q$. Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all of the methods use acceptance/rejection. If $p$ and $q$ are both less than 1, the method of Johnk (1964) is used; if either $p$ or $q$ is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used; if both $p$ and $q$ are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used. The value returned is less than 1.0 and greater than $\varepsilon$, where $\varepsilon$ is the smallest positive number such that $1.0 - \varepsilon$ is less than 1.0.

  – **Parameters**

    * `p` – a `double`, the first beta distribution parameter, p >0
    * `q` – a `double`, the second beta distribution parameter, q >0

  – **Returns** – a `double`, a pseudorandom number from a beta distribution

- *nextBinomial*

  `public int` **nextBinomial**`( int` **n**`, double` **p** `)`

  – **Description**

    Generate a pseudorandom number from a binomial distribution.

    `nextBinomial` generates pseudorandom numbers from a binomial distribution with parameters $n$ and $p$. $n$ and $p$ must be positive, and $p$ must be less than 1. The probability function (with $n = n$ and $p = p$) is

    $$f(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

    for $x = 0, 1, 2, \ldots, n$.

    The algorithm used depends on the values of $n$ and $p$. If $np < 10$ or if $p$ is less than a machine epsilon, the inverse CDF technique is used; otherwise, the BTPE algorithm of Kachitvichyanukul and Schmeiser (see Kachitvichyanukul 1982) is used. This is an acceptance/rejection method using a composition of four regions. (TPE equals Triangle, Parallelogram, Exponential, left and right.)

- **Parameters**
  - `n` – an `int`, the number of Bernoulli trials.
  - `p` – a `double`, the probability of success on each trial, $0 < p < 1$.
- **Returns** – an `int`, the pseudorandom number from a binomial distribution.

---

- *nextCauchy*
  `public double` **nextCauchy**`( )`

  - **Description**
    Generates a pseudorandom number from a Cauchy distribution. The probability density function is

    $$f(x) = \frac{1}{\pi(1+x^2)}$$

    Use of the inverse CDF technique would yield a Cauchy deviate from a uniform (0, 1) deviate, $u$, as $\tan[\pi(u - .5)]$. Rather than evaluating a tangent directly, however, `nextCauchy` generates two uniform (-1, 1) deviates, $x_1$ and $x_2$. These values can be thought of as sine and cosine values. If

    $$x_1^2 + x_2^2$$

    is less than or equal to 1, then $x_1/x_2$ is delivered as the Cauchy deviate; otherwise, $x_1$ and $x_2$ are rejected and two new uniform (-1, 1) deviates are generated. This method is also equivalent to taking the ratio of two independent normal deviates.
    Deviates from the Cauchy distribution with median $t$ and first quartile $t$ - $s$, that is, with density

    $$f(x) = \frac{s}{\pi\left[s^2 + (x-t)^2\right]}$$

    can be obtained by scaling the output from `nextCauchy`. To do this, first scale the output from `nextCauchy` by $S$ and then add $T$ to the result.
  - **Returns** – a `double`, a pseudorandom number from a Cauchy distribution

---

- *nextChiSquared*
  `public double` **nextChiSquared**`( double` **df** `)`

  - **Description**
    Generates a pseudorandom number from a Chi-squared distribution. `nextChiSquared` generates pseudorandom numbers from a chi-squared distribution with `df` degrees of freedom. If `df` is an even integer less than 17, the chi-squared deviate $r$ is generated as

$$r = -2\ln\left(\prod_{i=1}^{n} u_i\right)$$

where $n = \mathrm{df}/2$ and the $u_i$ are independent random deviates from a uniform (0, 1) distribution. If `df` is an odd integer less than 17, the chi-squared deviate is generated in the same way, except the square of a normal deviate is added to the expression above. If `df` is greater than 16 or is not an integer, and if it is not too large to cause overflow in the gamma random number generator, the chi-squared deviate is generated as a special case of a gamma deviate, using `nextGamma`. If overflow would occur in `nextGamma`, the chi-squared deviate is generated in the manner described above, using the logarithm of the product of uniforms, but scaling the quantities to prevent underflow and overflow.

- **Parameters**
  - `df` – a `double` which specifies the number of degrees of freedom. It must be positive.
- **Returns** – a `double`, a pseudorandom number from a Chi-squared distribution.

---

- *nextExponential*

`public double` **nextExponential**`( )`

- **Description**

  Generates a pseudorandom number from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$; for $x > 0$.
  `nextExponential` uses an antithetic inverse CDF technique; that is, a uniform random deviate $U$ is generated and the inverse of the exponential cumulative distribution function is evaluated at *1.0 - U* to yield the exponential deviate. Deviates from the exponential distribution with mean $\theta$ can be generated by using `nextExponential` and then multiplying the result by $\theta$.
- **Returns** – a `double` which specifies a pseudorandom number from a standard exponential distribution

---

- *nextExponentialMix*

`public double` **nextExponentialMix**`( double` **theta1**`, double` **theta2**`, double` **p** `)`

- **Description**

  Generate a pseudorandom number from a mixture of two exponential distributions. The probability density function is

$$f\left(x\right) = \frac{p}{\theta}e^{-x/\theta_1} + \frac{1-p}{\theta_2}e^{-x/\theta_2} \;\; for\, x > 0$$

  where $p = \mathrm{p}$, $\theta_1 = theta1$, and $\theta_2 = theta2$.

In the case of a convex mixture, that is, the case $0 < p < 1$, the mixing parameter $p$ is interpretable as a probability; and `nextExponentialMix` with probability $p$ generates an exponential deviate with mean $\theta_1$, and with probability *1 - p* generates an exponential with mean $\theta_2$. When $p$ is greater than 1, but less than $\theta_1/(\theta_1 - \theta_2)$, then either an exponential deviate with mean $\theta_2$ or the sum of two exponentials with means $\theta_1$ and $\theta_2$ is generated. The probabilities are $q = p - (p-1)\theta_1/\theta_2$ and *1 - q*, respectively, for the single exponential and the sum of the two exponentials.

- **Parameters**
  - `theta1` – a `double` which specifies the mean of the exponential distribution that has the larger mean.
  - `theta2` – a `double` which specifies the mean of the exponential distribution that has the smaller mean. `theta2` must be positive and less than or equal to `theta1`.
  - `p` – a `double` which specifies the mixing parameter. It must satisfy $0 \le p \le \text{theta1}/(\text{theta1} - \text{theta2})$.
- **Returns** – a `double`, a pseudorandom number from a mixture of the two exponential distributions.

---

- *nextGamma*

`public double` **`nextGamma`**`( double a )`

- **Description**

  Generates a pseudorandom number from a standard gamma distribution. Method `nextGamma` generates pseudorandom numbers from a gamma distribution with shape parameter $a$. The probability density function is

$$P = \frac{1}{\Gamma(a)} \int_o^x e^{-t} t^{a-1} dt$$

  Various computational algorithms are used depending on the value of the shape parameter $a$. For the special case of $a = 0.5$, squared and halved normal deviates are used; and for the special case of $a = 1.0$, exponential deviates (from method `nextExponential`) are used. Otherwise, if $a$ is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used; if $a$ is greater than 1.0, a ten-region rejection procedure developed by Schmeiser and Lal (1980) is used.

  The Erlang distribution is a standard gamma distribution with the shape parameter having a value equal to a positive integer; hence, `nextGamma` generates pseudorandom deviates from an Erlang distribution with no modifications required.
- **Parameters**
  - `a` – a `double`, the shape parameter of the gamma distribution. It must be positive.

– **Returns** – a `double`, a pseudorandom number from a standard gamma distribution

---

- *nextGeometric*
`public int` **nextGeometric**`( double` **p** `)`

  – **Description**
  Generate a pseudorandom number from a geometric distribution.
  `nextGeometric` generates pseudorandom numbers from a geometric distribution with parameter $p$, where $P = p$ is the probability of getting a success on any trial. A geometric deviate can be interpreted as the number of trials until the first success (including the trial in which the first success is obtained). The probability function is

$$f(x) = P(1 - P)^{x-1}$$

  for $x = 1, 2, \ldots$ and $0 < P < 1$.
  The geometric distribution as defined above has mean *1/P*.
  The *i*-th geometric deviate is generated as the smallest integer not less than $log(U_i)/log(1 - P)$, where the $U_i$ are independent uniform (0, 1) random numbers (see Knuth, 1981).
  The geometric distribution is often defined on 0, 1, 2, ..., with mean *(1 - P)/P*. Such deviates can be obtained by subtracting 1 from each element returned value.

  – **Parameters**
    * `p` – a `double`, the probability of success on each trial, $0 < p \leq 1$.

  – **Returns** – an `int`, a pseudorandom number from a geometric distribution.

---

- *nextHypergeometric*
`public int` **nextHypergeometric**`( int` **n**`, int` **m**`, int` **l** `)`

  – **Description**
  Generate a pseudorandom number from a hypergeometric distribution.
  Method `nextHypergeometric` generates pseudorandom numbers from a hypergeometric distribution with parameters $n$, $m$, and $l$. The hypergeometric random variable $x$ can be thought of as the number of items of a given type in a random sample of size $n$ that is drawn without replacement from a population of size $l$ containing $m$ items of this type. The probability function is

$$f(x) = \frac{\binom{m}{x}\binom{l-m}{n-x}}{\binom{l}{n}}$$

  for $x = \max(0, n - l + m), 1, 2, \ldots, \min(n, m)$.

If the `hypergeometric` probability function with parameters $n$, $m$, and $l$ evaluated at $n$ - $l$ + $m$ (or at 0 if this is negative) is greater than the machine epsilon, and less than 1.0 minus the machine epsilon, then `nextHypergeometric` uses the inverse CDF technique. The method recursively computes the `hypergeometric` probabilities, starting at $x = \max(0, n - l + m)$ and using the ratio *f (x = x + 1)/f(x = x)* (see Fishman 1978, page 457).
If the `hypergeometric` probability function is too small or too close to 1.0, then `nextHypergeometric` generates integer deviates uniformly in the interval $[1, l - i]$, for $i = 0, 1, \ldots$; and at the *I*-th step, if the generated deviate is less than or equal to the number of special items remaining in the lot, the occurrence of one special item is tallied and the number of remaining special items is decreased by one. This process continues until the sample size or the number of special items in the lot is reached, whichever comes first. This method can be much slower than the inverse CDF technique. The timing depends on $n$. If $n$ is more than half of $l$ (which in practical examples is rarely the case), the user may wish to modify the problem, replacing $n$ by $l$ - $n$, and to consider the deviates to be the number of special items *not* included in the sample.

- **Parameters**
  - `n` – an `int` which specifies the number of items in the sample, n >0
  - `m` – an `int` which specifies the number of special items in the population, or lot, m >0
  - `l` – an `int` which specifies the number of items in the lot, l >max(n,m)
- **Returns** – an `int` which specifies the number of special items in a sample of size n drawn without replacement from a population of size l that contains m such special items.

---

- *nextLogarithmic*
`public int` **nextLogarithmic**`( double` **a2** `)`

- **Description**
Generate a pseudorandom number from a logarithmic distribution.
Method `nextLogarithmic` generates pseudorandom numbers from a logarithmic distribution with parameter $a$. The probability function is

$$f\left(x\right) = -\frac{a^x}{x \ln\left(1 - a\right)}$$

for $x = 1, 2, 3, \ldots$, and $0 < a < 1$.
The methods used are described by Kemp (1981) and depend on the value of $a$. If $a$ is less than 0.95, Kemp's algorithm LS, which is a "chop-down" variant of an inverse CDF technique, is used. Otherwise, Kemp's algorithm LK, which gives special treatment to the highly probable values of 1 and 2, is used.
- **Parameters**

* a2 – a `double` which specifies the parameter of the logarithmic distribution, $0 < a \leq 1.0$.

– **Returns** – an `int`, a pseudorandom number from a logarithmic distribution.

---

- *nextLogNormal*

`public double` **nextLogNormal**`( double` **mean**`, double` **stdev** `)`

– **Description**

Generate a pseudorandom number from a lognormal distribution.
Method `nextLogNormal` generates pseudorandom numbers from a lognormal distribution with parameters `mean` and `stdev`. The scale parameter in the underlying normal distribution, `stdev`, must be positive. The method is to generate normal deviates with mean `mean` and standard deviation `stdev` and then to exponentiate the normal deviates.
With $\mu = mean$ and $\sigma = stdev$, the probability density function for the lognormal distribution is

$$f(x) = \frac{1}{\sigma x\sqrt{2\pi}} \exp\left[-\frac{1}{2\sigma^2}(\ln x - \mu)^2\right] \; for \, x > 0$$

The mean and variance of the lognormal distribution are $\exp(\mu + \sigma 2/2)$ and $\exp(2\mu + 2\sigma 2) - \exp(2\mu + \sigma 2)$, respectively.

– **Parameters**

* `mean` – a `double` which specifies the mean of the underlying normal distribution
* `stdev` – a `double` which specifies the standard deviation of the underlying normal distribution. It must be positive.

– **Returns** – a `double`, a pseudorandom number from a lognormal distribution

---

- *nextMultivariateNormal*

`public double[]` **nextMultivariateNormal**`( int` **k**`, com.imsl.math.Cholesky` **matrix** `)`

– **Description**

Generate pseudorandom numbers from a multivariate normal distribution.
`nextMultivariateNormal` generates pseudorandom numbers from a multivariate normal distribution with mean vector consisting of all zeroes and variance-covariance matrix whose Cholesky factor (or "square root") is `matrix`; that is, `matrix` is an upper triangular matrix such that the transpose of `matrix` times `matrix` is the variance-covariance matrix. First, independent random normal deviates with mean 0 and variance 1 are generated, and then the matrix containing these deviates is post-multiplied by `matrix`.
Deviates from a multivariate normal distribution with means other than zero can be generated by using `nextMultivariateNormal` and then by adding the means to the deviates.

– **Parameters**

  * `k` – an `int` which specifies the length of the multivariate normal vectors
  * `matrix` – is the `Cholesky` factorization of the variance-covariance matrix of order k

– **Returns** – a `double` array which contains the pseudorandom numbers from a multivariate normal distribution

---

• *nextNegativeBinomial*
`public int` **nextNegativeBinomial**`( double` **rk**`, double` **p** `)`

– **Description**
Generate a pseudorandom number from a negative binomial distribution. Method `nextNegativeBinomial` generates pseudorandom numbers from a negative binomial distribution with parameters rk and p. rk and p must be positive and $p$ must be less than 1. The probability function with ($r =$ rk and $p =$ p) is

$$f(x) = \binom{r+x-1}{x} (1-p)^r p^x$$

for $x = 0, 1, 2, \ldots$.
If $r$ is an integer, the distribution is often called the Pascal distribution and can be thought of as modeling the length of a sequence of Bernoulli trials until $r$ successes are obtained, where $p$ is the probability of getting a success on any trial. In this form, the random variable takes values $r$, *r + 1*, $r + 2, \ldots$ and can be obtained from the negative binomial random variable defined above by adding $r$ to the negative binomial variable. This latter form is also equivalent to the sum of $r$ geometric random variables defined as taking values $1, 2, 3, \ldots$.
If *rp/(1 - p)* is less than 100 and $(1-p)^r$ is greater than the machine epsilon, `nextNegativeBinomial` uses the inverse CDF technique; otherwise, for each negative binomial deviate, `nextNegativeBinomial` generates a gamma *(r, p/(1 - p))* deviate $y$ and then generates a Poisson deviate with parameter $y$.

– **Parameters**

  * `rk` – a `double` which specifies the negative binomial parameter, rk >0
  * `p` – a `double` which specifies the probability of success on each trial. It must be greater than machine precision and less than one.

– **Returns** – an `int` which specifies the pseudorandom number from a negative binomial distribution. If rk is an integer, the deviate can be thought of as the number of failures in a sequence of Bernoulli trials before rk successes occur.

---

• *nextNormal*
`public double` **nextNormal**`( )`

– **Description**
Generate a pseudorandom number from a standard normal distribution using an inverse CDF method. In this method, a uniform (0,1) random deviate is generated, then the inverse of the normal distribution function is evaluated at that point using `inverseNormal`. This method is slower than the acceptance/rejection technique used in the `nextNormalAR` to generate standard normal deviates. Deviates from the normal distribution with mean $x_m$ and standard deviation $x_{std}$ can be obtained by scaling the output from `nextNormal`. To do this first scale the output of `nextNormal` by $x_{std}$ and then add $x_m$ to the result.

– **Returns** – a `double` which represents a pseudorandom number from a standard normal distribution

---

- *nextNormalAR*
`public double` **nextNormalAR**`( )`

– **Description**
Generate a pseudorandom number from a standard normal distribution using an acceptance/rejection method.
`nextNormalAR` generates pseudorandom numbers from a standard normal (Gaussian) distribution using an acceptance/rejection technique due to Kinderman and Ramage (1976). In this method, the normal density is represented as a mixture of densities over which a variety of acceptance/rejection methods due to Marsaglia (1964), Marsaglia and Bray (1964), and Marsaglia, MacLaren, and Bray (1964) are applied. This method is faster than the inverse CDF technique used in `nextNormal` to generate standard normal deviates.
Deviates from the normal distribution with mean $x_m$ and standard deviation $x_{std}$ can be obtained by scaling the output from `nextNormalAR`. To do this first scale the output of `nextNormalAR` by $x_{std}$ and then add $x_m$ to the result.

– **Returns** – a `double` which represents a pseudorandom number from a standard normal distribution

---

- *nextPoisson*
`public int` **nextPoisson**`( double` **theta** `)`

– **Description**
Generate a pseudorandom number from a Poisson distribution.
Method `nextPoisson` generates pseudorandom numbers from a Poisson distribution with parameter `theta`. `theta`, which is the mean of the Poisson random variable, must be positive. The probability function (with $\theta = \text{theta}$) is

$$f(x) = e^{-\theta}\,\theta^x/x!$$

for $x = 0, 1, 2, \ldots$
If theta is less than 15, nextPoisson uses an inverse CDF method; otherwise the PTPE method of Schmeiser and Kachitvichyanukul (1981) (see also Schmeiser 1983) is used.
The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

– **Parameters**
  * theta – a double which specifies the mean of the Poisson distribution, theta >0
– **Returns** – an int, a pseudorandom number from a Poisson distribution

---

- *nextStudentsT*
  public double **nextStudentsT**( double **df** )

  – **Description**
  Generate a pseudorandom number from a Student's t distribution.
  nextStudentsT generates pseudo-random numbers from a Student's $t$ distribution with df degrees of freedom, using a method suggested by Kinderman, Monahan, and Ramage (1977). The method ("TMX" in the reference) involves a representation of the $t$ density as the sum of a triangular density over (-2, 2) and the difference of this and the $t$ density. The mixing probabilities depend on the degrees of freedom of the $t$ distribution. If the triangular density is chosen, the variate is generated as the sum of two uniforms; otherwise, an acceptance/rejection method is used to generate a variate from the difference density.
  For degrees of freedom less than 100, nextStudentsT requires approximately twice the execution time as nextNormalAR, which generates pseudorandom normal deviates. The execution time of nextStudentsT increases very slowly as the degrees of freedom increase. Since for very large degrees of freedom the normal distribution and the $t$ distribution are very similar, the user may find that the difference in the normal and the $t$ does not warrant the additional generation time required to use nextStudentsT instead of nextNormalAR.
  – **Parameters**
    * df – a double which specifies the number of degrees of freedom. It must be positive.
  – **Returns** – a double, a pseudorandom number from a Student's t distribution

---

- *nextTriangular*
  public double **nextTriangular**( )

– **Description**
Generate a pseudorandom number from a triangular distribution on the interval (0,1). The probability density function is $f(x) = 4x$, for $0 \le x \le .5$, and $f(x) = 4(1 - x)$, for $.5 < x \le 1$. `nextTriangular` uses an inverse CDF technique.

– **Returns** – a `double`, a pseudorandom number from a triangular distribution on the interval (0,1)

---

- *nextVonMises*
`public double` **nextVonMises**`( double` **c** `)`

– **Description**
Generate a pseudorandom number from a von Mises distribution.
Method `nextVonMises` generates pseudorandom numbers from a von Mises distribution with parameter $c$, which must be positive. With $c = C$, the probability density function is

$$f\left(x\right) = \frac{1}{2\pi I_0\left(c\right)}\exp\left[c\,\cos\left(x\right)\right]\ for\ -\pi < x < \pi$$

where $I_0(c)$ is the modified Bessel function of the first kind of order 0. The probability density equals 0 outside the interval $(-\pi, \pi)$.
The algorithm is an acceptance/rejection method using a wrapped Cauchy distribution as the majorizing distribution. It is due to Best and Fisher (1979).

– **Parameters**
  * `c` – a `double` which specifies the parameter of the von Mises distribution, $c > 7.4 \times 10^{-9}$.

– **Returns** – a `double`, a pseudorandom number from a von Mises distribution

---

- *nextWeibull*
`public double` **nextWeibull**`( double` **a** `)`

– **Description**
Generate a pseudorandom number from a Weibull distribution.
Method `nextWeibull` generates pseudorandom numbers from a Weibull distribution with shape parameter $a$. The probability density function is

$$f\left(x\right) = Ax^{A-1}e^{-x^A}\ for\, x \ge 0$$

`nextWeibull` uses an antithetic inverse CDF technique to generate a Weibull variate; that is, a uniform random deviate $U$ is generated and the inverse of the Weibull cumulative distribution function is evaluated at *1.0 - u* to yield the Weibull deviate.
Deviates from the two-parameter Weibull distribution, with shape parameter $a$ and scale parameter $b$, can be generated by using `nextWeibull` and then multiplying the result by $b$.

The Rayleigh distribution with probability density function,

$$r(x) = \frac{1}{\alpha^2} x\, e^{\left(-x^2/2\alpha^2\right)} \; for\, x \geq 0$$

is the same as a Weibull distribution with shape parameter $a$ equal to 2 and scale parameter $b$ equal to.

$$\sqrt{2\alpha}$$

hence, `nextWeibull` and simple multiplication can be used to generate Rayleigh deviates.

- **Parameters**
  * `a` – a `double` which specifies the shape parameter of the Weibull distribution, a >0
- **Returns** – a `double`, a pseudorandom number from a Weibull distribution

---

- *setMultiplier*
  `public void` **setMultiplier**`( int` **multiplier** `)`
  - **Description**
    Sets the multiplier for a linear congruential random number generator. If a multiplier is set then the linear congruential generator, defined in the base class java.util.Random, is replaced by the generator
    seed = (multiplier*seed) mod ($2^{31} - 1$)
    See Donald Knuth, The Art of Computer Programming, Volume 2, for guidelines in choosing a multiplier. Some possible values are 16807, 397204094, 950706376.
  - **Parameters**
    * `multiplier` – an `int` which represents the random number generator multiplier

---

- *setSeed*
  `public void` **setSeed**`( long` **seed** `)`
  - **Description**
    Sets the seed.
  - **Parameters**
    * `seed` – a `long` which represents the random number generator seed

---

- *skip*
  `public void` **skip**`( int` **n** `)`

– **Description**
  Resets the seed to skip ahead in the base linear congruential generator. This method can be used only if a linear congruential multiplier is explicitly defined by a call to setMultiplier. The method skips ahead in the deviates returned by the protected method next. The public methods use `next(int)` as their source of uniform random deviates. Some methods call it more than once. For instance, each call to nextDouble calls it twice.
– **Parameters**
  * `n` – is the number of random deviates to skip.

## Example: Random Number Generation

In this example, a discrete normal random sample of size 1000 is generated via `Random.nextGaussian`. `Random.setSeed` is first used to set the seed. After the `ChiSquaredTest` constructor is called, the random observations are added to the test one at a time to simulate streaming data. The Chi-squared test is performed using `Cdf.normal` as the cumulative distribution function object to see how well the random numbers fit the normal distribution.

```
import com.imsl.stat.*;

public class RandomEx1 implements CdfFunction {
    public double cdf(double x) {
        return Cdf.normal(x);
    }

    public static void main(String args[]) throws
    InverseCdf.DidNotConvergeException {
        int nObservations = 1000;
        Random r = new Random(123457L);
        ChiSquaredTest test =
        new ChiSquaredTest(new RandomEx1(), 10, 0);
        for (int k = 0;  k < nObservations;  k++) {
            test.update(r.nextNormal(), 1.0);
        }

        double p = test.getP();
        System.out.println("The P-value is "+p);
    }
}
```

### Output

```
The P-value is 0.5518855965158243
```

## *class* **FaureSequence**

Generates the low-discrepancy Faure sequence.

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set $x_1, \ldots, x_n \in [0,1]^d$, $d \geq 1$, is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0,1]^d$ of the form

$$E = [0, t_1) \times \cdots \times [0, t_d)\,,\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d,$$

$\lambda$ is the Lebesque measure, and *A(E;n)* is the number of the $x_j$ contained in *E*.

The sequence $x_1, x_2, \ldots$ of points in $[0,1]^d$ is a low-discrepancy sequence if there exists a constant *c(d)*, depending only on *d*, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all $n > 1$.

Generalized Faure sequences can be defined for any prime base $b \geq d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \geq d$, so the base defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer $n$ in its $b$-ary expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers, $0 \leq a_j(n) < b$.

The $j$-th coordinate of $x_n$ is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} a_d(n) b^{-k-1},\ 1 \leq j \leq d$$

The generator matrix for the series, $c_{kd}^{(j)}$, is defined to be

$$c_{kd}^{(j)} = j^{d-k} c_{kd}$$

and $c_{kd}$ is an element of the Pascal matrix,

$$c_{kd} = \begin{cases} \frac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the $b$-ary Gray code. The function *G(n)* maps the positive integer $n$ into the integer given by its $b$-ary expansion. The sequence computed by this function is $\vec{x}(G(n))$, where $\vec{x}$ is the generalized Faure sequence.

## Declaration

public class com.imsl.stat.FaureSequence
**extends** java.lang.Object
**implements** java.io.Serializable, RandomSequence, java.lang.Cloneable

## Constructors

- *FaureSequence*
  `public` **FaureSequence**`( int` **dim** `)`
  - **Description**
    Creates a Faure sequence with the default base. The base defaults to the smallest prime equal to or greater than dim.
  - **Parameters**
    * `dim` – is the dimension of the sequence.

- *FaureSequence*
  `public` **FaureSequence**`( int` **dim**`, int` **base**`, int` **nSkip** `)`
  - **Description**
    Creates a Faure sequence.
  - **Parameters**
    * `dim` – is the dimension of the sequence.

* base – is the base of the sequence, as described above. It must be at least as large as dim.
* nSkip – is the number of initial points to skip. If negative then $base^{m/2-1}$, where $m$ is the number of digits needed to represent the Integer.MAX_VALUE in the base, points are skipped.

### Methods

---

- *clone*
  `public java.lang.Object` **clone**`( )`
  - **Description**
    Returns a copy of this object.

---

- *getBase*
  `public int` **getBase**`( )`
  - **Description**
    Returns the base.

---

- *getCount*
  `public long` **getCount**`( )`

---

- *getDimension*
  `public int` **getDimension**`( )`
  - **Description**
    Returns the dimension of the sequence.

---

- *getSkip*
  `public int` **getSkip**`( )`
  - **Description**
    Returns the number of points skipped at the beginning of the sequence.

---

- *nextDouble*
  `public double` **nextDouble**`( )`
  - **Description**
    Returns the first value of the next point in the sequence. This method is intended for use when dim is 1.
  - **Returns** – a `double` array, the next sequence value.

---

- *nextPoint*
  public double[] **nextPoint**( )
  - **Description**
    Returns the next point in the sequence.
  - **Returns** – a `double` array, the next point in the sequence.

---

- *nextPrime*
  public static int **nextPrime**( int **n** )
  - **Description**
    Returns the smallest prime greater than or equal to n.
  - **Parameters**
    - `n` – is the first number to try as a prime.
  - **Returns** – a prime greater than or equal to n. If n is less than or equal to 2 then 2 is returned.

### Example: FaureSequence

In this example, ten points of the Faure sequence are computed. The points are in a four-dimensional cube.

```
import com.imsl.stat.FaureSequence;
import com.imsl.math.PrintMatrix;

public class FaureSequenceEx1 {
    public static void main(String args[]) {
        FaureSequence seq = new FaureSequence(4);
        double x[][] = new double[10][];
        for (int k = 0;  k < 10;  k++) {
            x[k] = seq.nextPoint();
        }
        new PrintMatrix("Faure Sequence").print(x);
    }
}
```

### Output

```
        Faure Sequence
     0      1      2      3
0  0.201  0.275  0.533  0.694
1  0.401  0.475  0.733  0.894
```

```
2  0.601  0.675  0.933  0.094
3  0.801  0.875  0.133  0.294
4  0.841  0.115  0.573  0.934
5  0.041  0.315  0.773  0.134
6  0.241  0.515  0.973  0.334
7  0.441  0.715  0.173  0.534
8  0.641  0.915  0.373  0.734
9  0.681  0.155  0.613  0.374
```

## *interface* **RandomSequence**

Interface implemented by generators of random or quasi-random multidimension sequences.

### Declaration

public interface com.imsl.stat.RandomSequence

### Methods

---

- *getDimension*
  `int` **getDimension**`( )`
  - **Description**
    Returns the dimension of the sequence.

- *nextPoint*
  `double[]` **nextPoint**`( )`
  - **Description**
    Returns the next multidimensional point in the sequence.
  - **Returns** – a `double` array of length *dimension*.

# Chapter 22

# Input/Output

Classes



## *class* AbstractFlatFile

Reads a text or binary file as a ResultSet.

In Java, the result of a database query is normally returned as a `ResultSet` object. This class is intended to support reading of text or binary flat files and returning them as a `ResultSet`.

A *flat file* is a rectangular data set where each row is an observation and each column is a variable. The data type in any one column is the same for all of the rows.

### Declaration

public abstract class com.imsl.io.AbstractFlatFile
**extends** java.lang.Object
**implements** java.sql.ResultSet

### Inner Class

*class* **AbstractFlatFile.FlatFileSQLException**

A SQLException thrown by the AbstractFlatFile class.

#### Declaration

protected static class com.imsl.io.AbstractFlatFile.FlatFileSQLException
**extends** java.sql.SQLException

### Constructor

---

- *AbstractFlatFile*
  `public` **AbstractFlatFile**`( )`
  - **Description**
    Initializes an AbstractFlatFile. Since `AbstractFlatFile` is abstract, it cannot be directly instantiated.

### Methods

---

- *absolute*
  `public boolean` **absolute**`( int` **row** `) throws java.sql.SQLException`
  - **Description**
    Moves the cursor to the given row number in this `ResultSet` object.
  - **Parameters**
    * `row` – an `int` which specifies a row, of the `ResultSet` object, where the cursor is to be moved
  - **Returns** – a `boolean` whose value is `true` if the cursor is on the result set; `false` otherwise
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since only forward operations are allowed

---

- *afterLast*
  `public void` **afterLast**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor to the end of this `ResultSet` object, just after the last row. This method has no effect if the result set contains no rows.

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since this method has not been implemented

---

• *beforeFirst*
`public void` **beforeFirst**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor to the front of this `ResultSet` object, just before the first row. This method has no effect if the result set contains no rows.

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since only forward operations are allowed

---

• *beginGet*
`protected void` **beginGet**`( )`

– **Description**
This method should be called at the start of every `get`*Type* method. It closes any `InputStreams` or `Readers` created by get methods in this object. It also resets the `wasNull` flag to `false`.

---

• *cancelRowUpdates*
`public void` **cancelRowUpdates**`( ) throws java.sql.SQLException`

– **Description**
Cancels the updates made to the current row in this `ResultSet` object. Since updates are not allowed, this method always throws an `SQLException`.

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *clearWarnings*
`public void` **clearWarnings**`( ) throws java.sql.SQLException`

– **Description**
Clears all warnings reported on this `ResultSet` object. After this method is called, the method `getWarnings` returns `null` until a new warning is reported for this `ResultSet` object.

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *close*

  `public void` **close**`( ) throws java.sql.SQLException`

  - **Description**

    Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.
  - **Throws**
    - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *deleteRow*

  `public void` **deleteRow**`( ) throws java.sql.SQLException`

  - **Description**

    Deletes the current row from this `ResultSet` object and from the underlying database. Since updates are not allowed, this method always throws an `SQLException`.
  - **Throws**
    - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *doGetBytes*

  `protected abstract byte[]` **doGetBytes**`( int` **columnIndex** `) throws java.sql.SQLException`

  - **Description**

    Implements the actual `getBytes()`. The bytes represent the raw values returned by the driver.
  - **Parameters**
    - `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - **Returns** – a `byte` array representation of the column value; if the value is SQL `null`, the value returned is `null`
  - **Throws**
    - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *doNext*

  `protected abstract boolean` **doNext**`( ) throws java.sql.SQLException`

  - **Description**

    Implements the operations on the file required by the method next().

– **Returns** – a `boolean`, `true` if the new current row is valid; `false` if there are no more rows
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *findColumn*
`public int` **findColumn**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Maps the given `ResultSet` column name to its `ResultSet` column index.
– **Parameters**
  * `columnName` – a `String` specifying the name of the column
– **Returns** – an `int` specifying the column index of the given column name
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if the `ResultSet` object does not contain `columnName` or a database access error occurs

---

• *findColumnName*
`protected java.lang.String` **findColumnName**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
Maps the given columnIndex into its column name.
– **Parameters**
  * `columnIndex` – an `int` specifying the index of a column for which the name is to be found
– **Returns** – a `String` containing the name of the column
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *first*
`public boolean` **first**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor to the first row in this `ResultSet` object.
– **Returns** – a `boolean` whose value is `true` if the cursor is on the result set; `false` otherwise
– **Throws**

* com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since only forward operations are allowed

---

- *getArray*
  `public java.sql.Array` **getArray**`( int` **columnIndex** `) throws java.sql.SQLException`

  – **Description**
  Returns the value of the designated column in the current row of this `ResultSet` object as an `Array` object in the Java programming language.

  – **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

  – **Returns** – a `Array` object representing an SQL `Array` value in the specified column

  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since this method is not implemented

---

- *getArray*
  `public java.sql.Array` **getArray**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

  – **Description**
  Returns the value of the designated column in the current row of this `ResultSet` object as an `Array` object in the Java programming language.

  – **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column

  – **Returns** – an `Array` object representing the SQL `ARRAY` value in the specified column

  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getAsciiStream*
  `public java.io.InputStream` **getAsciiStream**`( int` **columnIndex** `) throws java.sql.SQLException`

  – **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a stream of ASCII characters. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large

LONGVARCHAR values. The JDBC driver will do any necessary conversion from the database format into ASCII.
**Note:** All the data in the returned stream must be read prior to getting the value of any other column. The next call to a `get`*Type* method implicitly closes the stream. Also, a stream may return `0` when the method `InputStream.available` is called whether there is data available or not.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
– **Returns** – a `java.io.InputStream` that delivers the database column value as a stream of one-byte ASCII characters; if the value is SQL `NULL`, the value returned is `null`
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getAsciiStream*
`public java.io.InputStream` **getAsciiStream**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a stream of ASCII characters. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC driver will do any necessary conversion from the database format into ASCII.
**Note:** All the data in the returned stream must be read prior to getting the value of any other column. The next call to a `get`*Type* method implicitly closes the stream. Also, a stream may return `0` when the method `available` is called whether there is data available or not.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
– **Returns** – a `java.io.InputStream` that delivers the database column value as a stream of one-byte ASCII characters. If the value is SQL `NULL`, the value returned is `null`.
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getBigDecimal*
`public java.math.BigDecimal` **getBigDecimal**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `java.math.BigDecimal` with full precision.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – a `java.math.BigDecimal` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

- *getBigDecimal*
`public java.math.BigDecimal` **getBigDecimal**`( int` **columnIndex**`, int` **scale** `) throws java.sql.SQLException`

### Deprecated

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.BigDecimal` in the Java programming language.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `scale` – an `int` which specifies the number of digits to the right of the decimal point

– **Returns** – a `java.sql.BigDecimal` representation of the column value; if the value is SQL `NULL`, the value returned is `null`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getBigDecimal*
`public java.math.BigDecimal` **getBigDecimal**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `java.math.BigDecimal` with full precision.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column

– **Returns** – a `java.math.BigDecimal` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getBigDecimal*

`public java.math.BigDecimal` **getBigDecimal**`( java.lang.String` **columnName**`, int` **scale** `) throws java.sql.SQLException`

### Deprecated

– **Description**

Gets the value of the designated column in the current row of this `ResultSet` object as a `java.math.BigDecimal` in the Java programming language.

– **Parameters**

  * `columnName` – a `String` which specifies the SQL name of the column
  * `scale` – an `int` which specifies the number of digits to the right of the decimal point

– **Returns** – a `java.math.BigDecimal` representation of the column value; if the value is SQL `NULL`, the value returned is `null`

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getBinaryStream*

`public java.io.InputStream` **getBinaryStream**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**

Gets the value of the designated column in the current row of this `ResultSet` object as a binary stream of uninterpreted bytes. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARBINARY` values.

**Note:** All the data in the returned stream must be read prior to getting the value of any other column. The next call to a `get`*Type* method implicitly closes the stream. Also, a stream may return `0` when the method `InputStream.available` is called whether there is data available or not.

– **Parameters**

  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – a `java.io.InputStream` that delivers the database column value as a stream of uninterpreted bytes; if the value is SQL `NULL`, the value returned is `null`
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getBinaryStream*
`public java.io.InputStream` **getBinaryStream**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a stream of uninterpreted `bytes`. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARBINARY` values.
**Note:** All the data in the returned stream must be read prior to getting the value of any other column. The next call to a `get`*Type* method implicitly closes the stream. Also, a stream may return `0` when the method `available` is called whether there is data available or not.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
– **Returns** – a `java.io.InputStream` that delivers the database column value as a stream of uninterpreted bytes; if the value is SQL `NULL`, the result is `null`
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getBlob*
`public java.sql.Blob` **getBlob**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
Returns the value of the designated column in the current row of this `ResultSet` object as a `Blob` object in the Java programming language.
– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
– **Returns** – a `Blob` object representing the SQL `BLOB` value in the specified column
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

- *getBlob*
  public java.sql.Blob **getBlob**( java.lang.String **columnName** ) throws java.sql.SQLException
  - **Description**
    Returns the value of the designated column in the current row of this `ResultSet` object as a `Blob` object in the Java programming language.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `Blob` object representing the SQL `BLOB` value in the specified column
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

- *getBoolean*
  public boolean **getBoolean**( int **columnIndex** ) throws java.sql.SQLException
  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as a `boolean` in the Java programming language.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - **Returns** – a `boolean` representation of the column value; if the value is SQL `NULL`, the value returned is `false`
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

- *getBoolean*
  public boolean **getBoolean**( java.lang.String **columnName** ) throws java.sql.SQLException
  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as a `boolean` in the Java programming language.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `boolean` representation of the column value; if the value is SQL `NULL`, the value returned is `false`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getByte*
`public byte` **getByte**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `byte` in the Java programming language.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – a `byte` representation of the column value; if the value is SQL `NULL`, the value returned is `0`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

• *getByte*
`public byte` **getByte**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `byte` in the Java programming language.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column

– **Returns** – a `byte` representation of the column value; if the value is SQL `NULL`, the value returned is `0`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getBytes*
`public byte[]` **getBytes**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `byte` array in the Java programming language. The bytes represent the raw values returned by the driver.

– **Parameters**

* `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

- **Returns** – a `byte` array representation of the column value; if the value is SQL `NULL`, the value returned is `null`
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getBytes*

  `public byte[]` **getBytes**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

  - **Description**

    Gets the value of the designated column in the current row of this `ResultSet` object as a `byte` array in the Java programming language. The bytes represent the raw values returned by the driver.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `byte` array representation of the column value; if the value is SQL `NULL`, the value returned is `null`
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getCharacterStream*

  `public java.io.Reader` **getCharacterStream**`( int` **columnIndex** `) throws java.sql.SQLException`

  - **Description**

    Gets the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - **Returns** – a `java.io.Reader` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language.
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getCharacterStream*

  `public java.io.Reader` **getCharacterStream**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
– **Returns** – a `java.io.Reader` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getClob*
`public java.sql.Clob` **getClob**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
Returns the value of the designated column in the current row of this `ResultSet` object as a `Clob` object in the Java programming language.
– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
– **Returns** – a `Clob` object representing an SQL `Clob` value in the specified column
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getClob*
`public java.sql.Clob` **getClob**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Returns the value of the designated column in the current row of this `ResultSet` object as a `Clob` object in the Java programming language.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
– **Returns** – a `Clob` object representing the SQL `CLOB` value in the specified column
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getColumnClass*
  `public java.lang.Class` **getColumnClass**`( int` **columnIndex** `) throws java.sql.SQLException`

  – **Description**
    Returns the class of the items in the specified column. The default implementation returns the Class set using `getColumnClass`. If no class type is set the default implementation returns `Object.class`.
  – **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  – **Returns** – a `Class` object used to specify the class of the data in the column
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getColumnCount*
  `public abstract int` **getColumnCount**`( ) throws java.sql.SQLException`

  – **Description**
    Returns the number of columns in this `ResultSet` object.
  – **Returns** – an `int` which specifies the number of columns
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getConcurrency*
  `public int` **getConcurrency**`( ) throws java.sql.SQLException`

  – **Description**
    Returns the concurrency mode of this `ResultSet` object.
  – **Returns** – an `int` which specifies whether concurrency is read only or for update processes as well. Always returns `CONCUR_READ_ONLY`.
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getCursorName*
  `public java.lang.String` **getCursorName**`( ) throws java.sql.SQLException`

  – **Description**
    Gets the name of the SQL cursor used by this `ResultSet` object. The default implementation throws a SQLException.

– **Returns** – a `String` which specifies the SQL name for this `ResultSet` object's cursor.

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *getDate*

`public java.sql.Date` **getDate**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**

Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.

– **Parameters**

  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – a `java.sql.Date` representation of the column value; if the value is SQL `NULL`, the value returned is `null`

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

• *getDate*

`public java.sql.Date` **getDate**`( int` **columnIndex**`, java.util.Calendar` **cal** `) throws java.sql.SQLException`

– **Description**

Returns the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the date if the underlying database does not store timezone information.

– **Parameters**

  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `cal` – the `java.util.Calendar` object to use in constructing the date

– **Returns** – the column value as a `java.sql.Date` object; if the value is SQL `NULL`, the value returned is `null` in the Java programming language

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getDate*

`public java.sql.Date` **getDate**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column

– **Returns** – a `java.sql.Date` representation of the column value; if the value is SQL `NULL`, the value returned is `null`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getDate*
`public java.sql.Date` **getDate**`( java.lang.String` **columnName**`,`
`java.util.Calendar` **cal** `) throws java.sql.SQLException`

– **Description**
Returns the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Date` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the date if the underlying database does not store timezone information.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
  * `cal` – the `java.util.Calendar` object to use in constructing the date

– **Returns** – the column value as a `java.sql.Date` object; if the value is SQL `NULL`, the value returned is `null` in the Java programming language

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getDouble*
`public double` **getDouble**`( int` **columnIndex** `) throws`
`java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `double` in the Java programming language.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – a `double` representation of the column value; if the value is SQL `NULL`, the value returned is `0`

– **Throws**

* com.imsl.io.AbstractFlatFile.FlatFileSQLException – if a conversion or database access error occurs

---

- *getDouble*
  public double **getDouble**( java.lang.String **columnName** ) throws java.sql.SQLException

  - **Description**
    Gets the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.
  - **Parameters**
    * columnName – a String which specifies the SQL name of the column
  - **Returns** – a double representation of the column value; if the value is SQL NULL, the value returned is 0
  - **Throws**
    * com.imsl.io.AbstractFlatFile.FlatFileSQLException – if a database access error occurs

---

- *getFetchDirection*
  public int **getFetchDirection**( ) throws java.sql.SQLException

  - **Description**
    Returns the fetch direction for this ResultSet object.
  - **Returns** – an int which specifies the current fetch direction for this ResultSet object. Always returns FETCH_FORWARD.
  - **Throws**
    * com.imsl.io.AbstractFlatFile.FlatFileSQLException – if a database access error occurs

---

- *getFetchSize*
  public int **getFetchSize**( ) throws java.sql.SQLException

  - **Description**
    Returns the fetch size for this ResultSet object.
  - **Returns** – an int which specifies the current fetch size for this ResultSet object
  - **Throws**
    * com.imsl.io.AbstractFlatFile.FlatFileSQLException – if a database access error occurs

---

- *getFloat*
  public float **getFloat**( int **columnIndex** ) throws java.sql.SQLException

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `float` in the Java programming language.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – a `float` representation of the column value; if the value is SQL `NULL`, the value returned is `0`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

• *getFloat*
`public float` **getFloat**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `float` in the Java programming language.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column

– **Returns** – a `float` representation of the column value; if the value is SQL `NULL`, the value returned is `0`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getInt*
`public int` **getInt**`( int` **columnIndex** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as an `int` in the Java programming language.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

– **Returns** – an `int` representation of the column value; if the value is SQL `NULL`, the value returned is `0`

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

- *getInt*
  `public int` **getInt**`( java.lang.String` **columnName** `) throws java.sql.SQLException`
  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as an `int` in the Java programming language.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `int` representation of the column value; if the value is SQL `NULL`, the value returned is `0`
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getLong*
  `public long` **getLong**`( int` **columnIndex** `) throws java.sql.SQLException`
  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as a `long` in the Java programming language.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - **Returns** – a `long` representation of the column value; if the value is SQL `NULL`, the value returned is `0`
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

- *getLong*
  `public long` **getLong**`( java.lang.String` **columnName** `) throws java.sql.SQLException`
  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as a `long` in the Java programming language.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `long` representation of the column value; if the value is SQL `NULL`, the value returned is `0`
  - **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

* *getMetaData*
  `public java.sql.ResultSetMetaData` **getMetaData**`( ) throws java.sql.SQLException`
  – **Description**
    Retrieves the number, types and properties of this `ResultSet` object's columns.
  – **Returns** – a `ResultSetMetaData` which provides a description of this `ResultSet` object's columns
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

* *getObject*
  `public abstract java.lang.Object` **getObject**`( int` **columnIndex** `) throws java.sql.SQLException`
  – **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language.
    This method will return the value of the given column as a Java object. The type of the Java object will be the default Java object type corresponding to the column's SQL type, following the mapping for built-in types specified in the JDBC specification.
  – **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  – **Returns** – a `java.lang.Object` representation of the column value
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

* *getObject*
  `public java.lang.Object` **getObject**`( int` **columnIndex**`, java.util.Map` **map** `) throws java.sql.SQLException`
  – **Description**
    Returns the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language. This method uses the given `Map` object for the custom mapping of the SQL structured or distinct type that is being retrieved.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `map` – a `java.util.Map` object that contains the mapping from SQL type names to classes in the Java programming language
– **Returns** – an `Object` in the Java programming language representing the SQL value
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since this method has not been implimented

---

- *getObject*
`public java.lang.Object` **getObject**`( java.lang.String` **columnName** `)`
`throws java.sql.SQLException`

  – **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language.
  This method will return the value of the given column as a Java object. The type of the Java object will be the default Java object type corresponding to the column's SQL type, following the mapping for built-in types specified in the JDBC specification.
  This method may also be used to read datatabase-specific abstract data types.
  In the JDBC 2.0 API, the behavior of the method `getObject` is extended to materialize data of SQL user-defined types. When a column contains a structured or distinct value, the behavior of this method is as if it were a call to: `getObject(columnIndex, this.getStatement().getConnection().getTypeMap())`.
  – **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  – **Returns** – a `java.lang.Object` representation of the column value
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getObject*
`public java.lang.Object` **getObject**`( java.lang.String` **columnName**`,`
`java.util.Map` **map** `) throws java.sql.SQLException`

  – **Description**
  Returns the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language. This method uses the specified `Map` object for custom mapping if appropriate.
  – **Parameters**

- * columnName – a String which specifies the SQL name of the column
  * map – a java.util.Map object that contains the mapping from SQL type names to classes in the Java programming language
- **Returns** – an Object representing the SQL value in the specified column
- **Throws**
  * com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since this method is not implemented

---

- *getRef*
  `public java.sql.Ref` **getRef**`( int` **columnIndex** `) throws java.sql.SQLException`
  - **Description**
    Returns the value of the designated column in the current row of this ResultSet object as a Ref object in the Java programming language.
  - **Parameters**
    * columnIndex – an int which specifies the column. The first column is 1, the second is 2, ...
  - **Returns** – a Ref object representing the SQL REF value in the specified column
  - **Throws**
    * com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since this method has not been implimented

---

- *getRef*
  `public java.sql.Ref` **getRef**`( java.lang.String` **columnName** `) throws java.sql.SQLException`
  - **Description**
    Returns the value of the designated column in the current row of this ResultSet object as a Ref object in the Java programming language.
  - **Parameters**
    * columnName – a String which specifies the SQL name of the column
  - **Returns** – a Ref object representing the SQL REF value in the specified column
  - **Throws**
    * com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since this method is not implemented

---

- *getRow*
  `public int` **getRow**`( ) throws java.sql.SQLException`
  - **Description**
    Retrieves the current row number. The first row is number 1, the second number 2, and so on.

- **Returns** – an `int` which specifies the current row number; `0` if there is no current row
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getShort*

`public short` **getShort**`( int` **columnIndex** `) throws java.sql.SQLException`

- **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `short` in the Java programming language.
- **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
- **Returns** – a `short` representation of the column value; if the value is SQL `NULL`, the value returned is `0`
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

- *getShort*

`public short` **getShort**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

- **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `short` in the Java programming language.
- **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
- **Returns** – a `short` representation of the column value; if the value is SQL `NULL`, the value returned is `0`
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getStatement*

`public java.sql.Statement` **getStatement**`( ) throws java.sql.SQLException`

- **Description**
  Returns the `Statement` object that produced this `ResultSet` object. Since there is not statement, this method always throws an `SQLException`.

– **Returns** – the `Statment` object that produced this `ResultSet` object or `null` if the result set was produced some other way

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *getString*
`public java.lang.String` **getString**`( int` **columnIndex** `) throws java.sql.SQLException`

  – **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `String` in the Java programming language.

  – **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

  – **Returns** – a `String` representation of the column value; if the value is SQL `NULL`, the value returned is `null`

  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getString*
`public java.lang.String` **getString**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

  – **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `String` in the Java programming language.

  – **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column

  – **Returns** – a `String` representation of the column value; if the value is SQL `NULL`, the value returned is `null`

  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getTime*
`public java.sql.Time` **getTime**`( int` **columnIndex** `) throws java.sql.SQLException`

  – **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.

- **Parameters**
  - ∗ `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
- **Returns** – a `java.sql.Time` representation of the column value; if the value is SQL `NULL`, the value returned is `null`
- **Throws**
  - ∗ `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

• *getTime*
`public java.sql.Time` **getTime**`( int` **columnIndex**`, java.util.Calendar` **cal** `) throws java.sql.SQLException`

- **Description**
  Returns the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the time if the underlying database does not store timezone information.
- **Parameters**
  - ∗ `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - ∗ `cal` – the `java.util.Calendar` object to use in constructing the time
- **Returns** – the column value as a `java.sql.Time` object; if the value is SQL `NULL`, the value returned is `null` in the Java programming language
- **Throws**
  - ∗ `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getTime*
`public java.sql.Time` **getTime**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

- **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language.
- **Parameters**
  - ∗ `columnName` – a `String` which specifies the SQL name of the column
- **Returns** – a `java.sql.Time` representation of the column value; if the value is SQL `NULL`, the value returned is `null`
- **Throws**
  - ∗ `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

- *getTime*
`public java.sql.Time` **getTime**`( java.lang.String` **columnName**`, java.util.Calendar` **cal** `) throws java.sql.SQLException`

  – **Description**
  Returns the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Time` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the time if the underlying database does not store timezone information.
  – **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `cal` – the `java.util.Calendar` object to use in constructing the time
  – **Returns** – the column value as a `java.sql.Time` object; if the value is SQL `NULL`, the value returned is `null` in the Java programming language
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

- *getTimestamp*
`public java.sql.Timestamp` **getTimestamp**`( int` **columnIndex** `) throws java.sql.SQLException`

  – **Description**
  Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object in the Java programming language.
  – **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  – **Returns** – a `java.sql.Timestamp` representation of the column value; if the value is SQL `NULL`, the value returned is `null`
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

- *getTimestamp*
`public java.sql.Timestamp` **getTimestamp**`( int` **columnIndex**`, java.util.Calendar` **cal** `) throws java.sql.SQLException`

  – **Description**
  Returns the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value

for the timestamp if the underlying database does not store timezone information.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `cal` – the `java.util.Calendar` object to use in constructing the timestamp
– **Returns** – the column value as a `java.sql.Timestamp` object; if the value is SQL `NULL`, the value returned is `null` in the Java programming language
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getTimestamp*
`public java.sql.Timestamp` **getTimestamp**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
– **Returns** – a `java.sql.Timestamp` representation of the column value; if the value is SQL `NULL`, the value returned is `null`
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *getTimestamp*
`public java.sql.Timestamp` **getTimestamp**`( java.lang.String` **columnName**`, java.util.Calendar` **cal** `) throws java.sql.SQLException`

– **Description**
Returns the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object in the Java programming language. This method uses the given calendar to construct an appropriate millisecond value for the timestamp if the underlying database does not store timezone information.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
  * `cal` – the `java.util.Calendar` object to use in constructing the timestamp
– **Returns** – the column value as a `java.sql.Timestamp` object; if the value is SQL `NULL`, the value returned is `null` in the Java programming language

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getType*
  `public int` **getType**`( ) throws java.sql.SQLException`
  – **Description**
  Returns the type of this `ResultSet` object. The type is determined by the `Statement` object hat created the result set.
  – **Returns** – an `int` which specifies the type of this `ResultSet` object. Always returns `TYPE_FORWARD_ONLY`.
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getUnicodeStream*
  `public java.io.InputStream` **getUnicodeStream**`( int` **columnIndex** `)`
  `throws java.sql.SQLException`

## Deprecated

use `getCharacterStream` in place of `getUnicodeStream`

– **Description**
Gets the value of the designated column in the current row of this `ResultSet` object as as a stream of Unicode characters. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC driver will do any necessary conversion from the database format into Unicode. The byte format of the Unicode stream must be Java UTF-8, as specified in the Java virtual machine specification. **Note:** All the data in the returned stream must be read prior to getting the value of any other column. The next call to a `get`*Type* method implicitly closes the stream. Also, a stream may return `0` when the method `InputStream.available` is called whether there is data available or not.
– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
– **Returns** – a `java.io.InputStream` that delivers the database column value as a stream in Java UTF-8 byte format; if the value is SQL `NULL`, the value returned is `null`
– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getUnicodeStream*
  `public java.io.InputStream` **getUnicodeStream**`( java.lang.String` **columnName** `) throws java.sql.SQLException`

  **Deprecated**

  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as a stream of Unicode characters. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARCHAR` values. The JDBC driver will do any necessary conversion from the database format into Unicode. The byte format of the Unicode stream must be Java UTF-8, as defined in the Java virtual machine specification. **Note:** All the data in the returned stream must be read prior to getting the value of any other column. The next call to a `get`*Type* method implicitly closes the stream. Also, a stream may return `0` when the method `available` is called whether there is data available or not.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `java.io.InputStream` that delivers the database column value as a stream of two-byte Unicode characters. If the value is SQL `NULL`, the value returned is `null`.
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getURL*
  `public java.net.URL` **getURL**`( int` **columnIndex** `) throws java.sql.SQLException`

  - **Description**
    Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.net.URL` object.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - **Returns** – a `java.net.URL` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language
  - **Throws**

    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a conversion or database access error occurs

---

- *getURL*
  `public java.net.URL` **getURL**`( java.lang.String` **columnName** `) throws java.sql.SQLException`
  - **Description**
    Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.net.URL` object.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
  - **Returns** – a `java.net.URL` object that contains the column value; if the value is SQL `NULL`, the value returned is `null` in the Java programming language
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *getWarnings*
  `public java.sql.SQLWarning` **getWarnings**`( ) throws java.sql.SQLException`
  - **Description**
    Returns the first warning reported by calls on this `ResultSet` object. Subsequent warnings on this `ResultSet` object will be chained to the `SQLWarning` object that this method returns.
    The warning chain is automatically cleared each time a new row is read.
    **Note:** This warning chain only covers warnings caused by `ResultSet` methods. Any warning caused by `Statement` methods (such as reading OUT parameters) will be chained on the `Statement` object.
  - **Returns** – the first `SQLWarning` object reported or `null`
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *insertRow*
  `public void` **insertRow**`( ) throws java.sql.SQLException`
  - **Description**
    Inserts the contents of the insert row into this `ResultSet` object and into the database. Since updates are not allowed, this method always throws an `SQLException`.
  - **Throws**

- * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *isAfterLast*
  `public boolean` **isAfterLast**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether the cursor is after the last row in this `ResultSet` object.
  - **Returns** – a `boolean` whose value is `true` if the cursor is after the last row; `false` if the cursor is at any other position or the `ResultSet` contains no rows
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *isBeforeFirst*
  `public boolean` **isBeforeFirst**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether the cursor is before the first row in this `ResultSet` object.
  - **Returns** – a `boolean` whose value is `true` if the cursor is before the first row; `false` if the cursor is at any other position or the `ResultSet` contains no rows
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *isFirst*
  `public boolean` **isFirst**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether the cursor is on the first row of this `ResultSet` object.
  - **Returns** – a `boolean` whose value is `true` if the cursor is on the first row; `false` otherwise
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *isLast*
  `public boolean` **isLast**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether the cursor is on the last row of this `ResultSet` object. Note: Calling the method `isLast` may be expensive because the JDBC driver might need to fetch ahead one row in order to determine whether the current row is the last row in the result set.

– **Returns** – a `boolean` whose value is `true` if the cursor is on the last row; `false` otherwise

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

• *last*

`public boolean` **last**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor to the last row in this `ResultSet` object.

– **Returns** – a `boolean` whose value is `true` if the cursor is on the result set; `false` otherwise

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since this method has not been implemented

---

• *moveToCurrentRow*

`public void` **moveToCurrentRow**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor to the remembered cursor position, usually the current row. Since updates are not allowed, this method always throws an `SQLException`.

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *moveToInsertRow*

`public void` **moveToInsertRow**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor to the insert row. Since updates are not allowed, this method always throws an `SQLException`.

– **Throws**

  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *next*

`public boolean` **next**`( ) throws java.sql.SQLException`

– **Description**
Moves the cursor down one row from its current position. A `ResultSet` cursor is initially positioned before the first row; the first call to the method `next` makes

the first row the current row; the second call makes the second row the current row, and so on.

If an input stream is open for the current row, a call to the method `next` will implicitly close it. A `ResultSet` object's warning chain is cleared when a new row is read.

- **Returns** – a `boolean`, `true` if the new current row is valid; `false` if there are no more rows
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *previous*

  `public boolean` **`previous`**`( ) throws java.sql.SQLException`

  - **Description**

    Moves the cursor to the previous row in this `ResultSet` object.
  - **Returns** – a `boolean` whose value is `true` if the cursor is on the result set; `false` otherwise
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since only forward operations are allowed

---

- *refreshRow*

  `public void` **`refreshRow`**`( ) throws java.sql.SQLException`

  - **Description**

    Refreshes the current row with its most recent value in the database. Since updates are not allowed, this method always throws an `SQLException`.
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *relative*

  `public boolean` **`relative`**`( int` **`rows`** `) throws java.sql.SQLException`

  - **Description**

    Moves the cursor a relative number of rows, either positive or negative.
  - **Parameters**
    * `rows` – an `int` which specifies the number of rows in the `ResultSet` object to advance or regress
  - **Returns** – a `boolean` whose value is `true` if the cursor is on the result set; `false` otherwise
  - **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since only forward operations are allowed

---

- *rowDeleted*
  `public boolean` **rowDeleted**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether a row has been deleted. Since updates are not allowed, this always returns `false`.
  - **Returns** – a `boolean` which indicates whether a row has been deleted. Always returns `false` since updates are not allowed.
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *rowInserted*
  `public boolean` **rowInserted**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether the current row has had an insertion. Since updates are not allowed, this always returns `false`.
  - **Returns** – a `boolean` which indicates whether the current row had an insertion. Always returns `false` since updates are not allowed.
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *rowUpdated*
  `public boolean` **rowUpdated**`( ) throws java.sql.SQLException`
  - **Description**
    Indicates whether the current row has been updated. Since updates are not allowed, this always returns `false`.
  - **Returns** – a `boolean` which indicates whether a row has been updated. Always returns `false` since updates are not allowed.
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

---

- *setColumnClass*
  `protected void` **setColumnClass**`( int` **columnIndex**`, java.lang.Class` **columnClass** `)`

– **Description**
  Sets a column class.
– **Parameters**
  * `columnIndex` – an `int` specifying the index of a column
  * `columnClass` – a `Class` object used to specify the class of the data in the column

---

• *setColumnName*
`protected void` **setColumnName**( `int` **columnIndex**, `java.lang.String` **columnName** )

– **Description**
  Sets a column name. A subclass can define its own mechanism for naming columns. An alternate mechanism would require overriding the methods `findColumn` and `findColumnName`.
– **Parameters**
  * `columnIndex` – an `int` specifying the column index of the column to be named
  * `columnName` – a `String` specifying the name of the column

---

• *setFetchDirection*
`public void` **setFetchDirection**( `int` **direction** ) `throws java.sql.SQLException`

– **Description**
  Gives a hint as to the direction in which the rows in this `ResultSet` object will be processed.
– **Parameters**
  * `direction` – an `int` which specifies the expected direction this `ResultSet` object is to be processed
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if the fetch direction is not `FETCH_FORWARD`

---

• *setFetchSize*
`public void` **setFetchSize**( `int` **rows** ) `throws java.sql.SQLException`

– **Description**
  Gives the JDBC driver a hint as to the number of rows that should be fetched from the database when more rows are needed for this `ResultSet` object. If the fetch size specified is zero, the JDBC driver ignores the value and is free to make its own best guess as to what the fetch size should be. The default value is set by the `Statement` object that created the result set. The fetch size may be changed at any time.

– **Parameters**
  * `rows` – an `int` which specifies the number of rows to fetch
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs or the condition `0 = rows = this.getMaxRows()` is not satisfied

---

- *setWarning*
`protected void` **setWarning**`( java.sql.SQLWarning` **warning** `)`
  – **Description**
  Sets a `SQLWarning`.
  – **Parameters**
    * `warning` – a `SQLWarning` that is to be added to this object.

---

- *updateArray*
`public void` **updateArray**`( int` **column**`, java.sql.Array x ) throws java.sql.SQLException`
  – **Description**
  Updates the designated column with an `Array` value. Since updates are not allowed, this method always throws an `SQLException`.
  – **Parameters**
    * `column` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – a `java.sql.Array` which specifies the new column value
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateArray*
`public void` **updateArray**`( java.lang.String` **columnName**`, java.sql.Array x ) throws java.sql.SQLException`
  – **Description**
  Updates the designated column with an `Array` value. Since updates are not allowed, this method always throws an `SQLException`.
  – **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – a `java.sql.Array` which specifies the new column value
  – **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

- *updateAsciiStream*
  `public void` **updateAsciiStream**`(` `int` **columnIndex**`,` `java.io.InputStream x,` `int` **length** `) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an ascii stream value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – a `InputStream` which specifies the new column value
    * `length` – an `int` which specifies the stream length
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateAsciiStream*
  `public void` **updateAsciiStream**`(` `java.lang.String` **columnName**`,` `java.io.InputStream x,` `int` **length** `) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an ascii stream value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – a `InputStream` which specifies the new column value
    * `length` – an `int` which specifies the stream length
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBigDecimal*
  `public void` **updateBigDecimal**`(` `int` **columnIndex**`,` `java.math.BigDecimal x ) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with a `java.math.BigDecimal` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – a `java.math.BigDecimal` which specifies the new column value

- **Throws**
  - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBigDecimal*

  `public void` **updateBigDecimal**`( java.lang.String` **columnName**`, java.math.BigDecimal` **x** `) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a `java.sql.BigDecimal` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    - `columnName` – a `String` which specifies the SQL name of the column
    - `x` – a `java.sql.BigDecimal` which specifies the new column value
  - **Throws**
    - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBinaryStream*

  `public void` **updateBinaryStream**`( int` **columnIndex**`, java.io.InputStream` **x**`, int` **length** `) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a binary stream value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    - `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    - `x` – a `InputStream` which specifies the new column value
    - `length` – an `int` which specifies the stream length
  - **Throws**
    - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBinaryStream*

  `public void` **updateBinaryStream**`( java.lang.String` **columnName**`, java.io.InputStream` **x**`, int` **length** `) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a binary stream value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    - `columnName` – a `String` which specifies the SQL name of the column
    - `x` – a `InputStream` which specifies the new column value

* `length` – an `int` which specifies the stream length
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBlob*

  `public void` **updateBlob**`( int` **column**`, java.sql.Blob x ) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with an `java.sql.Blob` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `column` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – a `java.sql.Blob` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBlob*

  `public void` **updateBlob**`( java.lang.String` **columnName**`, java.sql.Blob x ) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with an `java.sql.Blob` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – a `java.sql.Blob` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateBoolean*

  `public void` **updateBoolean**`( int` **columnIndex**`, boolean x ) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a `boolean` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...

* `x` – a `boolean` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateBoolean*

`public void` **updateBoolean**`( java.lang.String` **columnName**`, boolean x ) throws java.sql.SQLException`

– **Description**

Updates the designated column with a `boolean` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnName` – a `String` which specifies the SQL name of the column
* `x` – a `boolean` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateByte*

`public void` **updateByte**`( int` **columnIndex**`, byte x ) throws java.sql.SQLException`

– **Description**

Updates the designated column with a `byte` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
* `x` – a `byte` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateByte*

`public void` **updateByte**`( java.lang.String` **columnName**`, byte x ) throws java.sql.SQLException`

– **Description**

Updates the designated column with a `byte` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnName` – a `String` which specifies the SQL name of the column

* x – a byte which specifies the new column value

– **Throws**
  * com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since updates are not allowed

---

• *updateBytes*
public void **updateBytes**( int **columnIndex**, byte[] x ) throws java.sql.SQLException

– **Description**
Updates the designated column with a byte array value. Since updates are not allowed, this method always throws an SQLException.

– **Parameters**
  * columnIndex – an int which specifies the column. The first column is 1, the second is 2, ...
  * x – a byte which specifies the new column value

– **Throws**
  * com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since updates are not allowed

---

• *updateBytes*
public void **updateBytes**( java.lang.String **columnName**, byte[] x ) throws java.sql.SQLException

– **Description**
Updates the designated column with a byte value. Since updates are not allowed, this method always throws an SQLException.

– **Parameters**
  * columnName – a String which specifies the SQL name of the column
  * x – a byte which specifies the new column value

– **Throws**
  * com.imsl.io.AbstractFlatFile.FlatFileSQLException – is always thrown since updates are not allowed

---

• *updateCharacterStream*
public void **updateCharacterStream**( int **columnIndex**, java.io.Reader x, int **length** ) throws java.sql.SQLException

– **Description**
Updates the designated column with a character stream value. Since updates are not allowed, this method always throws an SQLException.

– **Parameters**
  * columnIndex – an int which specifies the column. The first column is 1, the second is 2, ...

* `x` – a `Reader` which specifies the new column value
* `length` – an `int` which specifies the stream length

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateCharacterStream*

`public void` **updateCharacterStream**`( java.lang.String` **columnName**`, java.io.Reader` **reader**`, int` **length** `) throws java.sql.SQLException`

– **Description**

Updates the designated column with a character stream value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnName` – a `String` which specifies the SQL name of the column
* `reader` – a `Reader` which specifies the new column value
* `length` – an `int` which specifies the stream length

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateClob*

`public void` **updateClob**`( int` **column**`, java.sql.Clob` **x** `) throws java.sql.SQLException`

– **Description**

Updates the designated column with an `java.sql.Clob` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `column` – an `int` which specifies the column. The first column is 1, the second is 2, ...
* `x` – a `java.sql.Clob` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateClob*

`public void` **updateClob**`( java.lang.String` **columnName**`, java.sql.Clob` **x** `) throws java.sql.SQLException`

– **Description**

Updates the designated column with an `java.sql.Clob` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnName` – a `String` which specifies the SQL name of the column
* `x` – a `java.sql.Clob` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateDate*

`public void` **updateDate**`( int` **columnIndex**`, java.sql.Date x ) throws java.sql.SQLException`

– **Description**

Updates the designated column with a `java.sql.Date` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
* `x` – a `java.sql.Date` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateDate*

`public void` **updateDate**`( java.lang.String` **columnName**`, java.sql.Date x ) throws java.sql.SQLException`

– **Description**

Updates the designated column with a `java.sql.Date` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnName` – a `String` which specifies the SQL name of the column
* `x` – a `java.sql.Date` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateDouble*

`public void` **updateDouble**`( int` **columnIndex**`, double x ) throws java.sql.SQLException`

– **Description**

Updates the designated column with a `double` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – a `double` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateDouble*

  `public void` **updateDouble**`( java.lang.String` **columnName**`, double` **x** `) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a `double` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – a `double` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateFloat*

  `public void` **updateFloat**`( int` **columnIndex**`, float` **x** `) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a `float` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – a `float` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateFloat*

  `public void` **updateFloat**`( java.lang.String` **columnName**`, float` **x** `) throws java.sql.SQLException`

  - **Description**

    Updates the designated column with a `float` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**

    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – a `float` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateInt*
  `public void` **updateInt**`( int` **columnIndex**`, int x ) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an `int` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    * `x` – an `int` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateInt*
  `public void` **updateInt**`( java.lang.String` **columnName**`, int x ) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an `int` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – an `int` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateLong*
  `public void` **updateLong**`( int` **columnIndex**`, long x ) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with a `long` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**

  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `x` – a `long` which specifies the new column value
- **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateLong*
`public void` **updateLong**`( java.lang.String` **columnName**`, long` **x** `) throws java.sql.SQLException`
  - **Description**
  Updates the designated column with a `long` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnName` – a `String` which specifies the SQL name of the column
    * `x` – a `long` which specifies the new column value
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateNull*
`public void` **updateNull**`( int` **columnIndex** `) throws java.sql.SQLException`
  - **Description**
  Gives a nullable column a `null` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - **Throws**
    * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateNull*
`public void` **updateNull**`( java.lang.String` **columnName** `) throws java.sql.SQLException`
  - **Description**
  Updates the designated column with a `null` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**

- * `columnName` – a `String` which specifies the SQL name of the column
  - **Throws**
    - * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateObject*
  `public void` **updateObject**`( int` **columnIndex**`, java.lang.Object` **x** `)`
  `throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    - * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    - * `x` – an `Object` which specifies the new column value
  - **Throws**
    - * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateObject*
  `public void` **updateObject**`( int` **columnIndex**`, java.lang.Object` **x**`, int` **scale** `) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws an `SQLException`.
  - **Parameters**
    - * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
    - * `x` – an `Object` which specifies the new column value
    - * `scale` – for `java.sql.Types.DECIMAL` or `java.sql.Types.NUMERIC` types, this is the number of digits after the decimal point. For all other types this value will be ignored.
  - **Throws**
    - * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateObject*
  `public void` **updateObject**`( java.lang.String` **columnName**`,`
  `java.lang.Object` **x** `) throws java.sql.SQLException`
  - **Description**
    Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
  * `x` – a `java.sql.Object` which specifies the new column value
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateObject*
`public void` **updateObject**`( java.lang.String` **columnName**`, java.lang.Object` **x**`, int` **scale** `) throws java.sql.SQLException`

– **Description**
Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws an `SQLException`.
– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
  * `x` – an `Object` which specifies the new column value
  * `scale` – for `java.sql.Types.DECIMAL` or `java.sql.Types.NUMERIC` types, this is the number of digits after the decimal point. For all other types this value will be ignored.
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateRef*
`public void` **updateRef**`( int` **column**`, java.sql.Ref` **x** `) throws java.sql.SQLException`

– **Description**
Updates the designated column with an `java.sql.Ref` value. Since updates are not allowed, this method always throws an `SQLException`.
– **Parameters**
  * `column` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `x` – a `java.sql.Ref` which specifies the new column value
– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateRef*
`public void` **updateRef**`( java.lang.String` **columnName**`, java.sql.Ref` **x** `) throws java.sql.SQLException`

– **Description**
Updates the designated column with an `java.sql.Ref` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnName` – a `String` which specifies the SQL name of the column
* `x` – a `java.sql.Ref` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateRow*
`public void` **updateRow**`( ) throws java.sql.SQLException`

– **Description**
Updates the underlying database with the new contents of the current row of this `ResultSet` object. Since updates are not allowed, this method always throws an `SQLException`.

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateShort*
`public void` **updateShort**`( int` **columnIndex**`, short x ) throws java.sql.SQLException`

– **Description**
Updates the designated column with a `short` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**

* `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
* `x` – a `short` which specifies the new column value

– **Throws**

* `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateShort*
`public void` **updateShort**`( java.lang.String` **columnName**`, short x ) throws java.sql.SQLException`

– **Description**
Updates the designated column with a `short` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
  * `x` – a `short` which specifies the new column value

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateString*
`public void` **updateString**`( int` **columnIndex**`, java.lang.String x ) throws java.sql.SQLException`

– **Description**
Updates the designated column with a `String` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**
  * `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  * `x` – a `String` which specifies the new column value

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateString*
`public void` **updateString**`( java.lang.String` **columnName**`, java.lang.String x ) throws java.sql.SQLException`

– **Description**
Updates the designated column with a `String` value. Since updates are not allowed, this method always throws an `SQLException`.

– **Parameters**
  * `columnName` – a `String` which specifies the SQL name of the column
  * `x` – a `String` which specifies the new column value

– **Throws**
  * `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

• *updateTime*
`public void` **updateTime**`( int` **columnIndex**`, java.sql.Time x ) throws java.sql.SQLException`

– **Description**
Updates the designated column with a `java.sql.Time` value. Since updates are not allowed, this method always throws an `SQLException`.

- **Parameters**
  - `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - `x` – a `java.sql.Time` which specifies the new column value
- **Throws**
  - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateTime*

`public void` **updateTime**`( java.lang.String` **columnName**`, java.sql.Time x ) throws java.sql.SQLException`

- **Description**

  Updates the designated column with a `java.sql.Time` value. Since updates are not allowed, this method always throws an `SQLException`.
- **Parameters**
  - `columnName` – a `String` which specifies the SQL name of the column
  - `x` – a `java.sql.Time` which specifies the new column value
- **Throws**
  - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateTimestamp*

`public void` **updateTimestamp**`( int` **columnIndex**`, java.sql.Timestamp x ) throws java.sql.SQLException`

- **Description**

  Updates the designated column with a `java.sql.Timestamp` value. Since updates are not allowed, this method always throws an `SQLException`.
- **Parameters**
  - `columnIndex` – an `int` which specifies the column. The first column is 1, the second is 2, ...
  - `x` – a `java.sql.Timestamp` which specifies the new column value
- **Throws**
  - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *updateTimestamp*

`public void` **updateTimestamp**`( java.lang.String` **columnName**`, java.sql.Timestamp x ) throws java.sql.SQLException`

- **Description**

  Updates the designated column with a `java.sql.Timestamp` value. Since updates are not allowed, this method always throws an `SQLException`.

- **Parameters**
  - `columnName` – a `String` which specifies the SQL name of the column
  - `x` – a `java.sql.Timestamp` which specifies the new column value
- **Throws**
  - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – is always thrown since updates are not allowed

---

- *wasNull*

  `public boolean` **wasNull**`( ) throws java.sql.SQLException`

  - **Description**

    Reports whether the last column read had a value of SQL `NULL`. Note that you must first call one of the `get`*Type* methods on a column to try to read its value and then call the method `wasNull` to see if the value read was SQL `NULL`.
  - **Returns** – a `boolean`, `true` if the last column value read was SQL `NULL` and `false` otherwise
  - **Throws**
    - `com.imsl.io.AbstractFlatFile.FlatFileSQLException` – if a database access error occurs

# *class* FlatFile

Reads a text file as a ResultSet.

FlatFile extends AbstractFlatFile to handle text flat files.

As the file is read, it is split into lines using the readLine method. Each line is then split into tokens using a Tokenizer. Finally, each token string is converted into an Object using a Parser.

Parser is an interface defined within this class for converting a String into an Object. Parser objects for standard types are defined as static members of this class. By default, for each column its class is used to select one of these predefined parsers to parse that column.

## Declaration

| |
|---|
| public class com.imsl.io.FlatFile<br>**extends** com.imsl.io.AbstractFlatFile (page 771) |

## Inner Class

### *interface* **FlatFile.Parser**

Defines a method that parses a String into an Object.

### Declaration

public static interface com.imsl.io.FlatFile.Parser

### Method

---

- *parse*
  `java.lang.Object` **`parse`**`( java.lang.String` **`input`** `) throws java.sql.SQLException`
  - **Description**
    Parse a String into an Object.
  - **Parameters**
    * `input` – is the String to be parsed.
  - **Returns** – the value of the String as an Object.

### Fields

---

- public static final `FlatFile.Parser` **PARSE_BYTE**
  - Implements a Parser that converts a String to a Byte.
- public static final `FlatFile.Parser` **PARSE_SHORT**
  - Implements a Parser that converts a String to a Short.
- public static final `FlatFile.Parser` **PARSE_INTEGER**
  - Implements a Parser that converts a String to a Integer.
- public static final `FlatFile.Parser` **PARSE_LONG**
  - Implements a Parser that converts a String to a Long.
- public static final `FlatFile.Parser` **PARSE_FLOAT**

– Implements a Parser that converts a String to a Float.

- public static final `FlatFile.Parser` **PARSE_DOUBLE**
  – Implements a Parser that converts a String to a Double.

## Constructors

---

- *FlatFile*
  `public` **FlatFile**`( java.io.BufferedReader` **reader** `) throws java.io.IOException`
  – **Description**
  Creates a FlatFile with the CSV tokenizer. The CSV Tokenizer is for reading comma separated value files.
  – **Parameters**
    * `reader` – is the stream to be read.

- *FlatFile*
  `public` **FlatFile**`( java.io.BufferedReader` **reader**`, Tokenizer` **tokenizer** `) throws java.io.IOException`
  – **Description**
  Creates a FlatFile from a BufferedReader.
  – **Parameters**
    * `reader` – is the stream to be read.
    * `tokenizer` – splits a text line into tokens, one per column.

- *FlatFile*
  `public` **FlatFile**`( java.lang.String` **filename** `) throws java.io.IOException`
  – **Description**
  Creates a FlatFile from a CSV file. A CSV file is a comma separated value file.
  – **Parameters**
    * `filename` – is the name of the file to be read.

- *FlatFile*
  `public` **FlatFile**`( java.lang.String` **filename**`, Tokenizer` **tokenizer** `) throws java.io.IOException`
  – **Description**
  Creates a FlatFile from a file with the default tokenizer.

– **Parameters**
  * `filename` – is the name of the file to be read.
  * `tokenizer` – is the Tokenizer used to split lines into token strings.

## Methods

---

- *doGetBytes*
  `protected byte[]` **doGetBytes**`( int` **columnIndex** `) throws java.sql.SQLException`
  – **Description**
  Gets the value of the designated column in the current row as a `byte` array.
  – **Parameters**
    * `columnIndex` – the first column is 1, the second is 2, ...
  – **Returns** – the column value; if the value is SQL `NULL`, the value returned is `null`
  – **Throws**
    * `java.sql.SQLException` – if a database access error occurs

---

- *doNext*
  `protected boolean` **doNext**`( ) throws java.sql.SQLException`
  – **Description**
  Moves the cursor down one row from its current position. A `ResultSet` cursor is initially positioned before the first row; the first call to the method `next` makes the first row the current row; the second call makes the second row the current row, and so on.
  – **Returns** – `true` if the new current row is valid; `false` if there are no more rows
  – **Throws**
    * `java.sql.SQLException` – if a database access error occurs

---

- *getColumnCount*
  `public int` **getColumnCount**`( ) throws java.sql.SQLException`
  – **Description**
  Returns the number of columns in this `ResultSet` object.
  – **Returns** – the number of columns
  – **Throws**
    * `java.sql.SQLException` – if a database access error occurs

---

- *getObject*
  `public java.lang.Object` **getObject**`( int` **columnIndex** `) throws java.sql.SQLException`

  - **Description**
    Gets the value of the designated column in the current row of this `ResultSet` object as an `Object` in the Java programming language.
    This method will return the value of the given column as a Java object. The type of the Java object will be the default Java object type corresponding to the column's SQL type, following the mapping for built-in types specified in the JDBC specification.
    This method may also be used to read datatabase-specific abstract data types.
    In the JDBC 2.0 API, the behavior of method `getObject` is extended to materialize data of SQL user-defined types. When a column contains a structured or distinct value, the behavior of this method is as if it were a call to: `getObject(columnIndex, this.getStatement().getConnection().getTypeMap())`.
  - **Parameters**
    * `columnIndex` – the first column is 1, the second is 2, ...
  - **Returns** – a `java.lang.Object` holding the column value
  - **Throws**
    * `java.sql.SQLException` – if a database access error occurs

---

- *readLine*
  `protected java.lang.String` **readLine**`( ) throws java.io.IOException`

  - **Description**
    Reads and returns a line from the input.

---

- *setColumnClass*
  `protected void` **setColumnClass**`( int` **columnIndex**`, java.lang.Class` **columnClass** `)`

  - **Description copied from AbstractFlatFile** (**page 771**)
    Sets a column class.
  - **Parameters**
    * `columnIndex` – an `int` specifying the index of a column
    * `columnClass` – a `Class` object used to specify the class of the data in the column

---

- *setColumnParser*
  `protected void` **setColumnParser**`( int` **columnIndex**`, FlatFile.Parser` **columnParser** `)`

- **Description**
  Sets the Parser for the specified column.
- **Parameters**
  * `columnIndex` – the column index of the column
  * `columnParser` – is the Parser to be used to parse entries in the specified column.

---

- *setDateColumnParser*
  `protected void` **setDateColumnParser**`( int` **columnIndex**`, java.lang.String` **pattern**`, java.util.Locale` **locale** `)`
  - **Description**
    Creates for a pattern string and sets the Parser for the specified column.
  - **Parameters**
    * `pattern` – is used to construct a java.text.SimpleDateFormat object used to parse the column.
    * `locale` – is the Locale for the date format Parser.

## Example: Fisher Iris Data Set

The Fisher iris data set is frequently used as a sample statistical data set. This example reads the data set in a CVS (comma separated value) format.

The first few lines of the data set are as follows:

```
Species,Sepal Length,Sepal Width,Petal Length,Petal Width
 1.0, 5.1, 3.5, 1.4, .2
 1.0, 4.9, 3.0, 1.4, .2
 1.0, 4.7, 3.2, 1.3, .2
 1.0, 4.6, 3.1, 1.5, .2
 1.0, 5.0, 3.6, 1.4, .2
 1.0, 5.4, 3.9, 1.7, .4
```

The first line contains the column names, with a comma as the separator. The rest of the lines contain `double` data, one observation per line, with comma as a separator.

The class `FlatFileEx1` extends `com.imsl.io.FlatFile` . The `FlatFileEx1` constructor constructs a `BufferedReader` object and calls the `com.imsl.io.FlatFile` constructor. It then reads the line containing the column names. The column names are parsed and used to set the column names in `com.imsl.io.FlatFile`. All of the columns are also set to type `Double`.

The class `FlatFileEx1` is used in the method `main`. The data set is assumed to be in a file called “FisherIris.csv” in the same location as the example class file, so the `getResourceAsStream` can be used to open the file as a stream. A `com.imsl.stat.Summary` is created and used to compute statistics for the “Sepal Width” column.

```
import com.imsl.io.FlatFile;
import com.imsl.stat.Summary;
import java.io.*;
import java.sql.SQLException;
import java.util.StringTokenizer;


public class FlatFileEx1 extends FlatFile {
   public FlatFileEx1(InputStream is) throws IOException {
      super(new BufferedReader(new InputStreamReader(is)));
      String line = readLine();
      StringTokenizer st = new StringTokenizer(line, ",");
      for (int j = 0;  st.hasMoreTokens();  j++) {
         setColumnName(j+1, st.nextToken().trim());
         setColumnClass(j, Double.class);
      }
   }

   public static void main(String[] args) throws SQLException, IOException {
      InputStream is = FlatFileEx1.class.getResourceAsStream("FisherIris.csv");
      FlatFileEx1 iris = new FlatFileEx1(is);

      Summary summary = new Summary();
      while (iris.next()) {
         summary.update(iris.getDouble("Sepal Width"));
      }

      System.out.println("Sepal Width mean " + summary.getMean());
      System.out.println("Sepal Width variance " + summary.getVariance());
   }
}
```

## Output

```
Sepal Width mean 3.057333333333334
Sepal Width variance 0.18871288888888907
```

### Reference

Fisher, R.A. (1936), *The use of multiple measurements in taxonomic problems*, The Annals of Eugenics, 7, 179-188.

### Example: Space Separated Data

This example reads a set of stock prices in a space separated form.

The first few lines of the data set are as follows:

```
Date        Open    High    Low     Close   Volume
28-Apr-03   3.3     3.34    3.27    3.33    37224400
25-Apr-03   3.35    3.38    3.25    3.26    57117400
24-Apr-03   3.32    3.40    3.30    3.38    47019800
23-Apr-03   3.34    3.44    3.30    3.37    63243800
22-Apr-03   3.24    3.38    3.22    3.36    67392500
```

The first line contains the column names, with a comma as the separator. The rest of the lines contain data, one day per line. The first column is `Date` data and the last column is `int` data. All of the rest is `double` data. The data's class is set for each column. The parser is explicitly set for the date column, because it cannot be guessed by `FlatFile`. The date's locale is set to `US`, so that the example will work with a different default locale.

A `Tokenizer` is created and used that counts multiple separators (spaces) as one separator.

The class `FlatFileEx2` extends `com.imsl.io.FlatFile`. The `FlatFileEx2` constructor reads the line containing the column names, parses the names, and sets the column names.

The class `FlatFileEx2` is used in the method `main`. The data set is assumed to be in a file called "SUNW.txt" in the same location as the example class file, so the `getResourceAsStream` method can be used to open the file as a stream. Some of the columns are printed out for each stock price.

```
import com.imsl.io.*;
import java.text.DateFormat;
import java.io.*;
import java.sql.SQLException;
import java.util.StringTokenizer;
import java.sql.Date;


public class FlatFileEx2 extends FlatFile {
    static DateFormat dateFormat = DateFormat.getDateInstance();

    public FlatFileEx2(BufferedReader br, Tokenizer tokenizer) throws IOException {
        super(br, tokenizer);
```

```
        String line = readLine();
        StringTokenizer st = new StringTokenizer(line, " ", false);
        for (int j = 0;  st.hasMoreTokens();  j++) {
            setColumnName(j+1, st.nextToken().trim());
        }
        setColumnClass(1, Date.class); // Date
        setDateColumnParser(1, "dd-MMM-yy", java.util.Locale.US);
        setColumnClass(2, Double.class);  // Open
        setColumnClass(3, Double.class);  // High
        setColumnClass(4, Double.class);  // Low
        setColumnClass(5, Double.class);  // Close
        setColumnClass(6, Integer.class); // Volume
    }


    public static void main(String[] args) throws SQLException, IOException {
        InputStream is = FlatFileEx2.class.getResourceAsStream("SUNW.txt");
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        Tokenizer tokenizer = new Tokenizer(" ", (char)0, true);
        FlatFileEx2 reader = new FlatFileEx2(br, tokenizer);

        while (reader.next()) {
            Date date = reader.getDate("Date");
            double close = reader.getDouble("Close");
            int volume = reader.getInt("Volume");
            System.out.println(dateFormat.format(date) + "  " + close + "  " + volume);
        }
    }
}
```

### Output

```
Apr 28, 2003  3.33  37224400
Apr 25, 2003  3.26  57117400
Apr 24, 2003  3.38  47019800
Apr 23, 2003  3.37  63243800
Apr 22, 2003  3.36  67392500
Apr 21, 2003  3.28  58523800
Apr 17, 2003  3.24  101856900
Apr 16, 2003  3.32  54912900
Apr 15, 2003  3.35  33604200
```

```
Apr 14, 2003  3.29  38851800
Apr 11, 2003  3.31  38424000
Apr 10, 2003  3.37  38608500
Apr 9, 2003  3.28  50669700
Apr 8, 2003  3.31  46106400
Apr 7, 2003  3.36  47462900
Apr 4, 2003  3.34  48689900
Apr 3, 2003  3.48  38304400
Apr 2, 2003  3.49  48510200
Apr 1, 2003  3.36  38823800
Mar 31, 2003  3.26  38949300
Mar 28, 2003  3.42  27186700
Mar 27, 2003  3.56  40054700
Mar 26, 2003  3.5  30952400
Mar 25, 2003  3.45  63787600
Mar 24, 2003  3.45  34645400
Mar 21, 2003  3.72  53745900
Mar 20, 2003  3.65  47358500
Mar 19, 2003  3.57  51280600
Mar 18, 2003  3.55  51727400
Mar 17, 2003  3.53  69296600
Mar 14, 2003  3.24  59278900
Mar 13, 2003  3.31  58360700
Mar 12, 2003  3.08  71790300
Mar 11, 2003  3.21  42498400
```

## *class* **Tokenizer**

Breaks a line into tokens.

The Tokenizer divides a line into tokens separated by deliminators. There can be any number of deliminators set. All of the deliminators are treated equally.

There can be at most one quote character set. If it is set then deliminators inside of a quoted string are treated as part of the string and not as deliminators. The quotes are not returned as part of the token. To escape a quote, repeat it.

### Declaration

public class com.imsl.io.Tokenizer
**extends** java.lang.Object

## Constructor

- *Tokenizer*
  `public` **Tokenizer**`( java.lang.String` **deliminators**`, char` **quote**`, boolean` **mergeMultipleDeliminators** `)`
  - **Description**
    Creates a Tokenizer.
  - **Parameters**
    * `deliminators` – is a String containing the deliminator characters.
    * `quote` – is a char containing the quote character. If 0 then quoting is disabled.
    * `mergeMultipleDeliminators` – is true if multiple consecutive deliminators are to be treated as a single deliminator.

## Methods

- *countTokens*
  `public int` **countTokens**`( )`
  - **Description**
    Returns the number of times that the nextToken method can be called without generating an exception.

---

- *hasMoreTokens*
  `public boolean` **hasMoreTokens**`( )`
  - **Description**
    Returns true if a call to nextToken will not generate an exception.

---

- *nextToken*
  `public java.lang.String` **nextToken**`( )`
  - **Description**
    Returns the next token.
  - **Returns** – the next token.
  - **Throws**
    * `java.util.NoSuchElementException` – if there are no more tokens to be returned.

---

- *parse*
  `public void` **parse**`( java.lang.String` **line** `)`
  - **Description**
    Sets the line to be tokenized. Any tokens left from the previous line are discarded.
  - **Parameters**
    - `line` – is the line to be tokenized.

# Chapter 23

# Finance

**Classes**



## Usage Notes

Users can perform financial computations by using pre-defined data types. Most of the financial functions require one or more of the following:

- Date
- Number of payments per year
- A variable to indicate when payments are due
- Day count basis

The Bond class provides constants to indicate the number of payments for each year.

| Class member | Meaning |
|---|---|
| `Bond.ANNUAL` | One payment per year (Annual payment) |
| `Bond.SEMIANNUAL` | Two payments per year (Semi-annual payment) |
| `Bond.QUARTERLY` | Four payments per year (Quarterly payment) |

The `Finance` class provides constants to indicate when payments are due.

| Class member | Meaning |
|---|---|
| `Finance.AT_END_OF_PERIOD` | Payments are due at the end of the period |
| `Finance.AT_BEGINNING_OF_PERIOD` | Payments are due at the beginning of the period |

The `DayCountBasis` class provides constants to indicate the type of day count basis. Day count basis is the method for computing the number of days between two dates.

| Class member | Day count basis |
|---|---|
| `DayCountBasis.BasisNASD` | US (NASD) 30/360 |
| `DayCountBasis.BasisActualActual` | Actual/Actual |
| `DayCountBasis.BasisActual360` | Actual/360 |
| `DayCountBasis.BasisActual365` | Actual/365 |
| `DayCountBasis.Basis30e360` | European 30/360 |

### Additional Information

In preparing the finance and bond functions we incorporated standards used by *SIA Standard Securities Calculation Methods*.

More detailed information on finance and bond functionality can be found in the following manuals:

- *SIA Standard Securities Calculation Methods* 1993, vols. 1 and 2, Third Edition
- *Microsoft Excel 5*, *Worksheet Function Reference.*

## *interface* BasisPart

Component of com.imsl.finance.DayCountBasis. The day count basis consists of a month basis and a yearly basis. Each of these components implements this interface.

## Declaration

public interface com.imsl.finance.BasisPart

## Methods

---

- *daysBetween*
  `int` **daysBetween**`(` `java.util.GregorianCalendar` **date1**`,` `java.util.GregorianCalendar` **date2** `)`
  - **Description**
    Returns the number of days from `date1` to `date2`.
  - **Parameters**
    * `date1` – a `GregorianCalendar` which specifies the initial date
    * `date2` – a `GregorianCalendar` which specifies the final date
  - **Returns** – an `int` indicating the number of days from `date1` to `date2`.

- *daysInPeriod*
  `double` **daysInPeriod**`(` `java.util.GregorianCalendar` **date**`,` `int` **frequency** `)`
  - **Description**
    Returns the number of days in a coupon period.
  - **Parameters**
    * `date` – a `GregorianCalendar` which specifies the final date of the coupon period
    * `frequency` – is the number of coupon periods per year. This is typically 1, 2 or 4.
  - **Returns** – an `int` which specifies the number of days in the coupon period

- *getDaysInYear*
  `int` **getDaysInYear**`(` `java.util.GregorianCalendar` **settlement**`,` `java.util.GregorianCalendar` **maturity** `)`
  - **Description**
    Returns the number of days in the year.
  - **Parameters**
    * `settlement` – a `GregorianCalendar` date which specifies the settlement date
    * `maturity` – a `GregorianCalendar` date which specifies the maturity date
  - **Returns** – an `int` which specifies the number of days in the year

## *class* **Bond**

Collection of bond functions.

*rate* is an annualized rate of return based on the par value of the bills.

*yield* is an annualized rate based on the purchase price and reflects the actual yield to maturity.

*coupons* are interest payments on a bond.

*redemption* is the amount a bond pays at maturity.

*frequency* is the number of times a year that a bond makes interest payments.

*basis* is the method used to calculate dates. For example, sometimes computations are done assuming 360 days in a year.

*issue* is the day a bond is first sold.

*settlement* is the day a purchaser aquires a bond.

*maturity* is the day a bond's principal is repaid.

Discount bonds, also called *zero-coupon* bonds, do not pay interest during the life of the security, instead they sell at a discount to their value at maturity. The discount bond methods all have *settlement*, *maturity*, *basis* and *redemption* as arguments. In the following list these common arguments are omitted.

- price = pricedisc(rate)
- price = priceyield(yield)
- price = pricemat(issue, rate, yield)
- rate = disc(price)
- yield = yielddisc(price)

A related method is accrintm, which returns the interest that has accumulated on the discount bond.

US Treasury bills are a special case of discount bonds. The *basis* is fixed for treasury bills and the redemption value is assumed to be $100. So these functions have only *settlement* and *maturity* as common arguments.

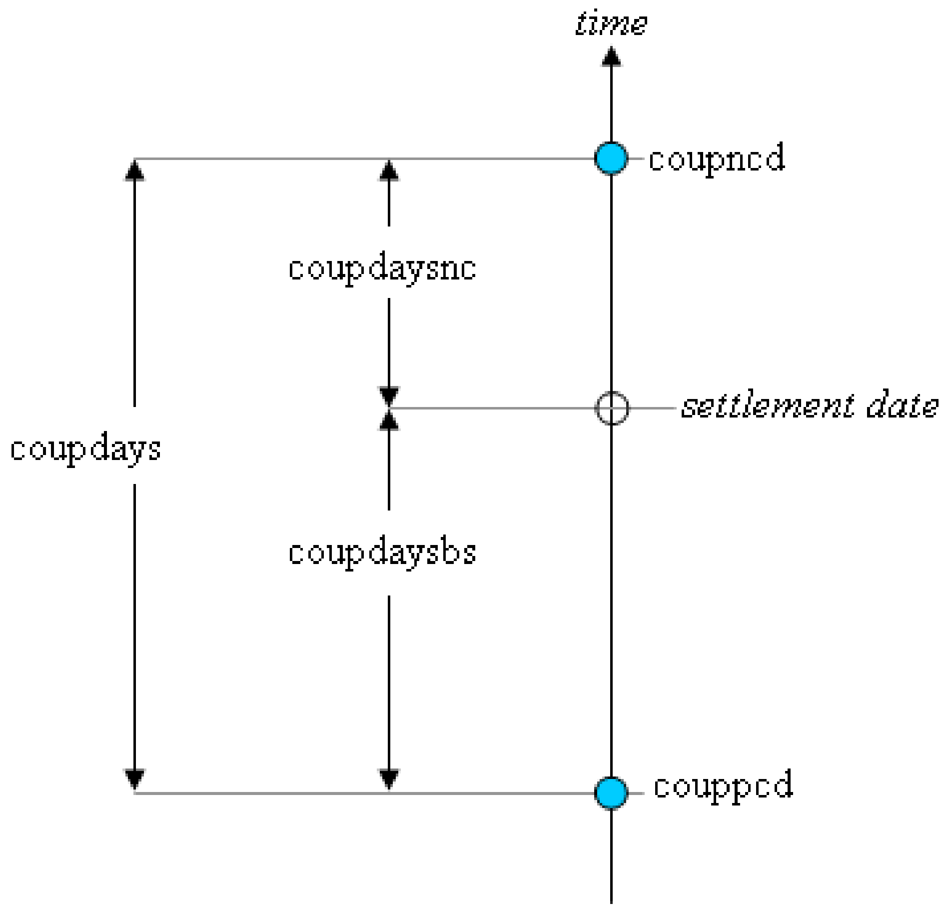
- price = tbillprice(rate)
- yield = tbillyield(price)
- yield = tbilleq(rate)

Most bonds pay interest periodically. The interest paying bond methods all have *settlement*, *maturity*, *basis* and *frequency* as arguments. Again supressing the common arguments,

- price = price(rate, yield, redemption)
- yield = yield(rate, price, redemption)
- redemption = received(price, rate)

A related method is accrint, which returns the interest that has accumulated at settlement from the previous coupon date.

In this diagram, the settlement date is shown as a hollow circle and the adjacent coupon dates are shown as filled circles.




- coupppcd is the coupon date immediately prior to the settlement date.
- coupncd is the coupon date immediately after the settlement date.

- coupdaybs is the number of days from the immediately prior coupon date to the settlement date.
- coupdaysnc is the number of days from the settlement date to the next coupon date.
- coupdays is the number of days between these two coupon dates.

A related method is coupnum, which returns the number of coupons payable between settlement and maturity.

Another related method is yearfrac, which returns the fraction of the year between two days.

Duration is used to measure the sensitivity of a bond to changes in interest rates. Convexity is a measure of the sensitivity of duration.

- duration
- modified duration
- convexity

### Declaration

public class com.imsl.finance.Bond
**extends** java.lang.Object

### Fields

- public static final `int` **ANNUAL**
  - Coupon payments are made annually.
- public static final `int` **SEMIANNUAL**
  - Coupon payments are made semiannually.
- public static final `int` **QUARTERLY**
  - Coupon payments are made quarterly.

### Constructor

- *Bond*
  `public` **Bond**`( )`

## Methods

---

- *accrint*
  `public static double` **accrint**`( java.util.GregorianCalendar` **issue**`, java.util.GregorianCalendar` **firstCoupon**`, java.util.GregorianCalendar` **settlement**`, double` **rate**`, double` **par**`, int` **frequency**`, DayCountBasis` **basis** `)`

  - **Description**
    Returns the interest which has accrued on a security that pays interest periodically. In the equation below, $A_i$ represents the number of days which have accrued for the $i$th quasi-coupon period within the odd period. (The quasi-coupon periods are periods obtained by extending the series of equal payment periods to before or after the actual payment periods.) $NC$ represents the number of quasi-coupon periods within the odd period, rounded to the next highest integer. (The odd period is a period between payments that differs from the usual equally spaced periods at which payments are made.) $NL_i$ represents the length of the normal ith quasi-coupon period within the odd period. $NL_i$ is expressed in days. Function `accrint` can be found by solving the following:

    $$par\left(\frac{rate}{frequency}\sum_{i=1}^{NC}\frac{A_i}{NL_i}\right)$$

  - **Parameters**
    * `issue` – a `GregorianCalendar` issue date of the security
    * `firstCoupon` – a `GregorianCalendar` date of the security's first interest date
    * `settlement` – a `GregorianCalendar` settlement date of the security
    * `rate` – a `double` which specifies the security's annual coupon rate
    * `par` – a `double` which specifies the security's par value
    * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
    * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
  - **Returns** – a `double` which specifies the accrued interest

---

- *accrintm*
  `public static double` **accrintm**`( java.util.GregorianCalendar` **issue**`, java.util.GregorianCalendar` **maturity**`, double` **rate**`, double` **par**`, DayCountBasis` **basis** `)`

  - **Description**

Returns the interest which has accrued on a security that pays interest at maturity.

$$= par \times rate \times \frac{A}{D}$$

In the above equation, $A$ represents the number of days starting at issue date to maturity date and $D$ represents the annual basis.

– **Parameters**
  * `issue` – a `GregorianCalendar` issue date of the security
  * `maturity` – a `GregorianCalendar` date of the security's maturity
  * `rate` – a `double` which specifies the security's annual coupon rate
  * `par` – a `double` which specifies the security's par value
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. see `DayCountBasis`

– **Returns** – a `double` which specifies the accrued interest

---

• *amordegrc*
`public static double` **amordegrc**`( double` **cost**`, java.util.GregorianCalendar` **issue**`, java.util.GregorianCalendar` **firstPeriod**`, double` **salvage**`, int` **period**`, double` **rate**`, DayCountBasis` **basis** `)`

– **Description**
Returns the depreciation for each accounting period. This method is similar to `amorlinc`. However, in this method a depreciation coefficient based on the asset life is applied during the evaluation of the function.

– **Parameters**
  * `cost` – a `double` which specifies the cost of the asset
  * `issue` – a `GregorianCalendar` issue date of the asset
  * `firstPeriod` – a `GregorianCalendar` date of the end of the first period
  * `salvage` – a `double` which specifies the asset's salvage value at the end of the life of the asset
  * `period` – an `int` which specifies the period
  * `rate` – a `double` which specifies the rate of depreciation
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. see `DayCountBasis`.

– **Returns** – a `double` which specifies the depreciation

---

• *amorlinc*
`public static double` **amorlinc**`( double` **cost**`, java.util.GregorianCalendar` **issue**`, java.util.GregorianCalendar` **firstPeriod**`, double` **salvage**`, int` **period**`, double` **rate**`, DayCountBasis` **basis** `)`

– **Description**
Returns the depreciation for each accounting period. This method is similar to `amordegrc`, except that `amordegrc` has a depreciation coefficient that is applied during the evaluation that is based on the asset life.

– **Parameters**

* `cost` – a `double` which specifes the cost of the asset
* `issue` – a `GregorianCalendar` issue date of the asset
* `firstPeriod` – a `GregorianCalendar` date of the end of the first period
* `salvage` – a `double` which specifies the asset's salvage value at the end of the life of the asset
* `period` – an `int` which specifies the period
* `rate` – a `double` which specifies the rate of depreciation
* `basis` – a `DayCountBasis` object which contains the type of day count basis to use. see `DayCountBasis`.

– **Returns** – a `double` which specifies the depreciation

---

- *convexity*
`public static double` **`convexity`**`( java.util.GregorianCalendar` **`settlement`**`, java.util.GregorianCalendar` **`maturity`**`, double` **`coupon`**`, double` **`yield`**`, int` **`frequency`**`, DayCountBasis` **`basis`** `)`

– **Description**
Returns the convexity for a security. Convexity is the sensitivity of the duration of a security to changes in yield. It is computed using the following:

$$\frac{\frac{1}{(q \times frequency)^2}\left\{\sum_{t=1}^{n} t\,(t+1)\left(\frac{coupon}{frequency}\right)q^{-t} + n\,(n+1)\,q^{-n}\right\}}{\left(\sum_{t=1}^{n}\left(\frac{coupon}{frequency}\right)q^{-t} + q^{-n}\right)}$$

where n is calculated from `coupnum`, and $q = 1 + \frac{yield}{frequency}$.

– **Parameters**

* `settlement` – a `GregorianCalendar` settlement date of the security
* `maturity` – a `GregorianCalendar` maturity date of the security
* `coupon` – a `double` which specifies the security's annual coupon rate
* `yield` – a `double` which specifires the security's annual yield
* `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
* `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.

– **Returns** – a `double` which specifies the convexity for a security

---

- *coupdaybs*
  `public static int` **coupdaybs**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, int` **frequency**`, DayCountBasis` **basis** `)`
  - **Description**
    Returns the number of days starting with the beginning of the coupon period and ending with the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.
  - **Parameters**
    * `settlement` – a `GregorianCalendar` settlement date of the security
    * `maturity` – a `GregorianCalendar` maturity date of the security
    * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
    * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
  - **Returns** – an `int` which specifies the number of days from the beginning of the coupon period to the settlement date

---

- *coupdays*
  `public static double` **coupdays**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, int` **frequency**`, DayCountBasis` **basis** `)`
  - **Description**
    Returns the number of days in the coupon period containing the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.
  - **Parameters**
    * `settlement` – a `GregorianCalendar` settlement date of the security
    * `maturity` – a `GregorianCalendar` maturity date of the security
    * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
    * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
  - **Returns** – an `int` which specifies the number of days in the coupon period that contains the settlement date

---

- *coupdaysnc*
  `public static int` **coupdaysnc**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, int` **frequency**`, DayCountBasis` **basis** `)`

- **Description**
  Returns the number of days starting with the settlement date and ending with the next coupon date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.
- **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
- **Returns** – an `int` which specifies the number of days from the settlement date to the next coupon date

---

- *coupncd*

`public static java.util.GregorianCalendar` **coupncd**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, int` **frequency**`, DayCountBasis` **basis** `)`

- **Description**
  Returns the first coupon date which follows the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.
- **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`
- **Returns** – an `int` which specifies the next coupon date after the settlement date

---

- *coupnum*

`public static int` **coupnum**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, int` **frequency**`, DayCountBasis` **basis** `)`

- **Description**
  Returns the number of coupons payable between the settlement date and the maturity date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

– **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.

– **Returns** – an `int` which specifies the number of coupons payable between the settlement date and maturity date

---

- *couppcd*

`public static java.util.GregorianCalendar` **couppcd**`(` `java.util.GregorianCalendar` **settlement**, `java.util.GregorianCalendar` **maturity**, `int` **frequency**, `DayCountBasis` **basis** `)`

– **Description**
Returns the coupon date which immediately precedes the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

– **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`

– **Returns** – an `int` which specifies the previous coupon date before the settlement date

---

- *disc*

`public static double` **disc**`(` `java.util.GregorianCalendar` **settlement**, `java.util.GregorianCalendar` **maturity**, `double` **price**, `double` **redemption**, `DayCountBasis` **basis** `)`

– **Description**
Returns the implied interest rate of a discount bond. The discount rate is the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments. It is computed using the following:

$$\frac{redemption - price}{price} \times \frac{B}{DSM}$$

In the equation above, $B$ represents the number of days in a year based on the annual basis and $DSM$ represents the number of days starting with the settlement date and ending with the maturity date.

– **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `price` – a `double` which specifies the security's price per $100 face value
  * `redemption` – a `double` which specifies the security's redemption value per $100 face value
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.

– **Returns** – a `double` which specifies the discount rate for a security

---

• *duration*

`public static double` **duration**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **coupon**`, double` **yield**`, int` **frequency**`, DayCountBasis` **basis** `)`

– **Description**

Returns the Macauley's duration of a security where the security has periodic interest payments. The Macauley's duration is the weighted-average time to the payments, where the weights are the present value of the payments. It is computed using the following:

$$\left(\frac{\frac{\frac{DSC}{E}100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}}+\sum_{k=1}^{N}\left(\left(\frac{100\times coupon}{freq\times\left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right)\left(k-1+\frac{DSC}{E}\right)\right)}{\frac{100}{\left(1+\frac{yield}{freq}\right)^{N-1+\frac{DSC}{E}}}+\sum_{k=1}^{N}\left(\frac{100\times coupon}{freq\times\left(1+\frac{yield}{freq}\right)^{k-1+\frac{DSC}{E}}}\right)}\right)\frac{1}{freq}$$

In the equation above, $DSC$ represents the number of days starting with the settlement date and ending with the next coupon date. $E$ represents the number of days within the coupon period. $N$ represents the number of coupons payable from the settlement date to the maturity date. $freq$ represents the frequency of the coupon payments annually.

– **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `coupon` – a `double` which specifies the security's annual coupon rate
  * `yield` – a `double` which specifies the security's annual yield
  * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly

* basis – a DayCountBasis object which contains the type of day count basis to use. See DayCountBasis.

- **Returns** – a double which specifies the annual duration of a security with periodic interest payments

---

- *intrate*
  public static double **intrate**( java.util.GregorianCalendar **settlement**, java.util.GregorianCalendar **maturity**, double **investment**, double **redemption**, DayCountBasis **basis** )

  - **Description**
    Returns the interest rate of a fully invested security. It is computed using the following:

$$\frac{redemption - investment}{investment} \times \frac{B}{DSM}$$

    In the equation above, $B$ represents the number of days in a year based on the annual basis, and $DSM$ represents the number of days in the period starting with the settlement date and ending with the maturity date.

  - **Parameters**
    * settlement – a GregorianCalendar settlement date of the security
    * maturity – a GregorianCalendar maturity date of the security
    * investment – a double which specifies the amount invested
    * redemption – a double which specifies the amount to be received at maturity
    * basis – a DayCountBasis object which contains the type of day count basis to use. See DayCountBasis.
  - **Returns** – a double which specifies the interest rate for a fully invested security

---

- *mduration*
  public static double **mduration**( java.util.GregorianCalendar **settlement**, java.util.GregorianCalendar **maturity**, double **coupon**, double **yield**, int **frequency**, DayCountBasis **basis** )

  - **Description**
    Returns the modified Macauley duration for a security with an assumed par value of $100. It is computed using the following:

$$\frac{duration}{1 + \frac{yield}{frequency}}$$

    where *duration* is calculated from mduration.

  - **Parameters**
    * settlement – a GregorianCalendar settlement date of the security

* `maturity` – a `GregorianCalendar` maturity date of the security
* `coupon` – a `double` which specifies the security's annual coupon rate
* `yield` – a `double` which specifies the security's annual yield
* `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
* `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.

– **Returns** – a `double` which specifies the modified Macauley duration for a security with an assumed par value of $100

---

- *price*

`public static double` **price**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **rate**`, double` **yield**`, double` **redemption**`, int` **frequency**`, DayCountBasis` **basis** `)`

– **Description**

Returns the price, per $100 face value, of a security that pays periodic interest. It is computed using the following:

$$\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{\left(N-1+\frac{DSC}{E}\right)}}+\sum_{k=1}^{N}\frac{100\times\frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{\left(k-1+\frac{DSC}{E}\right)}}-\left(100\times\frac{rate}{frequency}\times\frac{A}{E}\right)$$

In the above equation, $DSC$ represents the number of days in the period starting with the settlement date and ending with the next coupon date. $E$ represents the number of days within the coupon period. $N$ represents the number of coupons payable in the timeframe from the settlement date to the redemption date. $A$ represents the number of days in the timeframe starting with the beginning of coupon period and ending with the settlement date.

– **Parameters**

* `settlement` – a `GregorianCalendar` settlement date of the security
* `maturity` – a `GregorianCalendar` maturity date of the security
* `rate` – a `double` which specifies the security's annual coupon rate
* `yield` – a `double` which specifies the security's annual yield
* `redemption` – a `double` which specifies the security's redemption value per $100 face value
* `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
* `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.

– **Returns** – a `double` which specifies the price per $100 face value of a security that pays periodic interest

- *pricedisc*

  `public static double` **pricedisc**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **rate**`, double` **redemption**`, DayCountBasis` **basis** `)`

  - **Description**

    Returns the price of a discount bond given the discount rate. It is computed using the following:

    $$redemption - rate \times redemption \times \frac{DSM}{B}$$

    In the equation above, $DSM$ represents the number of days starting at the settlement date and ending with the maturity date. $B$ represents the number of days in a year based on the annual basis.

  - **Parameters**

    * `settlement` – a `GregorianCalendar` settlement date of the security
    * `maturity` – a `GregorianCalendar` maturity date of the security
    * `rate` – a `double` which specifies the security's discount rate
    * `redemption` – a `double` which specifies the security's redemption value per $100 face value
    * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.

  - **Returns** – a `double` which specifies the price per $100 face value of a discounted security

- *pricemat*

  `public static double` **pricemat**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, java.util.GregorianCalendar` **issue**`, double` **rate**`, double` **yield**`, DayCountBasis` **basis** `)`

  - **Description**

    Returns the price, per $100 face value, of a discount bond. It is computed using the following:

    $$\frac{100 + \left(\frac{DIM}{B} \times rate \times 100\right)}{1 + \left(\frac{DSM}{B} \times yield\right)} - \frac{A}{B} \times rate \times 100$$

    In the equation above, $B$ represents the number of days in a year based on the annual basis. $DSM$ represents the number of days in the period starting with the settlement date and ending with the maturity date. $DIM$ represents the number of days in the period starting with the issue date and ending with the maturity date. $A$ represents the number of days in the period starting with the issue date and ending with the settlement date.

– **Parameters**

  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `issue` – a `GregorianCalendar` issue date of the security
  * `rate` – a `double` which specifies the security's interest rate at issue date
  * `yield` – a `double` which specifies the security's annual yield
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. see `DayCountBasis`

– **Returns** – a `double` which specifies the price per $100 face value of a security that pays interest at maturity

---

• *priceyield*
`public static double` **priceyield**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **yield**`, double` **redemption**`, DayCountBasis` **basis** `)`

– **Description**
Returns the price of a discount bond given the yield. It is computed using the following:

$$\frac{redemption}{1+\left(\frac{DSM}{B}\right)yield}$$

In the equation above, $DSM$ represents the number of days starting at the settlement date and ending with the maturity date. $B$ represents the number of days in a year based on the annual basis.

– **Parameters**

  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `yield` – a `double` which specifies the security's yield
  * `redemption` – a `double` which specifies the security's redemption value per $100 face value
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`

– **Returns** – a `double` which specifies the price per $100 face value of a discounted security

---

• *received*
`public static double` **received**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **investment**`, double` **rate**`, DayCountBasis` **basis** `)`

– **Description**
Returns the amount one receives when a fully invested security reaches the maturity date. It is computed using the following:

$$\frac{investment}{1-\left(rate \times \frac{DIM}{B}\right)}$$

In the equation above, $B$ represents the number of days in a year based on the annual basis, and $DIM$ represents the number of days in the period starting with the issue date and ending with the maturity date.

- **Parameters**
  - `settlement` – a `GregorianCalendar` settlement date of the security
  - `maturity` – a `GregorianCalendar` maturity date of the security
  - `investment` – a `double` which specifies the amount invested in the security
  - `rate` – a `double` which specifies the security's rate at issue date
  - `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
- **Returns** – a `double` which specifies the amount received at maturity for a fully invested security

---

- *tbilleq*

`public static double` **`tbilleq(`** `java.util.GregorianCalendar` **`settlement,`** `java.util.GregorianCalendar` **`maturity,`** `double` **`rate`** `)`

- **Description**

  Returns the bond-equivalent yield of a Treasury bill. It is computed using the following:
  If $DSM <= 182$

$$\frac{365 \times rate}{360 - rate \times DSM}$$

otherwise,

$$\frac{-\frac{DSM}{365} + \sqrt{\left(\frac{DSM}{365}\right)^2 - \left(2 \times \frac{DSM}{365} - 1\right) \times \frac{rate \times DSM}{rate \times DSM - 360}}}{\frac{DSM}{365} - 0.5}$$

In the above equation, $DSM$ represents the number of days starting at settlement date to maturity date.

- **Parameters**
  - `settlement` – a `GregorianCalendar` settlement date of the Treasury bill
  - `maturity` – a `GregorianCalendar` maturity date of the Treasury bill. The maturity cannot be more than a year after the settlement.
  - `rate` – a `double` which specifies the Treasury bill's discount rate at issue date. The discount rate is an annualized rate of return based on the par value of the bills. The discount rate is calculated on a 360-day basis (twelve 30-day months).

– **Returns** – a `double` which specifies the bond-equivalent yield for the Treasury bill. This is an annualized rate based on the purchase price of the bills and reflects the actual yield to maturity.

---

- *tbillprice*

`public static double` **tbillprice**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **rate** `)`

  – **Description**

  Returns the price, per $100 face value, of a Treasury bill. It is computed using the following:

$$100\left(1-\frac{rate\times DSM}{360}\right)$$

  In the equation above, $DSM$ represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

  – **Parameters**

  * `settlement` – a `GregorianCalendar` settlement date of the Treasury bill
  * `maturity` – a `GregorianCalendar` maturity date of the Treasury bill. The maturity cannot be more than a year after the settlement
  * `rate` – a `double` which specifies the Treasury bill's discount rate at issue date. The discount rate is an annualized rate of return based on the par value of the bills. The discount rate is calculated on a 360-day basis (twelve 30-day months).

  – **Returns** – a `double` which specifies the price per $100 face value for the Treasury bill

---

- *tbillyield*

`public static double` **tbillyield**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **price** `)`

  – **Description**

  Returns the yield of a Treasury bill. It is computed using the following:

$$\frac{100-price}{price}\times\frac{360}{DSM}$$

  In the equation above, $DSM$ represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

  – **Parameters**

- * `settlement` – a `GregorianCalendar` settlement date of the Treasury bill
  * `maturity` – a `GregorianCalendar` maturity date of the Treasury bill. The maturity cannot be more than a year after the settlement.
  * `price` – a `double` which specifies the Treasury bill's price per $100 face value
- **Returns** – a `double` which specifies the yield for the Treasury bill. This is an annualized rate based on the purchase price of the bills and reflects the actual yield to maturity.

---

- *yearfrac*

`public static double` **`yearfrac`**`( java.util.GregorianCalendar` **`start`**`, java.util.GregorianCalendar` **`end`**`, DayCountBasis` **`basis`** `)`

- **Description**

Returns the fraction of a year represented by the number of whole days between two dates. It is computed using the following:

$$A/D$$

where $A$ equals the number of days from `start` to `end`, $D$ equals annual basis.

- **Parameters**
  * `start` – a `GregorianCalendar` start date of the security
  * `end` – a `GregorianCalendar` end date of the security
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
- **Returns** – a `double` which specifies the annual yield of a security that pays interest at maturity

---

- *yield*

`public static double` **`yield`**`( java.util.GregorianCalendar` **`settlement`**`, java.util.GregorianCalendar` **`maturity`**`, double` **`rate`**`, double` **`price`**`, double` **`redemption`**`, int` **`frequency`**`, DayCountBasis` **`basis`** `)`

- **Description**

Returns the yield of a security that pays periodic interest. If there is one coupon period use the following:

$$\frac{\left(\frac{redemption}{100}+\frac{rate}{frequency}\right)-\left[\frac{price}{100}+\left(\frac{A}{E}\times\frac{rate}{frequency}\right)\right]}{\frac{price}{100}+\left(\frac{A}{E}\times\frac{rate}{frequency}\right)}\times\frac{frequency\times E}{DSR}$$

In the equation above, $DSR$ represents the number of days in the period starting with the settlement date and ending with the redemption date. $E$ represents the number of days within the coupon period. $A$ represents the

number of days in the period starting with the beginning of coupon period and ending with the settlement date.
If there is more than one coupon period use the following:

$$price - \frac{redemption}{\left(\frac{1+yield}{frequency}\right)^{\frac{N-1+DSC}{E}}} - \left(\sum_{k=1}^{N} \frac{100 \times \frac{rate}{frequency}}{\left(\frac{1+yield}{frequency}\right)^{\frac{k-1+DSC}{E}}}\right) + 100 \times \frac{rate}{frequency} \times \frac{A}{E}$$

In the equation above, $DSC$ represents the number of days in the period from the settlement to the next coupon date. $E$ represents the number of days within the coupon period.$N$ represents the number of coupons payable in the period starting with the settlement date and ending with the redemption date. $A$ represents the number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

– **Parameters**
  * `settlement` – a `GregorianCalendar` settlement date of the security
  * `maturity` – a `GregorianCalendar` maturity date of the security
  * `rate` – a `double` which specifies the security's annual coupon rate
  * `price` – a `double` which specifies the security's price per $100 face value
  * `redemption` – a `double` which specifies the security's redemption value per $100 face value
  * `frequency` – an `int` which specifies the number of coupon payments per year; `ANNUAL` for annual, `SEMIANNUAL` for semiannual and `QUARTERLY` for quarterly
  * `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
– **Returns** – a `double` which specifies the yield of a security that pays periodic interest

---

• *yielddisc*
`public static double` **yielddisc**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, double` **price**`, double` **redemption**`, DayCountBasis` **basis** `)`

– **Description**
Returns the annual yield of a discount bond. It is computed using the following:

$$\frac{redemption - price}{price} \times \frac{B}{DSM}$$

In the equation above, $B$ represents the number of days in a year based on the annual basis, and $DSM$ represents the number of days starting with the settlement date and ending with the maturity date.

- **Parameters**
  - `settlement` – a `GregorianCalendar` settlement date of the security
  - `maturity` – a `GregorianCalendar` maturity date of the security
  - `price` – a `double` which specifies the security's price per $100 face value
  - `redemption` – a `double` which specifies the security's redemption value per $100 face value
  - `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
- **Returns** – a `double` which specifies the annual yield for a discounted security

---

- *yieldmat*

  `public static double` **yieldmat**`( java.util.GregorianCalendar` **settlement**`, java.util.GregorianCalendar` **maturity**`, java.util.GregorianCalendar` **issue**`, double` **rate**`, double` **price**`, DayCountBasis` **basis** `)`

  - **Description**

    Returns the annual yield of a security that pays interest at maturity. It is computed using the following:

$$\frac{\left[1+\left(\frac{DIM}{B}\times rate\right)\right]-\left[\frac{price}{100}+\left(\frac{A}{B}\times rate\right)\right]}{\frac{price}{100}+\left(\frac{A}{B}\times rate\right)}\times\frac{B}{DSM}$$

    In the equation above, $DIM$ represents the number of days in the period starting with the issue date and ending with the maturity date. $DSM$ represents the number of days in the period starting with the settlement date and ending with the maturity date. $A$ represents the number of days in the period starting with the issue date and ending with the settlement date. $B$ represents the number of days in a year based on the annual basis.
  - **Parameters**
    - `settlement` – a `GregorianCalendar` settlement date of the security
    - `maturity` – a `GregorianCalendar` maturity date of the security
    - `issue` – a `GregorianCalendar` issue date of the security
    - `rate` – a `double` which specifies the security's interest rate at date of issue
    - `price` – a `double` the security's price per $100 face value
    - `basis` – a `DayCountBasis` object which contains the type of day count basis to use. See `DayCountBasis`.
  - **Returns** – a `double` which specifies the annual yield of a security that pays interest at maturity

### Example: Accrued Interest - Periodic Payments

In this example, the accrued interest is calculated for a bond which pays interest semiannually. The day count basis used is 30/360.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class accrintEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar issue = parse("10/1/91");
        GregorianCalendar firstCoupon = parse("3/31/92");
        GregorianCalendar settlement = parse("11/3/91");
        double rate = .06;
        double par = 1000.;
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double accrint = Bond.accrint(issue, firstCoupon, settlement, rate,
        par, freq, dcb);
        System.out.println("The accrued interest is " +accrint);
    }
}
```

### Output

```
The accrued interest is 5.333333333333334
```

### Example: Accrued Interest - Payment at Maturity

In this example, the accrued interest is calculated for a bond which pays at maturity. The day count basis used is 30/360.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;
```

```
public class accrintmEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar issue = parse("10/1/91");
        GregorianCalendar settlement = parse("11/3/91");
        double rate = .06;
        double par = 1000.;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double accrintm = Bond.accrintm(issue, settlement, rate, par, dcb);
        System.out.println("The accrued interest is " +accrintm);
    }
}
```

## Output

```
The accrued interest is 5.333333333333334
```

## Example: Depreciation - French Accounting System

In this example, the depreciation for the second accounting period is calculated for an asset.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class amordegrcEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
```

```
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        double cost = 2400.;
        GregorianCalendar issue = parse("11/1/92");
        GregorianCalendar firstPeriod = parse("11/30/93");
        double salvage = 300.;
        int period = 2;
        double rate = .15;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double amordegrc = Bond.amordegrc(cost, issue, firstPeriod,
        salvage, period, rate, dcb);
        System.out.println("The depreciation for the second accounting " +
        "period is " +amordegrc);
    }
}
```

### Output

```
The depreciation for the second accounting period is 334.0
```

### Example: Depreciation - French Accounting System

In this example, the depreciation for the second accounting period is calculated for an asset.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class amorlincEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }
```

```
    public static void main(String args[]) throws ParseException {
        double cost = 2400.;
        GregorianCalendar issue = parse("11/1/92");
        GregorianCalendar firstPeriod = parse("11/30/93");
        double salvage = 300.;
        int period = 2;
        double rate = .15;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double amorlinc = Bond.amorlinc(cost, issue, firstPeriod,
        salvage, period, rate, dcb);
        System.out.println("The depreciation for the second accounting " +
        "period is " +amorlinc);
    }
}
```

### Output

```
The depreciation for the second accounting period is 360.0
```

### Example: Convexity for a Security

The convexity of a 10 year bond which pays interest semiannually is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class convexityEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/90");
        GregorianCalendar maturity = parse("7/1/00");
```

```
        double coupon = .075;
        double yield = .09;
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double convexity = Bond.convexity(settlement, maturity, coupon,
        yield, freq, dcb);
        System.out.println("The convexity of the bond with semiannual " +
        "interest payments is " + convexity);
    }
}
```

### Output

```
The convexity of the bond with semiannual interest payments is 59.404991291585645
```

### Example: Days - Beginning of Period to Settlement Date

In this example, the settlement date is 11/11/86. The number of days from the beginning of the coupon period to the settlement date is returned.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class coupdaybsEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("11/11/86");
        GregorianCalendar maturity = parse("3/1/99");
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        int coupdaybs = Bond.coupdaybs(settlement, maturity, freq, dcb);
        System.out.println("The number of days from the beginning of the" +
```

```
        "\ncoupon period to the settlement date is " + coupdaybs);
    }
}
```

### Output

```
The number of days from the beginning of the
coupon period to the settlement date is 71
```

### Example: Days in the Settlement Date Period

In this example, the settlement date is 11/11/86. The number of days in the coupon period containing this date is returned.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class coupdaysEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("11/11/86");
        GregorianCalendar maturity = parse("3/1/99");
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double coupdays = Bond.coupdays(settlement, maturity, freq, dcb);
        System.out.println("The number of days in the coupon period that " +
        "contains the settlement date is " + coupdays);
    }
}
```

### Output

```
The number of days in the coupon period that contains the settlement date is 182.5
```

### Example: Days - Settlement Date to Next Coupon Date

In this example, the settlement date is 11/11/86. The number of days from this date to the next coupon date is returned.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class coupdaysncEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("11/11/86");
        GregorianCalendar maturity = parse("3/1/99");
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        int coupdaysnc = Bond.coupdaysnc(settlement, maturity, freq, dcb);
        System.out.println("The number of days from the settlement date " +
        "to the next coupon date is " +coupdaysnc);
    }
}
```

### Output

```
The number of days from the settlement date to the next coupon date is 110
```

### Example: Next Coupon Date After the Settlement Date

In this example, the settlement date is 11/11/86. The previous coupon date before this date is returned.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class coupncdEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("11/11/86");
        GregorianCalendar maturity = parse("3/1/99");
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        GregorianCalendar coupncd = Bond.coupncd(settlement, maturity,
        freq, dcb);
        System.out.println("The next coupon date after the settlement date is "
        + dateFormat.format(coupncd.getTime()));
    }
}
```

### Output

```
The next coupon date after the settlement date is 3/1/87
```

### Example: Number of Payable Coupons

In this example, the settlement date is 11/11/86. The number of payable coupons between this date and the maturity date is returned.

```
import com.imsl.finance.*;
import java.text.*;
```

```
import java.util.*;

public class coupnumEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("11/11/86");
        GregorianCalendar maturity = parse("3/1/99");
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        int coupnum = Bond.coupnum(settlement, maturity, freq, dcb);
        System.out.println("The number of coupons payable between the " +
        "\nsettlement date and the maturity date is " + coupnum);
    }
}
```

### Output

```
The number of coupons payable between the
settlement date and the maturity date is 25
```

### Example: Previous Coupon Date Before the Settlement Date

In this example, the settlement date is 11/11/86. The previous coupon date before this date is returned.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class couppcdEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);
```

```
    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("11/11/86");
        GregorianCalendar maturity = parse("3/1/99");
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        GregorianCalendar couppcd = Bond.couppcd(settlement, maturity,
        freq, dcb);
        System.out.println("The previous coupon date before the settlement " +
        "date is " + dateFormat.format(couppcd.getTime()));
    }
}
```

### Output

```
The previous coupon date before the settlement date is 9/1/86
```

### Example: Discount Rate for a Security

In this example, the discount rate for a security is returned.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class discEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
```

```
        GregorianCalendar settlement = parse("2/15/92");
        GregorianCalendar maturity = parse("6/10/92");
        double price = 97.975;
        double redemption = 100.;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double disc = Bond.disc(settlement, maturity, price, redemption, dcb);
        System.out.println("The discount rate for the security is " +disc);
    }
}
```

### Output

```
The discount rate for the security is 0.06371767241379328
```

### Example: Duration of a Security with Periodic Payments

The annual duration of a 10 year bond which pays interest semiannually is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class durationEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double coupon = .075;
        double yield = .09;
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double duration = Bond.duration(settlement, maturity, coupon,
```

```
        yield, freq, dcb);
        System.out.println("The annual duration of the bond with" +
        "\nsemiannual interest payments is " + duration);
    }
}
```

## Output

```
The annual duration of the bond with
semiannual interest payments is 7.041953377972151
```

## Example: Interest Rate of a Fully Invested Security

The discount rate of a 10 year bond is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class intrateEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double investment = 7000.;
        double redemption = 10000.;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double intrate = Bond.intrate(settlement, maturity, investment,
        redemption, dcb);
        System.out.println("The interest rate of the bond is " +intrate);
    }
}
```

### Output

```
The interest rate of the bond is 0.042833672351744644
```

### Example: Modified Macauley Duration of a Security with Periodic Payments

The modified Macauley duration of a 10 year bond which pays interest semiannually is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class mdurationEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double coupon = .075;
        double yield = .09;
        int freq = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double mduration = Bond.mduration(settlement, maturity,
        coupon, yield, freq, dcb);
        System.out.println("The modified Macauley duration of the bond" +
        "\nwith semiannual interest payments is " + mduration);
    }
}
```

### Output

```
The modified Macauley duration of the bond
with semiannual interest payments is 6.738711366480527
```

### Example: Price of a Security

The price per $100 face value of a 10 year bond which pays interest semiannually is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class priceEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double rate = .06;
        double yield = .07;
        double redemption = 105.;
        int frequency = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double price = Bond.price(settlement, maturity, rate, yield,
        redemption, frequency, dcb);
        System.out.println("The price of the bond is " +price);
    }
}
```

### Output

```
The price of the bond is 95.40662777118231
```

### Example: Price of a Discounted Security

The price per $100 face value of a discounted 1 year bond is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class pricediscEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/86");
        double rate = .05;
        double redemption = 100.;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double pricedisc = Bond.pricedisc(settlement, maturity,
        rate, redemption, dcb);
        System.out.println("The price of the discounted bond is " +pricedisc);
    }
}
```

### Output

```
The price of the discounted bond is 95.0
```

### Example: Price of a Security that Pays at Maturity

The price per $100 face value of 1 year bond that pays interest at maturity is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class pricematEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("8/1/85");
        GregorianCalendar maturity = parse("7/1/86");
        GregorianCalendar issue = parse("7/1/85");
        double rate = .05;
        double yield = .05;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double pricemat = Bond.pricemat(settlement, maturity, issue,
        rate, yield, dcb);
        System.out.println("The price of the bond is " +pricemat);
    }
}
```

### Output

```
The price of the bond is 99.98173970783533
```

The price of a discounted 1 year bond is returned in this example.

```
package com.imsl.example.finance;
import com.imsl.finance.*;
import java.text.*;
import java.util.*;
```

```
public class priceyieldEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s)
    throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double yield = 0.010055244588347783;
        double redemption = 105.;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double priceyield = Bond.priceyield(settlement, maturity,
        yield, redemption, dcb);
        System.out.println("The price of the discounted bond is "
        + priceyield);
    }
}
```

```
The price of the discounted bond is 95.40663
```

### Example: Amount Received at Maturity for a Fully Invested Security

The amount to be received at maturity for a 10 year bond is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class receivedEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
```

```
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double investment = 7000.;
        double discount = .06;
        DayCountBasis dcb = DayCountBasis.BasisActual365;
        double received = Bond.received(settlement, maturity,
        investment, discount, dcb);
        System.out.println("The amount received at maturity for the bond is " +
        NumberFormat.getCurrencyInstance().format(received));
    }
}
```

### Output

```
The amount received at maturity for the bond is $17,514.40
```

### Example: Bond-Equivalent Yield

The bond-equivalent yield for a 1 year Treasury bill is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class tbilleqEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/86");
```

```
        double discount = .05;
        double tbilleq = Bond.tbilleq(settlement, maturity, discount);
        System.out.println("The bond-equivalent yield for the T-bill is "
        + tbilleq);
    }
}
```

### Output

```
The bond-equivalent yield for the T-bill is 0.05270709977197674
```

### Example: Treasury Bill Price

The price per $100 face value for a 1 year Treasury bill is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class tbillpriceEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/86");
        double discount = .05;
        double tbillprice = Bond.tbillprice(settlement, maturity, discount);
        System.out.println("The price per $100 face value for the T-bill is "
        + tbillprice);
    }
}
```

## Output

```
The price per $100 face value for the T-bill is 94.93055555555556
```

## Example: Treasury Bill Yield

The yield for a 1 year Treasury bill is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class tbillyieldEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/86");
        double price = 94.93;
        double tbillyield = Bond.tbillyield(settlement, maturity, price);
        System.out.println("The yield for the T-bill is " +tbillyield);
    }
}
```

## Output

```
The yield for the T-bill is 0.05267616080486118
```

## Example: Year Fraction

The year fraction of a 30/360 year starting 8/1/85 and ending 7/1/86 is returned in this example.

```
import com.imsl.finance.*;
```

```
import java.text.*;
import java.util.*;

public class yearfracEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar start = parse("8/1/85");
        GregorianCalendar end = parse("7/1/86");
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yearfrac = Bond.yearfrac(start, end, dcb);
        System.out.println("The year fraction of the 30/360 period is " +
        yearfrac);
    }
}
```

### Output

```
The year fraction of the 30/360 period is 0.9166666666666666
```

### Example: Yield on a Security

The yield on a 10 year bond which pays interest semiannually is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class yieldEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
```

```
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double rate = .06;
        double price = 95.40663;
        double redemption = 105.;
        int frequency = Bond.SEMIANNUAL;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yield = Bond.yield(settlement, maturity, rate, price,
        redemption, frequency, dcb);
        System.out.println("The yield of the bond is " + yield);
    }
}
```

### Output

```
The yield of the bond is 0.06999999682842895
```

### Example: Yield on a Discounted Security

The yield on a discounted 10 year bond is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class yielddiscEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
```

```
        GregorianCalendar settlement = parse("7/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        double price = 95.40663;
        double redemption = 105.;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yielddisc = Bond.yielddisc(settlement, maturity, price,
        redemption, dcb);
        System.out.println("The yield on the discounted bond is " + yielddisc);
    }
}
```

### Output

```
The yield on the discounted bond is 0.010055244588347783
```

### Example: Yield on a Security Which Pays at Maturity

The yield on a bond which pays at maturity is returned in this example.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class yieldmatEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    static private GregorianCalendar parse(String s) throws ParseException {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTime(dateFormat.parse(s));
        return cal;
    }

    public static void main(String args[]) throws ParseException {
        GregorianCalendar settlement = parse("8/1/85");
        GregorianCalendar maturity = parse("7/1/95");
        GregorianCalendar issue = parse("7/1/85");
        double rate = .06;
        double price = 95.40663;
        DayCountBasis dcb = DayCountBasis.BasisNASD;
        double yieldmat = Bond.yieldmat(settlement, maturity, issue, rate,
```

```
        price, dcb);
        System.out.println("The yield on a bond which pays at maturity is " +
        yieldmat);
    }
}
```

### Output

```
The yield on a bond which pays at maturity is 0.06739051278091948
```

## *class* DayCountBasis

The Day Count Basis. Rules for computing the number or days between two dates or number of days in a year. For many securities, computations are based on rules other than on the actual calendar.

### Declaration

public class com.imsl.finance.DayCountBasis
**extends** java.lang.Object

### Fields

---

- public static final `BasisPart` **BasisPartNASD**
  - Computations based on the assumption of 30 days per month and 360 days per year.
- public static final `BasisPart` **BasisPart30E360**
  - Computations based on the assumption of 30 days per month and 360 days per year. This computes the number of days between two dates differently than BasisPartNASD for months with other than 30 days.
- public static final `BasisPart` **BasisPart365**
  - Computations based on the assumption of 365 days per year.
- public static final `BasisPart` **BasisPartActual**

- Computations are based on the actual calendar.

- public static final `DayCountBasis` **BasisNASD**
  - Computations based on the assumption of 30 days per month and 360 days per year.

- public static final `DayCountBasis` **BasisActualActual**
  - Computations are based on the actual calendar.

- public static final `DayCountBasis` **BasisActual360**
  - Computations are based on the number of days in a month based on the actual calendar value and the number of days, but assuming 360 days per year.

- public static final `DayCountBasis` **BasisActual365**
  - Computations are based on the number of days in a month based on the actual calendar value and the number of days, but assuming 365 days per year.

- public static final `DayCountBasis` **Basis30e360**
  - Computations based on the assumption of 30 days per month and 360 days per year.

## Constructor

---

- *DayCountBasis*
  `public` **DayCountBasis**`( BasisPart` **monthBasis**`, BasisPart` **yearBasis** `)`
  - **Description**
    Creates a new DayCountBasis.
  - **Parameters**
    * `monthBasis` – is the month basis
    * `yearBasis` – is the year basis

## Methods

---

- *getMonthBasis*
  `public BasisPart` **getMonthBasis**`( )`
  - **Description**
    Returns the (days in month) portion of the Day Count Basis.

– **Returns** – a `BasisPart` object which represents the month Basis for this `DayCountBasis`

---

- *getYearBasis*
  `public BasisPart` **getYearBasis**`( )`
  – **Description**
  Returns the (days in year) portion of the Day Count Basis.
  – **Returns** – a `BasisPart` object which represents the year Basis for this `DayCountBasis`

# *class* Finance

Collection of finance functions.

### Declaration

public class com.imsl.finance.Finance
**extends** java.lang.Object

### Fields

---

- public static final `int` **AT_END_OF_PERIOD**
  – Flag used to indicate that payment is made at the end of each period.
- public static final `int` **AT_BEGINNING_OF_PERIOD**
  – Flag used to indicate that payment is made at the beginning of each period.

### Methods

---

- *cumipmt*
  `public static double` **cumipmt**`( double` **rate**`, int` **nper**`, double` **pv**`, int` **start**`, int` **end**`, int` **when** `)`

- **Description**

  Returns the cumulative interest paid between two periods. It is computed using the following:

$$\sum_{i=start}^{end} interest_i$$

  where $interest_i$ is computed from `ipmt` for the $i$th period.

- **Parameters**
  * `rate` – a `double`, the interest rate
  * `nper` – an `int`, the total number of payment periods
  * `pv` – a `double`, the present value
  * `start` – an `int`, the first period in the caclulation. Periods are numbered starting with one.
  * `end` – an `int`, the last period in the calculation
  * `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD
- **Returns** – a `double`, the cumulative interest paid between the first period and the last period

---

- *cumprinc*

  `public static double` **`cumprinc`**`( double` **`rate`**`, int` **`nper`**`, double` **`pv`**`, int` **`start`**`, int` **`end`**`, int` **`when`** `)`

  - **Description**

    Returns the cumulative principal paid between two periods. It is computed using the following:

$$\sum_{i=start}^{end} principal_i$$

    where $principal_i$ is computed from `ppmt` for the $i$th period.

  - **Parameters**
    * `rate` – a `double`, the interest rate
    * `nper` – an `int`, the total number of payment periods
    * `pv` – a `double`, the present value
    * `start` – an `int`, the first period in the calculation. Periods are numbered starting with one.
    * `end` – an `int`, the last period in the calculation
    * `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD.
  - **Returns** – a `double`, the cumulative principal paid between the first period and the last period

---

- *db*

  **`public static double db( double cost, double salvage, int life, int period, int month )`**

  – **Description**

  Returns the depreciation of an asset using the fixed-declining balance method. Method `db` varies depending on the specified value for the argument period, see table below.

  If period = 1,

  $$\text{cost} \times \text{rate} \times \frac{\text{month}}{12}$$

  If period = life,

  $$(\text{cost} - \text{total depreciation from periods}) \times \text{rate} \times \frac{12 - \text{month}}{12}$$

  If period other than 1 or life,

  $$(\text{cost} - \text{total depreciation from priorperiods}) \times rate$$

  where

  $$rate\ = 1 - \left(\frac{\text{salvage}}{\text{cost}}\right)^{\left(\frac{1}{life}\right)}$$

  NOTE: *rate* is rounded to three decimal places.

  – **Parameters**
    * `cost` – a `double`, the initial cost of the asset
    * `salvage` – a `double`, the salvage value of the asset
    * `life` – an `int`, the number of periods over which the asset is being depreciated
    * `period` – an `int`, the period for which the depreciation is to be computed
    * `month` – an `int`, the number of months in the first year

  – **Returns** – a `double`, the depreciation of an asset for a specified period using the fixed-declining balance method

---

- *ddb*

  **`public static double ddb( double cost, double salvage, int life, int period, double factor )`**

  – **Description**

  Returns the depreciation of an asset using the double-declining balance method. It is computed using the following:

  $$[cost - salvage\,(total\,depreciation\,from\,prior\,periods)]\,\frac{factor}{life}$$

– **Parameters**

  * `cost` – a `double`, the initial cost of the asset
  * `salvage` – a `double`, the salvage value of the asset
  * `life` – an `int`, the number of periods over which the asset is being depreciated
  * `period` – an `int`, the period
  * `factor` – a `double`, the rate at which the balance declines

– **Returns** – a `double`, the depreciation of an asset for a specified period

---

• *dollarde*
`public static double` **dollarde**`( double` **fractionalDollar**`, int` **fraction** `)`

– **Description**
Converts a fractional price to a decimal price. It is computed using the following:

$$idollar + (fractionalDollar - idollar) \times \frac{10^{(ifrac+1)}}{fraction}$$

where *idollar* is the integer part of *fractionalDollar*, and *ifrac* is the integer part of $\log(fraction)$.

– **Parameters**

  * `fractionalDollar` – a `double`, a fractional number
  * `fraction` – an `int`, the denominator

– **Returns** – a `double`, the dollar price expressed as a decimal number

---

• *dollarfr*
`public static double` **dollarfr**`( double` **decimalDollar**`, int` **fraction** `)`

– **Description**
Converts a decimal price to a fractional price. It is computed using the following:

$$idollar + \frac{decimalDollar - idollar}{10^{(ifrac+1)}/fraction}$$

where *idollar* is the integer part of the *decimalDollar*, and *ifrac* is the integer part of $log(fraction)$.

– **Parameters**

  * `decimalDollar` – a `double`, a decimal number
  * `fraction` – a `int`, the denominator

– **Returns** – a `double`, a dollar price expressed as a fraction

---

• *effect*
`public static double` **effect**`( double` **nominalRate**`, int` **nper** `)`

– **Description**
Returns the effective annual interest rate. The nominal interest rate is the periodically-compounded interest rate as stated on the face of a security. The effective annual interest rate is computed using the following:

$$\left(1+\frac{nominalRate}{nper}\right)^{nper}-1$$

– **Parameters**
  * `nominalRate` – a `double`, the nominal interest rate
  * `nper` – an `int`, the number of compounding periods per year

– **Returns** – a `double`, the effective annual interest rate

---

- *fv*
`public static double` **fv**`( double` **rate**`, int` **nper**`, double` **pmt**`, double` **pv**`, int` **when** `)`

– **Description**
Returns the future value of an investment. The future value is the value, at some time in the future, of a current amount and a stream of payments. It can be found by solving the following:
If $rate = 0$,

$$pv + pmt \times nper + fv = 0$$

If $rate \neq 0$,

$$pv(1+rate)^{nper} + pmt\left[1+rate\left(when\right)\right]\frac{(1+rate)^{nper}-1}{rate} + fv = 0$$

– **Parameters**
  * `rate` – a `double`, the interest rate
  * `nper` – an `int`, the total number of payment periods
  * `pmt` – a `double`, the payment made in each period
  * `pv` – a `double`, the present value
  * `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD

– **Returns** – a `double`, the future value of an investment

---

- *fvschedule*
`public static double` **fvschedule**`( double` **principal**`, double[]` **schedule** `)`

– **Description**
Returns the future value of an initial principal taking into consideration a schedule of compound interest rates. It is computed using the following:

$$\sum_{i=1}^{count}\left(principal \times schedule_i\right)$$

where $schedule_i$ = interest rate at the $i$th period, and the count is `schedule.length`.

– **Parameters**
  * `principal` – a `double`, the present value
  * `schedule` – a `double` array of interest rates to apply

– **Returns** – a `double`, the future value of an initial principal

---

• *ipmt*

`public static double` **`ipmt`**`( double` **`rate`**`, int` **`period`**`, int` **`nper`**`, double` **`pv`**`, double` **`fv`**`, int` **`when`** `)`

– **Description**
Returns the interest payment for an investment for a given period. It is computed using the following:

$$\left\{ pv\,(1+rate)^{nper-1} + pmt\,(1+rate \times when)\,\frac{(1+rate)^{nper-1}}{rate} \right\} rate$$

– **Parameters**
  * `rate` – a `double`, the interest rate
  * `period` – an `int`, the payment period
  * `nper` – an `int`, the total number of periods
  * `pv` – a `double`, the present value
  * `fv` – a `double`, the future value
  * `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD

– **Returns** – a `double`, the interest payment for a given period for an investment

---

• *irr*

`public static double` **`irr`**`( double[]` **`pmt`** `)`

– **Description**
Returns the internal rate of return for a schedule of cash flows. It is found by solving the following:

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1+rate)^i}$$

where $value_i$ = the $ith$ cash flow, $rate$ is the internal rate of return, and count is `pmt.length`.

– **Parameters**
  * `pmt` – a `double` array which contains cash flow values which occur at regular intervals

– **Returns** – a `double`, the internal rate of return

- *irr*

  `public static double` **irr**`( double[]` **pmt**`, double` **guess** `)`

  – **Description**

    Returns the internal rate of return for a schedule of cash flows. It is found by solving the following:

    $$0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^i}$$

    where $value_i$ = the $ith$ cash flow, $rate$ is the internal rate of return.

  – **Parameters**

    * `pmt` – a `double` array which contains cash flow values which occur at regular intervals
    * `guess` – a `double` value which represents an initial guess at the return value from this function

  – **Returns** – a `double`, the internal rate of return

- *mirr*

  `public static double` **mirr**`( double[]` **value**`, double` **financeRate**`, double` **reinvestRate** `)`

  – **Description**

    Returns the modified internal rate of return for a schedule of periodic cash flows. The modified internal rate of return differs from the ordinary internal rate of return in assuming that the cash flows are reinvested at the cost of capital, not at the internal rate of return. It also eliminates the multiple rates of return problem. It is computed using the following:

    $$\left\{ \frac{-(pnpv)(1 + reinvestRate)^{nper}}{(nnpv)(1 + financeRate)} \right\}^{\frac{1}{nper-1}} - 1$$

    where $pnpv$ is calculated from `npv` for positive values in `value` using `reinvestRate`, $nnpv$ is calculated from `npv` for negative values in `value` using `financeRate`, and `nper` = `value.length`.

  – **Parameters**

    * `value` – a `double` array of cash flows
    * `financeRate` – a `double`, the interest you pay on the money you borrow
    * `reinvestRate` – a `double`, the interest rate you receive on the cash flows

  – **Returns** – a `double`, the modified internal rate of return

- *nominal*

  `public static double` **nominal**`( double` **effectiveRate**`, int` **nper** `)`

– **Description**
Returns the nominal annual interest rate. The nominal interest rate is the interest rate as stated on the face of a security. It is computed using the following:

$$\left[(1 + \mathit{effectiveRate})^{\frac{1}{nper}} - 1\right] \times nper$$

– **Parameters**
  * `effectiveRate` – a `double`, the effective interest rate
  * `nper` – an `int`, the number of compounding periods per year

– **Returns** – a `double`, the nominal annual interest rate

---

• *nper*
`public static double` **`nper`**`( double` **`rate`**`, double` **`pmt`**`, double` **`pv`**`, double` **`fv`**`, int` **`when`** `)`

– **Description**
Returns the number of periods for an investment for which periodic, and constant payments are made and the interest rate is constant. It can be found by solving the following:
If $rate = 0$,

$$pv + pmt \times nper + fv = 0$$

If $rate \neq 0$,

$$pv(1 + rate)^{nper} + pmt\left[1 + rate\,(when)\right]\frac{(1 + rate)^{nper} - 1}{rate} + fv = 0$$

– **Parameters**
  * `rate` – a `double`, the interest rate
  * `pmt` – a `double`, the payment
  * `pv` – a `double`, the present value
  * `fv` – a `double`, the future value
  * `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD

– **Returns** – an `int`, the number of periods for an investment

---

• *npv*
`public static double` **`npv`**`( double` **`rate`**`, double[]` **`value`** `)`

– **Description**
Returns the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate. It is found by solving the following:

$$\sum_{i=1}^{count} \frac{value_i}{(1 + rate)^i}$$

where $value_i$ = the $i$th cash flow, and count is `value.length`.

- **Parameters**
  - `rate` – a `double`, the interest rate per period
  - `value` – a `double` array of equally-spaced cash flows
- **Returns** – a `double`, the net present value of the investment

---

- *pmt*

`public static double` **pmt**`( double` **rate**`, int` **nper**`, double` **pv**`, double` **fv**`, int` **when** `)`

- **Description**

  Returns the periodic payment for an investment. It can be found by solving the following:

  If $rate = 0$,

$$pv + pmt \times nper + fv = 0$$

  If $rate \neq 0$,

$$pv(1 + rate)^{nper} + pmt\left[1 + rate\,(when)\right]\frac{(1 + rate)^{nper} - 1}{rate} + fv = 0$$

- **Parameters**
  - `rate` – a `double`, the interest rate
  - `nper` – an `int`, the total number of periods
  - `pv` – a `double`, the present value
  - `fv` – a `double`, the future value
  - `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD
- **Returns** – a `double`, the interest payment for a given period for an investment

---

- *ppmt*

`public static double` **ppmt**`( double` **rate**`, int` **period**`, int` **nper**`, double` **pv**`, double` **fv**`, int` **when** `)`

- **Description**

  Returns the payment on the principal for a specified period. It is computed using the following:

$$payment_i - interest_i$$

  where $payment_i$ is computed from `pmt` for the $i$th period, $interest_i$ is calculated from `ipmt` for the $i$th period.

- **Parameters**
  - `rate` – a `double`, the interest rate
  - `period` – an `int`, the payment period

- ∗ `nper` – an `int`, the total number of periods
    - ∗ `pv` – a `double`, the present value
    - ∗ `fv` – a `double`, the future value
    - ∗ `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD
  - **Returns** – a `double`, the payment on the principal for a given period

---

- *pv*
  `public static double` **`pv`**`( double` **`rate`**`, int` **`nper`**`, double` **`pmt`**`, double` **`fv`**`, int` **`when`** `)`
  - **Description**
    Returns the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate. It can be found by solving the following:
    If $rate = 0$,
    $$pv + pmt \times nper + fv = 0$$
    If $rate \neq 0$,
    $$pv(1 + rate)^{nper} + pmt\,[1 + rate\,(when)]\,\frac{(1 + rate)^{nper} - 1}{rate} + fv = 0$$
  - **Parameters**
    - ∗ `rate` – a `double`, the interest rate per period
    - ∗ `nper` – an `int`, the number of periods
    - ∗ `pmt` – a `double`, the payment made each period
    - ∗ `fv` – a `double`, the annuity's value after the last payment
    - ∗ `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD
  - **Returns** – a `double`, the present value of the investment

---

- *rate*
  `public static double` **`rate`**`( int` **`nper`**`, double` **`pmt`**`, double` **`pv`**`, double` **`fv`**`, int` **`when`** `)`
  - **Description**
    Returns the interest rate per period of an annuity. rate is calculated by iteration and can have zero or more solutions. It can be found by solving the following:
    If $rate = 0$,
    $$pv + pmt \times nper + fv = 0$$
    If $rate \neq 0$,
    $$pv(1 + rate)^{nper} + pmt\,[1 + rate\,(when)]\,\frac{(1 + rate)^{nper} - 1}{rate} + fv = 0$$

– **Parameters**

* `nper` – an `int`, the number of periods
* `pmt` – a `double`, the payment made each period
* `pv` – a `double`, the present value
* `fv` – a `double`, the annuity's value after the last payment
* `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD

– **Returns** – a `double`, the interest rate per period of an annuity

---

• *rate*

`public static double` **rate**`( int` **nper**`, double` **pmt**`, double` **pv**`, double` **fv**`, int` **when**`, double` **guess** `)`

– **Description**

Returns the interest rate per period of an annuity with an initial guess. rate is calculated by iteration and can have zero or more solutions. It can be found by solving the following:

If $rate = 0$,

$$pv + pmt \times nper + fv = 0$$

If $rate \neq 0$,

$$pv(1 + rate)^{nper} + pmt\,[1 + rate\,(when)]\,\frac{(1 + rate)^{nper} - 1}{rate} + fv = 0$$

– **Parameters**

* `nper` – an `int`, the number of periods
* `pmt` – a `double`, the payment made each period
* `pv` – a `double`, the present value
* `fv` – a `double`, the annuity's value after the last payment
* `when` – an `int`, the time in each period when the payment is made, either AT_END_OF_PERIOD or AT_BEGINNING_OF_PERIOD
* `guess` – a `double` value which represents an initial guess at the interest rate per period of an annuity

– **Returns** – a `double`, the interest rate per period of an annuity

---

• *sln*

`public static double` **sln**`( double` **cost**`, double` **salvage**`, int` **life** `)`

– **Description**

Returns the depreciation of an asset using the straight line method. It is computed using the following:

$$cost - salvage/life$$

– **Parameters**

* `cost` – a `double`, the initial cost of the asset
* `salvage` – a `double`, the salvage value of the asset
* `life` – an `int`, the number of periods over which the asset is being depreciated

– **Returns** – a `double`, the straight line depreciation of an asset for one period

---

- *syd*

`public static double` **`syd`**`( double` **`cost`**`, double` **`salvage`**`, int` **`life`**`, int` **`per`** `)`

– **Description**

Returns the depreciation of an asset using the sum-of-years digits method. It is computed using the following:

$$(cost - salvage)(per)\ \frac{(life + 1)\,(life)}{2}$$

– **Parameters**

* `cost` – a `double`, the initial cost of the asset
* `salvage` – a `double`, the salvage value of the asset
* `life` – an `int`, the number of periods over which the asset is being depreciated
* `per` – an `int`, the period

– **Returns** – a `double`, the sum-of-years digits depreciation of an asset

---

- *vdb*

`public static double` **`vdb`**`( double` **`cost`**`, double` **`salvage`**`, int` **`life`**`, int` **`start`**`, int` **`end`**`, double` **`factor`**`, boolean` **`no_sl`** `)`

– **Description**

Returns the depreciation of an asset for any given period using the variable-declining balance method. It is computed using the following:
If $no\_sl = 0$,

$$\sum_{i=start+1}^{end} ddb_i$$

If $no\_sl \neq 0$,

$$A + \sum_{i=k}^{end} \frac{cost - A - salvage}{end - k + 1}$$

where $ddb_i$ is computed from `ddb` for the $i$th period. $k =$ the first period where straight line depreciation is greater than the depreciation using the double-declining balance method.

$$A = \sum_{i=start+1}^{k-1} ddb_i$$

- **Parameters**
  - `cost` – a `double`, the initial cost of the asset
  - `salvage` – a `double`, the salvage value of the asset
  - `life` – an `int`, the number of periods over which the asset is being depreciated
  - `start` – an `int`, the initial period for the calculation
  - `end` – an `int`, the final period for the calculation
  - `factor` – a `double`, the rate at which the balance declines
  - `no_sl` – a `boolean` flag. If true, do not switch to straight-line depreciation even when the depreciation is greater than the declining balance calculation.
- **Returns** – a `double`, the depreciation of the asset

---

- *xirr*

`public static double` **`xirr`**`( double[]` **`pmt`**`, java.util.Date[]` **`dates`** `)`

- **Description**

  Returns the internal rate of return for a schedule of cash flows. It is not necessary that the cash flows be periodic. It can be found by solving the following:

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^{\frac{d_i - d_1}{365}}}$$

  In the equation above, $d_i$ represents the $i$th payment date. $d_1$ represents the 1st payment date. *value* represents the $i$th cash flow. *rate* is the internal rate of return, and count is `pmt.length`.
- **Parameters**
  - `pmt` – a `double` array which contains cash flow values which correspond to a schedule of payments in dates
  - `dates` – a `Date` array which contains a schedule of payment dates
- **Returns** – a `double`, the internal rate of return

---

- *xirr*

`public static double` **`xirr`**`( double[]` **`pmt`**`, java.util.Date[]` **`dates`**`, double` **`guess`** `)`

- **Description**

  Returns the internal rate of return for a schedule of cash flows with a user supplied initial guess. It is not necessary that the cash flows be periodic. It can be found by solving the following:

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^{\frac{d_i - d_1}{365}}}$$

In the equation above, $d_i$ represents the $i$th payment date. $d_1$ represents the 1st payment date. *value* represents the $i$th cash flow. *rate* is the internal rate of return. Count is `pmt.length`.

- **Parameters**
  - `pmt` – a `double` array which contains cash flow values which correspond to a schedule of payments in dates
  - `dates` – a `Date` array which contains a schedule of payment dates
  - `guess` – a `double` value which represents an initial guess at the return value from this function
- **Returns** – a `double`, the internal rate of return

---

- *xnpv*

  `public static double` **`xnpv`**`( double` **`rate`**`, double[]` **`value`**`, java.util.Date[]` **`dates`** `)`

  - **Description**

    Returns the present value for a schedule of cash flows. It is not necessary that the cash flows be periodic. It is computed using the following:

$$\sum_{i=1}^{count} \frac{value_i}{(1 + rate)^{(d_i - d_1)/365}}$$

    In the equation above, $d_i$ represents the $i$th payment date, $d_1$ represents the first payment date, $value_i$ represents the $i$th cash flow. and count is `value.length`
  - **Parameters**
    - `rate` – a `double`, the interest rate
    - `value` – a `double` array containing the cash flows
    - `dates` – a `Date` array which contains a schedule of payment dates
  - **Returns** – a `double`, the present value

### Example: Cumulative Interest Example

The amount of interest paid in the first year of a 30 year fixed rate mortgage is computed. The amount financed is $200,000 at an interest rate of 7.25% for 30 years.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class cumipmtEx1 {
    public static void main(String args[]) {
        double rate = 0.0725/12;
        int periods = 12*30;
```

```
        double pv = 200000;
        int start = 1;
        int end = 12;
        double total;

        total = Finance.cumipmt(rate, periods, pv, start, end,
        Finance.AT_END_OF_PERIOD);

        System.out.println("First year interest = " +
        NumberFormat.getCurrencyInstance().format(total));
    }
}
```

### Output

```
First year interest = ($14,436.52)
```

### Example: Cumulative Principal Example

The amount of principal paid in the first year of a 30 year fixed rate mortgage is computed. The amount financed is $200,000 at an interest rate of 7.25% for 30 years.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class cumprincEx1 {
    public static void main(String args[]) {
        double rate = 0.0725/12;
        int periods = 12*30;
        double pv = 200000;
        int start = 1;
        int end = 12;
        double total;

        total = Finance.cumprinc(rate, periods, pv, start, end,
        Finance.AT_END_OF_PERIOD);

        System.out.println("First year principal = " +
        NumberFormat.getCurrencyInstance().format(total));
    }
}
```

### Output

```
First year principal = ($1,935.71)
```

### Example: Depreciation - Fixed Declining Balance Method

The depreciation of an asset with an initial cost of $2500 and a salvage value of $500 over a period of 3 years is calculated. Here month is 6 since the life of the asset did not begin until the seventh month of the first year.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class dbEx1 {
    public static void main(String args[]) {
        double cost = 2500;
        double salvage = 500;
        int life = 3;
        int month = 6;

        for (int period = 1;  period <= life+1;  period++) {
            double db = Finance.db(cost, salvage, life, period, month);
            System.out.println("For period "+period+"    db = " +
            NumberFormat.getCurrencyInstance().format(db));
        }
    }
}
```

### Output

```
For period 1    db = $518.75
For period 2    db = $822.22
For period 3    db = $481.00
For period 4    db = $140.69
```

### Example: Depreciation - Double-Declining Balance Method

The depreciation of an asset with an initial cost of $2500 and a salvage value of $500 over a period of 2 years is calculated. A factor of 2 is used (the double-declining balance method).

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class ddbEx1 {
    public static void main(String args[]) {
        double cost = 2500;
        double salvage = 500;
        double factor = 2;
        int life = 24;

        for (int period = 1;  period <= life;  period++) {
            double ddb = Finance.ddb(cost, salvage, life, period, factor);
            System.out.println("For period "+period+"    ddb = " +
            NumberFormat.getCurrencyInstance().format(ddb));
        }
    }
}
```

## Output

```
For period 1    ddb = $208.33
For period 2    ddb = $190.97
For period 3    ddb = $175.06
For period 4    ddb = $160.47
For period 5    ddb = $147.10
For period 6    ddb = $134.84
For period 7    ddb = $123.60
For period 8    ddb = $113.30
For period 9    ddb = $103.86
For period 10    ddb = $95.21
For period 11    ddb = $87.27
For period 12    ddb = $80.00
For period 13    ddb = $73.33
For period 14    ddb = $67.22
For period 15    ddb = $61.62
For period 16    ddb = $56.48
For period 17    ddb = $51.78
For period 18    ddb = $47.46
For period 19    ddb = $22.09
For period 20    ddb = $0.00
For period 21    ddb = $0.00
```

```
For period 22    ddb = $0.00
For period 23    ddb = $0.00
For period 24    ddb = $0.00
```

### Example: Price Conversion - Fractional Dollars

A fractional dollar price, in this case 1 3/8, is converted to a decimal price.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class dollardeEx1 {
    public static void main(String args[]) {
        double fractionalDollar = 1.3;
        int fraction = 8;

        double dollardec = Finance.dollarde(fractionalDollar, fraction);
        System.out.println("The fractional dollar 1.3 = " +
        NumberFormat.getCurrencyInstance().format(dollardec));
    }
}
```

### Output

```
The fractional dollar 1.3 = $1.38
```

### Example: Price Conversion - Decimal Dollars

A decimal dollar price, in this case $1.38, is converted to a fractional price.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class dollarfrEx1 {
    public static void main(String args[]) {
        double decimalDollar = 1.38;
        int fraction = 8;

        double dollarfrc = Finance.dollarfr(decimalDollar, fraction);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("The decimal dollar $1.38 as a fractional dollar = "
```

```
        + nf.format(dollarfrc));
    }
}
```

### Output

```
The decimal dollar $1.38 as a fractional dollar = 1.3
```

### Example: Effective Rate

In this example the effective interest rate is computed given that the nominal rate is 6.0% and that the interest will be compounded quarterly.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class effectEx1 {
    public static void main(String args[]) {
        double nominalRate = .06;
        int nper = 4;
        double effectiveRate;

        effectiveRate = Finance.effect(nominalRate, nper);
        NumberFormat nf = NumberFormat.getPercentInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("The effective rate of the nominal rate, 6.0%, " +
        "compounded quarterly is " +nf.format(effectiveRate));
    }
}
```

### Output

```
The effective rate of the nominal rate, 6.0%, compounded quarterly is 6.14%
```

### Example: Future Value of an Investment

A couple starts setting aside $30,000 a year when they are 45 years old. They expect to earn 5% interest on the money compounded yearly. The future value of the investment is computed for a 20 year period.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class fvEx1 {
    public static void main(String args[]) {
        double rate = .05;
        int nper = 20;
        double payment = -30000.00;
        double pv = -30000.00;
        int when = Finance.AT_BEGINNING_OF_PERIOD;

        double fv = Finance.fv(rate, nper, payment, pv, when);
        System.out.println("After 20 years, the value of the investments " +
        "will be " + NumberFormat.getCurrencyInstance().format(fv));
    }
}
```

## Output

```
After 20 years, the value of the investments will be $1,121,176.49
```

### Example: Future Value - Adustable Rates

An investment of $10,000 is made. The investment will grow at the rate of 5.1% the first year, with the rate increasing by .1% each year thereafter for a total of 5 years. The future value of the investment is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class fvscheduleEx1 {
    public static void main(String args[]) {
        double principal = 10000.0;
        double[] schedule = {.050, .051, .052, .053, .054};
        double fvschedule;

        fvschedule = Finance.fvschedule(principal, schedule);
        System.out.println("After 5 years the $10,000 investment will have " +
        "grown to " + NumberFormat.getCurrencyInstance().format(fvschedule));
    }
}
```

### Output

```
After 5 years the $10,000 investment will have grown to $12,884.77
```

### Example: Interest Payments

The interest due the second year on a $100,000 25 year loan is calculated. The loan is at 8%.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class ipmtEx1 {
    public static void main(String args[]) {
        double rate = .08;
        int per = 2;
        int nper = 25;
        double pv = 100000.00;
        double fv = 0.0;
        int when = Finance.AT_END_OF_PERIOD;

        double ipmt = Finance.ipmt(rate, per, nper, pv, fv, when);
        System.out.println("The interest due the second year on the " +
        "$100,000 loan is " + NumberFormat.getCurrencyInstance().format(ipmt));
    }
}
```

### Output

```
The interest due the second year on the $100,000 loan is ($7,890.57)
```

### Example: Internal Rate of Return

A farmer buys 10 young cows and a bull for $4500. The first year he does not expect to sell any calves, he just expects to feed them. Thereafter, he expects to be able to sell calves to offset the cost of feed. He expects them to be productive for 9 years, after which time he will liquidate the herd. The internal rate of return is computed after 9 years.

```
import com.imsl.finance.*;
import java.text.NumberFormat;
```

```
public class irrEx1 {
    public static void main(String args[]) {
        double[] pmt = {-4500., -800., 800., 800., 600., 600.,
        800., 800., 700., 3000.};

        double irr = Finance.irr(pmt);
        NumberFormat nf = NumberFormat.getPercentInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("After 9 years, the internal rate of return on " +
        "the cows is " + nf.format(irr));
    }
}
```

## Output

```
After 9 years, the internal rate of return on the cows is 7.21%
```

## Example: Modified Internal Rate of Return

A farmer uses a $4500 loan to buy 10 young cows and a bull. The interest rate on the loan is 8%. He expects to reinvest the profits received in any one year in the money market and receive 5.5%. The first year he does not expect to sell any calves, he just expects to feed them. Thereafter, he expects to be able to sell calves to offset the cost of feed. He expects them to be productive for 9 years, after which time he will liquidate the herd. The modified internal rate of return is computed after 9 years.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class mirrEx1 {
    public static void main(String args[]) {
        double[] value = {-4500., -800., 800., 800., 600., 600.,
        800., 800., 700., 3000.};
        double financeRate = .08;
        double reinvestRate = .055;
        double mirr = Finance.mirr(value, financeRate, reinvestRate);
        NumberFormat nf = NumberFormat.getPercentInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("After 9 years, the modified internal rate of " +
        "return on the cows is " +nf.format(mirr));
    }
```

```
}
```

### Output

```
After 9 years, the modified internal rate of return on the cows is 6.66%
```

### Example: Nominal Rate

In this example the nominal interest rate is computed given that the effective rate is 6.14% and that the interest has been compounded quarterly.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class nominalEx1 {
    public static void main(String args[]) {
        double effectiveRate = .0614;
        int nper = 4;

        double nominalRate = Finance.nominal(effectiveRate, nper);
        NumberFormat nf = NumberFormat.getPercentInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("The nominal rate of the effective rate, 6.14%, " +
        "compounded quarterly is " + nf.format(nominalRate));
    }
}
```

### Output

```
The nominal rate of the effective rate, 6.14%, compounded quarterly is 6%
```

### Example: Number of Periods for an Investment

Someone obtains a $20,000 loan at 7.25% to buy a car. They want to make $350 a month payments. Here, the number of payments necessary to pay off the loan is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class nperEx1 {
```

```
    public static void main(String args[]) {
        double rate = 0.0725/12;
        double pmt = -350.;
        double pv = 20000;
        double fv = 0.;
        int when = Finance.AT_BEGINNING_OF_PERIOD;
        double nperiods;

        nperiods = Finance.nper(rate, pmt, pv, fv, when);

        System.out.println("Number of payment periods = " +nperiods);
    }
}
```

### Output

```
Number of payment periods = 69.78051136628257
```

### Example: Net Present Value of an Investment

A lady wins a $10 million lottery. The money is to be paid out at the end of each year in $500,000 payments for 20 years. The current treasury bill rate of 6% is used as the discount rate. Here, the net present value of her prize is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class npvEx1 {
    public static void main(String args[]) {
        double rate = 0.06;
        double[] value = new double[20];

        for (int i = 0; i < 20; i++) value[i] = 500000.;
        double npv = Finance.npv(rate, value);

        System.out.println("The net present value of the $10 million " +
        "prize is " + NumberFormat.getCurrencyInstance().format(npv));
    }
}
```

## Output

```
The net present value of the $10 million prize is $5,734,960.61
```

## Example: Periodic Payments

The payment due each year on a 25 year, $100,000 loan is calculated. The loan is at 8%.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class pmtEx1 {
    public static void main(String args[]) {
        double rate = .08;
        int nper = 25;
        double pv = 100000.00;
        double fv = 0.0;
        int when = Finance.AT_END_OF_PERIOD;

        double pmt = Finance.pmt(rate, nper, pv, fv, when);
        System.out.println("The payment due each year on the $100,000 loan is "
        + NumberFormat.getCurrencyInstance().format(pmt));
    }
}
```

## Output

```
The payment due each year on the $100,000 loan is ($9,367.88)
```

## Example: Principal Payments

The payment on the principal the first year on a 25 year, $100,000 loan is calculated. The loan is at 8%.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class ppmtEx1 {
    public static void main(String args[]) {
        double rate = .08;
        int per = 1;
```

```
        int nper = 25;
        double pv = 100000.00;
        double fv = 0.0;
        int when = Finance.AT_END_OF_PERIOD;

        double ppmt = Finance.ppmt(rate, per, nper, pv, fv, when);
        System.out.println("The payment on the principal the first year " +
        "of the $100,000 loan is " +
        NumberFormat.getCurrencyInstance().format(ppmt));
    }
}
```

### Output

```
The payment on the principal the first year of the $100,000 loan is ($1,367.88)
```

### Example: Present Value of an Investment

A lady wins a $10 million lottery. The money is to be paid out at the end of each year in $500,000 payments for 20 years. The current treasury bill rate of 6% is used as the discount rate. Here, the present value of her prize is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class pvEx1 {
    public static void main(String args[]) {
        double rate = 0.06;
        double pmt = 500000.;
        double fv = 0.;
        int nper = 20;
        int when = Finance.AT_END_OF_PERIOD;

        double pv = Finance.pv(rate, nper, pmt, fv, when);

        System.out.println("The present value of the $10 million prize is " +
        NumberFormat.getCurrencyInstance().format(pv));
    }
}
```

## Output

```
The present value of the $10 million prize is ($5,734,960.61)
```

## Example: Interest Rate

Someone obtains a $20,000 loan to buy a car. They make $350 a month payments for 70 months. Here, the interest rate of the loan is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class rateEx1 {
    public static void main(String args[]) {
        double rate;
        int nper = 70;
        double pmt = -350.;
        double pv = 20000;
        double fv = 0.;
        int when = Finance.AT_BEGINNING_OF_PERIOD;

        rate = Finance.rate(nper, pmt, pv, fv, when)*12;
        NumberFormat nf = NumberFormat.getPercentInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("The computed interest rate on the loan is " +
        nf.format(rate));
    }
}
```

## Output

```
The computed interest rate on the loan is 7.35%
```

## Example: Depreciation - Straight Line Method

The straight line depreciation for one period of an asset with a life of 24 months, an initial cost of $2500 and a salvage value of $500 is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;
```

```
public class slnEx1 {
    public static void main(String args[]) {
        double cost = 2500;
        double salvage = 500;
        int life = 24;

        double sln = Finance.sln(cost, salvage, life);
        System.out.println("The straight line depreciation of the asset " +
        "for one period is " + NumberFormat.getCurrencyInstance().format(sln));
    }
}
```

### Output

```
The straight line depreciation of the asset for one period is $83.33
```

### Example: Depreciation - Sum-of-years' Digits

The sum-of-years' digits depreciation for the 14th year of an asset with a life of 15 years, an initial cost of $25000 and a salvage value of $5000 is computed.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class sydEx1 {
    public static void main(String args[]) {
        double cost = 25000;
        double salvage = 5000;
        int life = 15;
        int per = 14;

        double syd = Finance.syd(cost, salvage, life, per);
        System.out.println("The depreciation allowance for the 14th year is " +
        NumberFormat.getCurrencyInstance().format(syd));
    }
}
```

### Output

```
The depreciation allowance for the 14th year is $333.33
```

### Example: Depreciation - Variable Declining Balance

The depreciation between the 10th and 15th year of an asset with a life of 15 years, an initial cost of $25000 and a salvage value of $5000 is computed. The variable-declining balance method is used.

```
import com.imsl.finance.*;
import java.text.NumberFormat;

public class vdbEx1 {
    public static void main(String args[]) {
        double cost = 25000;
        double salvage = 5000;
        int life = 15;
        int start = 10;
        int end = 15;
        double factor = 2.;
        boolean no_sl = false;

        double vdb = Finance.vdb(cost, salvage, life, start, end,
        factor, no_sl);
        System.out.println("The depreciation allowance between the " +
        "10th and 15th year is " +
        NumberFormat.getCurrencyInstance().format(vdb));
    }
}
```

### Output

```
The depreciation allowance between the 10th and 15th year is $976.69
```

### Example: Internal Rate of Return - Variable Schedule

A farmer buys 10 young cows and a bull for $4500. The first year he does not expect to sell any calves, he just expects to feed them. Thereafter, he expects to be able to sell calves to offset the cost of feed. He expects them to be productive for 9 years, after which time he will liquidate the herd. The internal rate of return is computed after 9 years.

```
import com.imsl.finance.*;
import java.text.NumberFormat;
import java.text.*;
import java.util.*;
```

```
public class xirrEx1 {
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    private static Date parse(String s) throws ParseException {
        return dateFormat.parse(s);
    }

    public static void main(String args[]) throws ParseException {
        double[] pmt = {-4500., -800., 800., 800., 600., 600.,
        800., 800., 700., 3000.};
        Date dates[] = {
            parse("1/1/98"), parse("10/1/98"), parse("5/5/99"),
            parse("5/5/00"), parse("6/1/01"), parse("7/1/02"),
            parse("8/30/03"), parse("9/15/04"), parse("10/15/05"),
            parse("11/1/06")
        };
        double xirr = Finance.xirr(pmt, dates);
        NumberFormat nf = NumberFormat.getPercentInstance();
        nf.setMaximumFractionDigits(2);
        System.out.println("After approximately 9 years, the internal rate " +
        "of return on the cows is " + nf.format(xirr));
    }
}
```

## Output

```
After approximately 9 years, the internal rate of return on the cows is 7.69%
```

## Example: Present Value of a Schedule of Cash Flows

In this example, the present value of 3 payments, $1,000, $2,000, and $1,000, with an interest rate of 5% made on January 3, 1997, January 3, 1999, and January 3, 2000 is computed.

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class xnpvEx1 {
```

```
    static final DateFormat dateFormat =
    DateFormat.getDateInstance(DateFormat.SHORT,  Locale.US);

    private static Date parse(String s) throws ParseException {
        return dateFormat.parse(s);
    }

    public static void main(String args[]) throws ParseException {
        double rate = 0.05;
        double value[]= {1000.,2000., 1000.};
        Date dates[] = {parse("1/3/1997"), parse("1/3/1999"),
        parse("1/3/2000")};

        double pv = Finance.xnpv(rate, value, dates);
        System.out.println("The present value of the schedule of cash " +
        "flows is " + NumberFormat.getCurrencyInstance().format(pv));
    }
}
```

## Output

```
The present value of the schedule of cash flows is $3,677.90
```

# Chapter 24

# Charting

Classes

# *class* **Chart**

The root node of the chart tree.

This chart node creates the following child nodes: com.imsl.chart.Background, com.imsl.chart.ChartTitle and com.imsl.chart.Legend.

## Declaration

public class com.imsl.chart.Chart
**extends** com.imsl.chart.ChartNode (page 920)
**implements** java.lang.Cloneable, java.awt.print.Printable

## Constructors

- *Chart*
  `public` **Chart**`( )`
  - **Description**
    This is the root of our tree, it has no parent. This creates the Chart with a null component

- *Chart*
  `public` **Chart**`( java.awt.Component` **component** `)`
  - **Description**
    This is the root of our tree, it has no parent. This creates the Chart with the named component
  - **Parameters**
    * `component` – the `Component` that contains the chart.

- *Chart*
  `public` **Chart**`( java.awt.Image` **image** `)`
  - **Description**
    This is the root of our tree, it has no parent. This creates the Chart drawn into the image.
  - **Parameters**
    * `image` – the `Image` into which the chart is to be drawn.

## Methods

---

- *addLegendItem*
  `public void` **addLegendItem**`( int` **type**`, ChartNode` **node** `)`
  - **Description**
    Adds a legend to this `ChartNode`
  - **Parameters**
    - `type` – an `int` which specifies the `LegendItem` type. 0 = DATA_TYPE_NONE; 1 = DATA_TYPE_LINE; 2 = DATA_TYPE_MARKER; 3 = DATA_TYPE_FILL
    - `node` – the `ChartNode` object to which this legend is to be added

---

- *addMouseListener*
  `public void` **addMouseListener**`( java.awt.event.MouseListener` **listener** `)`
  - **Description**
    Adds a MouseListener to the component associated with this chart. If the component is null the listener will be saved and added to the component when it is assigned.

---

- *addMouseMotionListener*
  `public void` **addMouseMotionListener**`( java.awt.event.MouseMotionListener` **listener** `)`
  - **Description**
    Adds a MouseMotionListener to the component associated with this chart. If the component is null the listener will be saved and added to the component when it is assigned.

---

- *clone*
  `public java.lang.Object` **clone**`( )`
  - **Description**
    Returns a clone of the graphics tree.
  - **Returns** – an `Object` which is a clone of this graphics tree

---

- *clone*
  `protected java.lang.Object` **clone**`( java.util.Hashtable` **hashClonedNode** `)`
  - **Description**
    Returns a clone of this node.

– **Parameters**
  * `hashClonedNode` – the `Hashtable` to be cloned
– **Returns** – an `Object` which is a clone of this node

---

- *copy*
`public void` **copy**`( )`

  – **Description**
  Copy the chart to the clipboard.

---

- *finalize*
`protected void` **finalize**`( ) throws java.lang.Throwable`

---

- *paint*
`public synchronized void` **paint**`( Draw` **draw** `)`

  – **Description**
  Paints this node and all of its children.
  – **Parameters**
    * `draw` – a `Draw` object to be painted

---

- *paint*
`public void` **paint**`( java.awt.Graphics` **g** `)`

  – **Description**
  Paints this node and all of its children. This should be called whenever the paint member function in the Component used in this object's constructor is called.
  – **Parameters**
    * `g` – `Graphics` object to be painted

---

- *paintChart*
`public void` **paintChart**`( java.awt.Graphics` **graphics** `)`

  – **Description**
  Draw the chart using the given Graphics object.
  – **Parameters**
    * `graphics` – is the object for which the chart is to be drawn.

---

- *paintImage*
`public java.awt.Image` **paintImage**`( )`

  – **Description**
  Returns an Image of the chart.

– **Returns** – an `Image` containing a picture of the chart. Call flush() on the image when it is no longer needed.

---

- *pick*
  `public void` **pick**`( java.awt.event.MouseEvent` **event** `)`

  – **Description**
  Fire the PickListeners for the nodes hit by the event.

  – **Parameters**

  * `event` – `MouseEvent/code whose position determines which nodes have been selected`

---

- *print*
  `public int` **print**`( java.awt.Graphics` **graphics**`, java.awt.print.PageFormat` **pageFormat**`, int` **param** `) throws java.awt.print.PrinterException`

  – **Description**
  This method implements the Printable interface. It prints the chart on a single page. The output is scaled to fill the page as much as possible while preserving the aspect ratio.

---

- *repaint*
  `public void` **repaint**`( )`

  – **Description**
  Prepares the chart to be repainted by deleting any double buffering image.

---

- *setComponent*
  `public void` **setComponent**`( java.awt.Component` **component** `)`

  – **Description**
  Sets the Component for this chart. Also registers MouseListeners or MouseMotionListeners that could not be added previously.

---

- *update*
  `public void` **update**`( java.awt.Graphics` **g** `)`

---

- *writePNG*
  `public void` **writePNG**`( java.io.OutputStream` **os**`, int` **width**`, int` **height** `) throws java.io.IOException`

  – **Description**
  Writes the chart as an PNG file. PNG () is a lossless bitmap format. This method requires either J2SE 1.4 or later .

– **Parameters**
  * `os` – is the output stream to which the PNG image is to be written.
  * `width` – is the width of the output image.
  * `height` – is the height of the output image.
– **Throws**
  * `java.io.IOException` – if there is a problem writing the image to the stream.
  * `java.lang.NoClassDefFoundError` – if an older version of J2SE is used and the Java Advanced Imaging Toolkit cannot be found.

---

• *writeSVG*
`public void` **writeSVG**`( java.io.Writer` **writer**`, boolean` **useCSS** `) throws java.io.IOException`

– **Description**
  Writes the chart as an SVG file. This method requires the library.
– **Parameters**
  * `writer` – is the output character stream
  * `useCSS` – is true if the CSS style attribute is to be used
– **Throws**
  * `java.io.IOException` – if there is a problem writing the file.
  * `java.lang.NoClassDefFoundError` – if the Batik library cannot be found.

## *class* ChartNode

The base class of all of the nodes in the chart tree.

### Declaration

public abstract class com.imsl.chart.ChartNode
**extends** java.lang.Object
**implements** java.io.Serializable, java.lang.Cloneable

### Fields

---

• public static final `int` **AXIS_X**
  – Flag to indicate x-axis.

- public static final `int` **AXIS_Y**
  - Flag to indicate y-axis.
- public static final `int` **AXIS_X_TOP**
  - Flag to indicate x-axis placed on top of the chart.
- public static final `int` **AXIS_Y_RIGHT**
  - Flag to indicate y-axis placed to the right of the chart.
- public static final `int` **AUTOSCALE_OFF**
  - Flag used to indicate that autoscaling is turned off.
- public static final `int` **AUTOSCALE_DATA**
  - Flag used to indicate that autoscaling is to be done by scanning the data nodes.
- public static final `int` **AUTOSCALE_WINDOW**
  - Flag used to indicate that autoscaling is to be done by using the "Window" attribute.
- public static final `int` **AUTOSCALE_NUMBER**
  - Flag used to indicate that autoscaling is to adjust the "Number" attribute.
- public static final `int` **AUTOSCALE_DENSITY**
  - Flag used to indicate that autoscaling is to adjust the "Density" attribute. This applies only to time axes.
- public static final `int` **BAR_TYPE_VERTICAL**
  - Flag to indicate a vertical bar chart.
- public static final `int` **BAR_TYPE_HORIZONTAL**
  - Flag to indicate a horizontal bar chart.
- public static final `int` **DATA_TYPE_LINE**
  - Value for attribute "DataType" indicating that the data points should be connected with line segments. This is the default setting.
- public static final `int` **DATA_TYPE_MARKER**
  - Value for attribute "DataType" indicating that a marker should be drawn at each data point.
- public static final `int` **DATA_TYPE_FILL**

– Value for attribute “DataType” indicating that the area between the lines connecting the data points and the horizontal reference line (y = attribute “Reference”) should be filled. This is an area chart.

- public static final `int` **DATA_TYPE_PICTURE**
  – Value for attribute “DataType” indicating that an image (attribute “Image”) should be drawn at each data point. This can be used to draw fancy markers.
- public static final `double[]` **DASH_PATTERN_SOLID**
  – Flag to draw solid line.
- public static final `double[]` **DASH_PATTERN_DOT**
  – Flag to draw a dotted line.
- public static final `double[]` **DASH_PATTERN_DASH**
  – Flag to draw a dashed line.
- public static final `double[]` **DASH_PATTERN_DASH_DOT**
  – Flag to draw a dash-dot pattern line.
- public static final `int` **FILL_TYPE_NONE**
  – Value for attribute “FillType” and “FillOutlineType” indicating that the region is not to be drawn.
- public static final `int` **FILL_TYPE_SOLID**
  – Value for attribute “FillType” and “FillOutlineType” indicating that the region is to be drawn using the solid color specified by the attribute FillColor or FillOutlineColor.
- public static final `int` **FILL_TYPE_GRADIENT**
  – Value for attribute “FillType” indicating that the region is to be drawn in a color gradient as specified by the attribute Gradient.
- public static final `int` **FILL_TYPE_PAINT**
  – Value for attribute “FillType” indicating that the region is to be drawn using the texture specified by the attribute FillPaint.
- public static final `int` **LABEL_TYPE_NONE**
  – Flag used to indicate the an element is not to be labeled.
- public static final `int` **LABEL_TYPE_X**
  – Flag used to indicate that an element is to be labeled with the value of its x-coordinate.

- public static final `int` **LABEL_TYPE_Y**
  - Flag used to indicate that an element is to be labeled with the value of its y-coordinate.
- public static final `int` **LABEL_TYPE_TITLE**
  - Flag used to indicate that an element is to be labeled with the value of its title attribute.
- public static final `int` **LABEL_TYPE_PERCENT**
  - Flag used to indicate that a pie slice is to be labeled with a percentage value. This attribute only applies to pie charts.
- public static final `int` **MARKER_TYPE_PLUS**
  - Flag for a plus-shaped data marker.
- public static final `int` **MARKER_TYPE_ASTERISK**
  - Flag for a asterisk data marker.
- public static final `int` **MARKER_TYPE_X**
  - Flag for a x-shaped data marker.
- public static final `int` **MARKER_TYPE_HOLLOW_SQUARE**
  - Flag for a hollow square data marker.
- public static final `int` **MARKER_TYPE_FILLED_SQUARE**
  - Flag for a filled square data marker.
- public static final `int` **MARKER_TYPE_HOLLOW_TRIANGLE**
  - Flag for hollow triangle data marker.
- public static final `int` **MARKER_TYPE_FILLED_TRIANGLE**
  - Flag for a filled triangle data marker.
- public static final `int` **MARKER_TYPE_HOLLOW_DIAMOND**
  - Flag for a hollow diamond data marker.
- public static final `int` **MARKER_TYPE_FILLED_DIAMOND**
  - Flag for a filled diamond data marker.
- public static final `int` **MARKER_TYPE_DIAMOND_PLUS**
  - Flag for a plus in a diamond data marker.
- public static final `int` **MARKER_TYPE_SQUARE_X**

– Flag for an x in a square data marker.

- public static final `int` **MARKER_TYPE_SQUARE_PLUS**
  – Flag for a plus in a square data marker.
- public static final `int` **MARKER_TYPE_OCTAGON_X**
  – Flag for a x in an octagon data marker.
- public static final `int` **MARKER_TYPE_OCTAGON_PLUS**
  – Flag for a plus in an octagon data marker.
- public static final `int` **MARKER_TYPE_HOLLOW_CIRCLE**
  – Flag for a hollow circle data marker.
- public static final `int` **MARKER_TYPE_FILLED_CIRCLE**
  – Flag for a filled circle data marker.
- public static final `int` **MARKER_TYPE_CIRCLE_X**
  – Flag for an x in a circle data marker.
- public static final `int` **MARKER_TYPE_CIRCLE_PLUS**
  – Flag for a plus in a circle data marker.
- public static final `int` **MARKER_TYPE_CIRCLE_CIRCLE**
  – Flag for a circle in a circle data marker.
- public static final `int` **TEXT_X_LEFT**
  – Value for attribute "TextAlignment" indicating that the text should be left adjusted. This is the default setting.
- public static final `int` **TEXT_X_CENTER**
  – Value for attribute "TextAlignment" indicating that the text should be centered.
- public static final `int` **TEXT_X_RIGHT**
  – Value for attribute "TextAlignment" indicating that the text should be right adjusted.
- public static final `int` **TEXT_Y_BOTTOM**
  – Value for attribute "TextAlignment" indicating that the text should be drawn on the baseline. This is the default setting.
- public static final `int` **TEXT_Y_CENTER**

– Value for attribute "TextAlignment" indicating that the text should be vertically centered.

- public static final `int` **TEXT_Y_TOP**
  – Value for attribute "TextAlignment" indicating that the text should be drawn with the top of the letters touching the top of the drawing region.
- public static final `int` **TRANSFORM_LINEAR**
  – Flag used to indicate that the axis uses linear scaling.
- public static final `int` **TRANSFORM_LOG**
  – Flag used to indicate that the axis uses logarithmic scaling.
- public static final `int` **TRANSFORM_CUSTOM**
  – Flag used to indicate that the axis using a custom transformation.

### Constructor

---

- *ChartNode*
  `public` **ChartNode**`( ChartNode` **parent** `)`
  – **Description**
  Construct a `ChartNode` object.
  – **Parameters**
  * `parent` – the `ChartNode` parent of this object

### Methods

---

- *addPickListener*
  `public void` **addPickListener**`( PickListener` **pickListener** `)`
  – **Description**
  Adds a PickListener to this node. Unlike simple attributes, the pickListener is added to a list of existing PickListeners defined at this node. The existing listeners remain defined at this node. If this pickListener is already registered in this node, it will not be added again.
  – **Parameters**
  * `pickListener` – the `PickListener` to be added to this node

---

- *clone*
  `protected java.lang.Object` **clone**`( java.util.Hashtable` **hashClonedNode** `)`

  – **Description**
  Returns a deep-copy clone of this node. Each class derived from this class should override this function IF the derived class contains ChartNode objects or double[] arrays as member data. The overriden function should call this function and then clone each of its ChartNode data members. For example, in AxisXY we have

```
protected Object clone(Hashtable hashClonedNode)
{
    AxisXY t = (AxisXY)super.clone(hashClonedNode);
    t.axisX = (Axis1D)axisX.clone(hashClonedNode);
    t.axisY = (Axis1D)axisY.clone(hashClonedNode);
    return t;
}
```

  – **Parameters**
    * `hashClonedNode` – `Hashtable` of nodes that have already been cloned. We need to clone each ChartNode exactly once even if multiple references to it exist in the graphics tree. In this hashtable keys are existing ChartNode objects and values are their clones.

---

- *clone*
  `protected final java.util.Hashtable` **clone**`( java.util.Hashtable` **hashIn**`, java.util.Hashtable` **hashClonedNode** `)`

  – **Description**
  Returns a deep copy of a Hashtable. We assume the keys are immutable (e.g. Strings) and so do not have to be cloned. We cannot just use Hashtable.clone() because we want to specially handle cloning of ChartNodes that may occur in the hashtable. (Need to clone each ChartNode exactly once even if multiple references to it exist in the graphics tree.)

---

- *clone*
  `protected java.lang.Object` **clone**`( java.lang.Object` **value**`, java.util.Hashtable` **hashClonedNode** `)`

  – **Description**
  Returns a deep copy of an Object. Handles non-immutable object types ChartNode, Hashtable, Vector, double[], String[], and int[]. (Immutable objects can just be reused, they do not have to be cloned.)

If other non-immutable object types are used in the tree then the nodes where they are defined should override this function to handle the cloning. The new function calls super.clone(value, hashClonedNode) for values handled here.

---

- *clone*
`protected final java.util.Vector` **clone**`( java.util.Vector` **vecIn**, `java.util.Hashtable` **hashClonedNode** `)`
  - **Description**
    Returns a deep copy of a vector of ChartNode's.

---

- *firePickListeners*
`public void` **firePickListeners**`( java.awt.event.MouseEvent` **event** `)`
  - **Description**
    Fires the pick listeners defined at this node and at all of its ancestors, if the event "hits" the node.
  - **Parameters**
    - `event` – `MouseEvent` which determines which nodes have been selected

---

- *getALT*
`public java.lang.String` **getALT**`( )`
  - **Description**
    Returns the value of the "ALT" attribute.
  - **Returns** – The value of the "ALT" attribute.

---

- *getAttribute*
`public java.lang.Object` **getAttribute**`( java.lang.String` **name** `)`
  - **Description**
    Gets an attribute.
  - **Parameters**
    - `name` – a `String` which contains the name of the attribute

---

- *getAutoscaleInput*
`public int` **getAutoscaleInput**`( )`
  - **Description**
    Returns the value of the "AutoscaleInput" attribute.
  - **Returns** – the `int` value of the "AutoscaleInput" attribute.

---

- *getAutoscaleMinimumTimeInterval*
`public int` **getAutoscaleMinimumTimeInterval**`( )`

– **Description**
Returns the value of the “AutoscaleMinimumTimeInterval” attribute.
– **Returns** – The `int` value of the “AutoscaleMinimumTimeInterval” attribute.

---

• *getAutoscaleOutput*
`public int` **getAutoscaleOutput**`( )`

– **Description**
Returns the value of the “AutoscaleOutput” attribute.
– **Returns** – The `int` value of the “AutoscaleOutput” attribute.

---

• *getAxis*
`public Axis` **getAxis**`( )`

– **Description**
Returns the value of the “Axis” attribute.
– **Returns** – the `Axis` value of the “Axis” attribute

---

• *getBackground*
`public Background` **getBackground**`( )`

– **Description**
Returns the value of the “Background” attribute. This is the node used to draw the chart’s background.
– **Returns** – The `Background` value of the “Background” attribute, if defined. Otherwise, null is returned.

---

• *getBarGap*
`public double` **getBarGap**`( )`

– **Description**
Returns the value of the “BarGap” attribute.
– **Returns** – the `double` value of the “BarGap” attribute, if defined. Otherwise, 0.0 is returned.

---

• *getBarType*
`public int` **getBarType**`( )`

– **Description**
Returns the value of the “BarType” attribute.
– **Returns** – an `int` which specifies BarType

---

• *getBarWidth*
`public double` **getBarWidth**`( )`

– **Description**
Returns the value of the “BarWidth” attribute.

– **Returns** – the `double` value of the “BarWidth” attribute, if defined. Otherwise, 0.5 is returned.

---

- *getBooleanAttribute*
`public boolean` **getBooleanAttribute**`( java.lang.String` **name**`, boolean` **defaultValue** `)`

  – **Description**
  Convenience routine to get a Boolean-valued attribute.

  – **Parameters**
    * `name` – a `String` which contains the name of the attribute
    * `defaultValue` – the `boolean` default value of the attribute

  – **Returns** – the `boolean` value of the attribute, if defined and if its value is of type Boolean. Otherwise defaultValue is returned.

---

- *getChart*
`public Chart` **getChart**`( )`

  – **Description**
  Returns the value of the “Chart” attribute. This is the root node of the chart tree.

  – **Returns** – The `Chart` value of the attribute, if defined. Otherwise, null is returned.

---

- *getChartTitle*
`public ChartTitle` **getChartTitle**`( )`

  – **Description**
  Returns the value of the “ChartTitle” attribute.

  – **Returns** – the `ChartTitle` value of the attribute.

---

- *getChildren*
`public final ChartNode[]` **getChildren**`( )`

  – **Description**
  Returns an array of the children of this node. If there are no children, a 0-length array is returned.

  – **Returns** – a `ChartNode` array which contains the children of this node

---

- *getClipData*
`public boolean` **getClipData**`( )`

– **Description**
Returns the value of the "ClipData" attribute.
– **Returns** – The `boolean` value of the attribute, if defined. Otherwise, true is returned.

---

• *getColorAttribute*
`public java.awt.Color` **getColorAttribute**`( java.lang.String` **name** `)`

– **Description**
Convenience routine to get a Color-valued attribute.
– **Parameters**
∗ `name` – a `String` which contains the name of the attribute.
– **Returns** – the `Color` value of the attribute, if defined and if its value is of type Color. Otherwise, a default color value is returned.

---

• *getComponent*
`public java.awt.Component` **getComponent**`( )`

– **Description**
Returns the value of the "Component" attribute. This is the AWT object into which the chart is rendered.
– **Returns** – The `Component` value of the attribute, if defined. Otherwise, null is returned.

---

• *getConcatenatedViewport*
`public double[]` **getConcatenatedViewport**`( )`

– **Description**
Returns the value of the "Viewport" attribute concatenated with the "Viewport" attributes set in its ancestor nodes.
– **Returns** – a `double`[4] array containing xmin, xmax, ymin, ymax

---

• *getCustomTransform*
`public Transform` **getCustomTransform**`( )`

– **Description**
Returns the value of the "CustomTransform" attribute.
– **Returns** – an `Transform` which contains the value of the "Transform" attribute

---

• *getDataType*
`public int` **getDataType**`( )`

– **Description**
Returns the value of the "DataType" attribute.

– **Returns** – The `int` value of the "DataType" attribute, if defined. Otherwise, DATA_TYPE_LINE is returned.

---

- *getDensity*
  `public int` **getDensity**`( )`
  – **Description**
  Returns the value of the "Density" attribute.
  – **Returns** – The `int` value of the "Density" attribute, if defined. Otherwise, a default value of zero is returned.

---

- *getDoubleAttribute*
  `public double` **getDoubleAttribute**`( java.lang.String` **name**`, double` **defaultValue** `)`
  – **Description**
  Convenience routine to get a Double-valued attribute.
  – **Parameters**
    * `name` – a `String` which contains the name of the attribute
    * `defaultValue` – the `double` default value of the attribute.
  – **Returns** – the `double` value of the attribute, if defined and if its value is of type Double. Otherwise defaultValue is returned.

---

- *getDoubleBuffering*
  `public boolean` **getDoubleBuffering**`( )`
  – **Description**
  Returns the value of the "DoubleBuffering" attribute.
  – **Returns** – The `boolean` value of the "DoubleBuffering" attribute, if defined. Otherwise, false is returned.

---

- *getExplode*
  `public double` **getExplode**`( )`
  – **Description**
  Returns the value of the "Explode" attribute.
  – **Returns** – The `double` value of the "Explode" attribute, if defined. Otherwise, a default value of zero is returned. (The pie slice begins at the center.)

---

- *getFillColor*
  `public java.awt.Color` **getFillColor**`( )`
  – **Description**
  Returns the value of the "FillColor" attribute.

– **Returns** – The `Color` value of the "FillColor" attribute, if defined. Otherwise, a default color value is returned.

---

- *getFillOutlineColor*
  `public java.awt.Color` **getFillOutlineColor**`( )`
  – **Description**
  Returns the value of the "FillOutlineColor" attribute.
  – **Returns** – The `Color` value of the "FillOutlineColor" attribute, if defined. Otherwise, a default color value is returned.

---

- *getFillOutlineType*
  `public int` **getFillOutlineType**`( )`
  – **Description**
  Returns the value of the "FillOutlineType" attribute.
  – **Returns** – The `int` value of the "FillOutlineType" attribute, if defined. Otherwise, FILL_TYPE_SOLID is returned.

---

- *getFillPaint*
  `public java.awt.Paint` **getFillPaint**`( )`
  – **Description**
  Returns the value of the "FillPaint" attribute.
  – **Returns** – The value of the "FillPaint" attribute, if defined. Otherwise, null is returned.

---

- *getFillType*
  `public int` **getFillType**`( )`
  – **Description**
  Returns the value of the "FillType" attribute.
  – **Returns** – The `int` value of the "FillType" attribute, if defined. Otherwise, FILL_TYPE_SOLID is returned.

---

- *getFont*
  `public java.awt.Font` **getFont**`( )`
  – **Description**
  Convenience routine which gets a `Font` object based on the "FontName", "FontStyle" and "FontSize" attributes. There is *no* "Font" attribute.

---

- *getFontName*
  `public java.lang.String` **getFontName**`( )`

- **Description**
  Returns the value of the "FontName" attribute.
- **Returns** – The `String` value of the "FontName" attribute, if defined. Otherwise, the empty string is returned.

---

- *getFontSize*
  `public int` **getFontSize**`( )`
  - **Description**
    Returns the value of the "FontSize" attribute.
  - **Returns** – The `int` value of the "FontSize" attribute, if defined. Otherwise, 10 is returned.

---

- *getFontStyle*
  `public int` **getFontStyle**`( )`
  - **Description**
    Returns the value of the "FontStyle" attribute.
  - **Returns** – The `int` value of the "FontStyle" attribute, if defined. Otherwise, java.awt.Font.PLAIN is returned.

---

- *getGradient*
  `public java.awt.Color[]` **getGradient**`( )`
  - **Description**
    Returns the value of the "Gradient" attribute.
  - **Returns** – a `Color` array which contains the color value of the "Gradient" attribute, if defined. Otherwise, null is returned. The array is of length four, containing {colorLL, colorLR, colorUR, colorUL}.

---

- *getHREF*
  `public java.lang.String` **getHREF**`( )`
  - **Description**
    Returns the value of the "HREF" attribute.
  - **Returns** – The value of the "HREF" attribute.

---

- *getImage*
  `public java.awt.Image` **getImage**`( )`
  - **Description**
    Returns the value of the "Image" attribute.
  - **Returns** – the `Image` value of the "Image" attribute

---

- *getIntegerAttribute*
  `public int` **getIntegerAttribute**`( java.lang.String` **name**`, int` **defaultValue** `)`
  - **Description**
    Convenience routine to get an Integer-valued attribute.
  - **Parameters**
    * `name` – a `String` which contains the name of the attribute.
    * `defaultValue` – the `int` default value of the attribute
  - **Returns** – the `int` value of the attribute, if defined and if its value is of type Integer. Otherwise defaultValue is returned.

---

- *getLabelType*
  `public int` **getLabelType**`( )`
  - **Description**
    Returns the value of the "LabelType" attribute. If the attribute has not been set LABEL_TYPE_NONE is returned.
  - **Returns** – The `int` value of the "LabelType" attribute.

---

- *getLegend*
  `public Legend` **getLegend**`( )`
  - **Description**
    Returns the value of the "Legend" attribute.
  - **Returns** – the `Legend` value of the "Legend" attribute

---

- *getLineColor*
  `public java.awt.Color` **getLineColor**`( )`
  - **Description**
    Returns the value of the "LineColor" attribute.
  - **Returns** – The `LineColor` value of the "LineColor" attribute, if defined. Otherwise, a default color value is returned.

---

- *getLineDashPattern*
  `public double[]` **getLineDashPattern**`( )`
  - **Description**
    Returns the value of the "LineDashPattern" attribute.
  - **Returns** – `double` array containing the value of the "LineDashPattern" attribute, if defined. Otherwise, null is returned.

---

- *getLineWidth*
  `public double` **getLineWidth**`( )`

- **Description**
  Returns the value of the "LineWidth" attribute.
- **Returns** – The `double` value of the "LineWidth" attribute, if defined. Otherwise, the default value of one is returned.

---

- *getLocale*
  `public java.util.Locale` **getLocale**`( )`
  - **Description**
    Returns the value of the "Locale" attribute.
  - **Returns** – The `Locale` value of the "Locale" attribute, if defined. Otherwise, a default value is returned.

---

- *getMarkerColor*
  `public java.awt.Color` **getMarkerColor**`( )`
  - **Description**
    Returns the value of the "MarkerColor" attribute. Otherwise, a default color value is returned.
  - **Returns** – a `Color` which contains the "MarkerColor" value

---

- *getMarkerDashPattern*
  `public double[]` **getMarkerDashPattern**`( )`
  - **Description**
    Returns the value of the "MarkerPattern" attribute.
  - **Returns** – The `double` array which contains the value of the "MarkerPattern" attribute, if defined. Otherwise, null is returned.

---

- *getMarkerSize*
  `public double` **getMarkerSize**`( )`
  - **Description**
    Returns the value of the "MarkerSize" attribute.
  - **Returns** – The `double` value of the "MarkerSize" attribute, if defined. Otherwise, a default of 1.0 is returned.

---

- *getMarkerThickness*
  `public double` **getMarkerThickness**`( )`
  - **Description**
    Returns the value of the "MarkerThickness" attribute.
  - **Returns** – The `double` value of the "MarkerThickness" attribute, if defined. Otherwise, a default of 1.0 is returned.

- *getMarkerType*
  `public int` **getMarkerType**`( )`

  - **Description**
    Returns the value of the "MarkerType" attribute.
  - **Returns** – The `int` value of the "MarkerType" attribute, if defined. Otherwise, a default of MARKER_TYPE_PLUS is returned.

- *getName*
  `public java.lang.String` **getName**`( )`

  - **Description**
    Returns the value of the "Name" attribute.
  - **Returns** – The `String` value of the "Name" attribute, if defined. Otherwise, the empty string is returned.

- *getNumber*
  `public int` **getNumber**`( )`

  - **Description**
    Returns the value of the "Number" attribute.
  - **Returns** – The `int` value of the "Number" attribute, if defined. Otherwise, zero is returned.

- *getPaint*
  `public boolean` **getPaint**`( )`

  - **Description**
    Returns the value of the "Paint" attribute.
  - **Returns** – The `boolean` value of the "Paint" attribute, if defined. Otherwise, true is returned.

- *getParent*
  `public ChartNode` **getParent**`( )`

  - **Description**
    Returns the parent of this node. Note that this is *not* an attribute setting. Note that there is no setParent function.
  - **Returns** – A `ChartNode` object which contains this node's parent. This is null in the case of the root node of the chart tree, since that node has no parent.

- *getReference*
  `public double` **getReference**`( )`

- **Description**
  Returns the value of the "Reference" attribute.
- **Returns** – The `double` value of the "Reference" attribute, if defined. Otherwise, zero is returned.

---

- *getScreenAxis*
  `public AxisXY` **getScreenAxis**`( )`
  - **Description**
    Returns the value of the "ScreenAxis" attribute. This provides a default mapping from the user coordinates [0,1] by [0,1] to the screen. This is set by the root Chart node, so there is no setScreenAxis function.
  - **Returns** – The `AxisXY` value of the "ScreenAxis" attribute

---

- *getScreenSize*
  `public java.awt.Dimension` **getScreenSize**`( )`
  - **Description**
    Returns the value of the "ScreenSize" attribute.
  - **Returns** – The `Dimension` value of the "ScreenSize" attribute, if defined. Otherwise, the size of the "Component" attribute is returned. If neither the "ScreenSize" nor the "Component" attributes are defined then null is returned.

---

- *getScreenViewport*
  `public int[]` **getScreenViewport**`( )`
  - **Description**
    Returns the value of the "Viewport" attribute scaled by the screen size.
  - **Returns** – the `int`[4] value of the "Viewport" attribute scaled by the screen size containing the pixel coordinates for xmin, xmax, ymin, ymax

---

- *getSize*
  `public java.awt.Dimension` **getSize**`( )`
  - **Description**
    Returns the value of the "Size" attribute.
  - **Returns** – the `Dimension` value of the "Size" attribute

---

- *getSkipWeekends*
  `public boolean` **getSkipWeekends**`( )`
  - **Description**
    Returns the value of the "SkipWeekends" attribute. If true then autoscaling will not select an interval of less than a day.

– **Returns** – the value of the "SkipWeekend" attribute..

---

- *getStringAttribute*
  `public java.lang.String` **getStringAttribute**`( java.lang.String` **name** `)`

  – **Description**
  Convenience routine to get a String-valued attribute.

  – **Parameters**

  * `name` – a `String` which contains the name of the attribute.

  – **Returns** – the `String` value of the attribute, if defined and if its value is of type String.

---

- *getTextAngle*
  `public int` **getTextAngle**`( )`

  – **Description**
  Returns the value of the "TextAngle" attribute.

  – **Returns** – The `int` value of the "TextAngle" attribute, if defined. Otherwise, zero is returned.

---

- *getTextColor*
  `public java.awt.Color` **getTextColor**`( )`

  – **Description**
  Returns the value of the "TextColor" attribute.

  – **Returns** – The `Color` value of the "TextColor" attribute, if defined. Otherwise, a default color value is returned.

---

- *getTextFormat*
  `public java.text.Format` **getTextFormat**`( )`

  – **Description**
  Returns the value of the "TextFormat" attribute.

  – **Returns** – The `Format` value of the "TextFormat" attribute, if defined. Otherwise, a default format is returned. The default is a NumberFomat that allows exactly two digits after the decimal.

---

- *getTickLength*
  `public double` **getTickLength**`( )`

  – **Description**
  Returns the value of the "TickLength" attribute.

  – **Returns** – The `double` value of the "TickLength" attribute, if defined. Otherwise, 1.0 is returned.

- *getTitle*
`public Text` **getTitle**`( )`
  - **Description**
    Returns the value of the "Title" attribute.
  - **Returns** – the `Text` value of the "Title" attribute

- *getToolTip*
`public java.lang.String` **getToolTip**`( )`
  - **Description**
    Returns the value of the "ToolTip" attribute.
  - **Returns** – the `String` value of the "ToolTip" attribute

- *getTransform*
`public int` **getTransform**`( )`
  - **Description**
    Returns the value of the "Transform" attribute.
  - **Returns** – an `int` which contains the value of the "Transform" attribute

- *getViewport*
`public double[]` **getViewport**`( )`
  - **Description**
    Returns the value of the "Viewport" attribute.
  - **Returns** – a `double`[4] array containing xmin, xmax, ymin, ymax

- *getX*
`public double[]` **getX**`( )`
  - **Description**
    Returns the value of the "X" attribute.
  - **Returns** – the `double` array which contains the value of the "X" attribute

- *getY*
`public double[]` **getY**`( )`
  - **Description**
    Returns the value of the "Y" attribute.
  - **Returns** – the `double` array which contains the value of the "Y" attribute

- *isAncestorOf*
`public boolean` **isAncestorOf**`( ChartNode` **node** `)`

– **Description**
Returns true if this node is an ancestor of the argument node.

– **Parameters**

 * `node` – a `ChartNode` object

– **Returns** – a `boolean`, true if this node is an ancestor of the argument, node

---

- *isAttributeSet*
`public boolean` **isAttributeSet**`( java.lang.String` **name** `)`

  – **Description**
  Determines if an attribute is defined (may have been inherited).

  – **Parameters**

   * `name` – a `String` which contains the name of the attribute

  – **Returns** – a `boolean`, true if the attribute is defined for this node. The definition may have been inherited from its parent node.

---

- *isAttributeSetAtThisNode*
`public boolean` **isAttributeSetAtThisNode**`( java.lang.String` **name** `)`

  – **Description**
  Determines if an attribute is defined in this node (not inherited).

  – **Parameters**

   * `name` – a `String` which contains the name of the attribute

  – **Returns** – a `boolean`, true if the attribute is defined in this node. The definition must have been set directly in this node, not just inherited from its parent node.

---

- *isBitSet*
`public static boolean` **isBitSet**`( int` **flag**`, int` **mask** `)`

  – **Description**
  Returns true if the bit set in flag is set in mask.

  – **Parameters**

   * `flag` – the `int` which contains the bit to be tested against mask
   * `mask` – the `int` which is used as the mask

  – **Returns** – a `boolean`, true if the bit set in flag is set in mask

---

- *paint*
`public abstract void` **paint**`( Draw` **draw** `)`

  – **Description**
  Paints this node and all of its children.

  – **Parameters**

* `draw` – the `Draw` object to be painted

---

- *parseColor*
  `public static java.awt.Color` **parseColor**`( java.lang.String` **nameColor** `)`

  - **Description**
    Returns a color specified by name or a red-green-blue triple.
  - **Parameters**
    * `nameColor` – is the name of a color (this name is not case sensitive) or a comma separated list of red, green, blue values all in the range 0 to 255. For example, "red" or "255,0,0".
  - **Returns** – the named `Color`.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if the color name is not known.

---

- *remove*
  `public final void` **remove**`( )`

  - **Description**
    Removes the node from its parents list of children.

---

- *removePickListener*
  `public void` **removePickListener**`( PickListener` **pickListener** `)`

  - **Description**
    Removes a `PickListener` from this node.
  - **Parameters**
    * `pickListener` – the `PickListener` to be removed from this node

---

- *setALT*
  `public void` **setALT**`( java.lang.String` **value** `)`

  - **Description**
    Sets the value of the "ALT" attribute. The "ALT" attribute is used when client-side image maps are generated. A client-side image map has an entry for each node in which the chart attribute `HREF` is defined. Some browsers use the alt tag value as tooltip text. *
  - **Parameters**
    * `value` – "ALT" value.

---

- *setAttribute*
  `public void` **setAttribute**`( java.lang.String` **name**`, java.lang.Object` **value** `)`
  - **Description**
    Sets an attribute.
  - **Parameters**
    * `name` – a `String` which contains the name of the attribute to be set
    * `value` – an `Object` which contains the value of the attribute

---

- *setAutoscaleInput*
  `public void` **setAutoscaleInput**`( int` **value** `)`
  - **Description**
    Sets the value of the "AutoscaleInput" attribute. This attribute determines what inputs are use for autoscaling.
  - **Parameters**
    * `value` – "AutoscaleInput" value. Legal values are

| | |
|---|---|
| AUTOSCALE_OFF | Do not do autoscaling. |
| AUTOSCALE_DATA | Use the data values. This is the default. |
| AUTOSCALE_WINDOW | Use the "Window" attribute value. |

---

- *setAutoscaleMinimumTimeInterval*
  `public void` **setAutoscaleMinimumTimeInterval**`( int` **value** `)`
  - **Description**
    Sets the value of the "AutoscaleMinimumTimeInterval" attribute. This attribute determines the minimum tick mark interval for autoscaled time axes.
  - **Parameters**
    * `value` – "AutoscaleMinimumTimeInterval" value. Legal values are:

| | |
|---|---|
| MILLISECOND | Millisecond |
| SECOND | Second |
| MINUTE | Minute |
| HOUR_OF_DAY | Hour |
| DAY_OF_WEEK | Day |
| WEEK_OF_YEAR | Week |
| MONTH | Month |
| YEAR | Year |

The default is MILLISECOND.

---

- *setAutoscaleOutput*
  `public void` **setAutoscaleOutput**`( int` **value** `)`

– **Description**
Sets the value of the "AutoscaleOutput" attribute. This attribute determines what attributes to change as a result of autoscaling.

– **Parameters**

* `value` – "AutoscaleOutput" value. Legal values are bitwise-or combinations of the following:

| | |
|---|---|
| AUTOSCALE_OFF | Do not do autoscaling. |
| AUTOSCALE_WINDOW | Change the "Window" attribute value. |
| AUTOSCALE_NUMBER | Change the "Number" attribute value. |
| AUTOSCALE_DENSITY | Change the "Density" attribute value. |

The default is (AUTOSCALE_NUMBER `|` AUTOSCALE_WINDOW `|` AUTOSCALE_DENSITY).

---

• *setBarGap*
`public void` **setBarGap**`( double` **value** `)`

– **Description**
Sets the value of the "BarGap" attribute. This is the gap between bars in a group. A gap of 1.0 means that space between bars is the same as the width of an individual bar in the group.

– **Parameters**

* `value` – the `double` "BarGap" value

---

• *setBarType*
`public void` **setBarType**`( int` **value** `)`

– **Description**
Sets the value of the "BarType" attribute.

– **Parameters**

* `value` – an `int` which specifies BarType. Legal values are BAR_TYPE_VERTICAL or BAR_TYPE_HORIZONTAL.

---

• *setBarWidth*
`public void` **setBarWidth**`( double` **value** `)`

– **Description**
Sets the value of the "BarWidth" attribute. This is the width of all of the groups of bars at each index.

– **Parameters**

* `value` – the `double` "BarWidth" value.

- *setChartTitle*
  `public void` **setChartTitle**`( ChartTitle` **value** `)`

  – **Description**
    Sets the value of the "ChartTitle" attribute. This is effective only in the Chart node, where it replaces the existing ChartTitle node. The Chart node constructor creates a ChartTitle node and uses it to define its "ChartTitle" attribute, so there is generally no need to call this routine.
  – **Parameters**
    * `value` – `ChartTitle` node

- *setClipData*
  `public void` **setClipData**`( boolean` **value** `)`

  – **Description**
    Sets the value of the "ClipData" attribute. This indicates that the data elements are to be clipped to the current window.
  – **Parameters**
    * `value` – "ClipData" value

- *setCustomTransform*
  `public void` **setCustomTransform**`( Transform` **value** `)`

  – **Description**
    Sets the value of the "CustomTransform" attribute. This is used only if the "Transform" attribute is set to TRANSFORM_CUSTOM.
  – **Parameters**
    * `value` – an object implementing the Transform interface.

- *setDataType*
  `public void` **setDataType**`( int` **value** `)`

  – **Description**
    Sets the value of the "DataType" attribute.
  – **Parameters**
    * `value` – "DataType" value. This should be some xor-ed combination of DATA_TYPE_LINE, DATA_TYPE_MARKER, DATA_TYPE_FILL, DATA_TYPE_ERROR_X, DATA_TYPE_ERROR_Y, and DATA_TYPE_ERROR_PICTURE.

- *setDensity*
  `public void` **setDensity**`( int` **value** `)`

– **Description**
  Sets the value of the "Density" attribute. This attribute controls the number of minor tick marks in the interval between major tick marks.
– **Parameters**
  * `value` – `int` "Density" value which specifies the number of minor tick marks per major tick mark.

---

- *setDoubleBuffering*
  `public void` **setDoubleBuffering**`( boolean` **value** `)`

  – **Description**
    Sets the value of the "DoubleBuffering" attribute. Double buffering reduces flicker when the screen is updated. This attribute only has an effect if it is set at the root node of the chart tree.
  – **Parameters**
    * `value` – `boolean` "DoubleBuffering" value

---

- *setExplode*
  `public void` **setExplode**`( double` **value** `)`

  – **Description**
    Sets the value of the "Explode" attribute. This attribute controls how far from the center pie slices are drawn. The scale is proportional to the pie chart's radius.
  – **Parameters**
    * `value` – a `double` "Explode" value. This attribute controls how far from the center pie slices are drawn. The scale is proportional to the pie chart's radius.

---

- *setFillColor*
  `public void` **setFillColor**`( java.awt.Color` **color** `)`

  – **Description**
    Sets the value of the "FillColor" attribute.
  – **Parameters**
    * `color` – `Color` "FillColor" value

---

- *setFillColor*
  `public void` **setFillColor**`( java.lang.String` **color** `)`

  – **Description**
    Sets the "FillColor" attribute to a color specified by name.
  – **Parameters**

* `color` – `String` name of a color.

---

- *setFillOutlineColor*
  `public void` **setFillOutlineColor**`( java.awt.Color` **color** `)`
  - **Description**
    Sets the value of the "FillOutlineColor" attribute.
  - **Parameters**
    * `color` – a `Color` "FillOutlineColor" value.

---

- *setFillOutlineColor*
  `public void` **setFillOutlineColor**`( java.lang.String` **color** `)`
  - **Description**
    Sets the value of the "FillOutlineColor" attribute to a color specified by name.
  - **Parameters**
    * `color` – `String` name of a color.

---

- *setFillOutlineType*
  `public void` **setFillOutlineType**`( int` **value** `)`
  - **Description**
    Sets the value of the "FillOutlineType" attribute.
  - **Parameters**
    * `value` – "FillOutlineType" value. This value should be FILL_TYPE_NONE or FILL_TYPE_SOLID.

---

- *setFillPaint*
  `public void` **setFillPaint**`( javax.swing.ImageIcon` **imageIcon** `)`
  - **Description**
    Sets the value of the "FillPaint" attribute.
  - **Parameters**
    * `imageIcon` – is used to create a Paint object that is used as the value of the "FillPaint" attribute.

---

- *setFillPaint*
  `public void` **setFillPaint**`( java.awt.Paint` **value** `)`
  - **Description**
    Sets the value of the "FillPaint" attribute.
  - **Parameters**
    * `value` – "FillPaint" value.

---

- *setFillPaint*
  `public void` **setFillPaint**`( java.net.URL` **urlImage** `)`
  - **Description**
    Sets the value of the "FillPaint" attribute.
  - **Parameters**
    * `urlImage` – is the URL of an image used to set the FillPaint attribute.

---

- *setFillType*
  `public void` **setFillType**`( int` **value** `)`
  - **Description**
    Sets the value of the "FillType" attribute.
  - **Parameters**
    * `value` – "FillType" value. This value should be FILL_TYPE_NONE, FILL_TYPE_SOLID, FILL_TYPE_GRADIENT or FILL_TYPE_PAINT.

---

- *setFont*
  `public void` **setFont**`( java.awt.Font` **font** `)`
  - **Description**
    Sets the value of the font attributes. This function sets the "FontName", "FontStyle" and "FontSize" attributes. There is *no* "Font" attribute.
  - **Parameters**
    * `font` – `Font` object whose components are used to set three different attributes.

---

- *setFontName*
  `public void` **setFontName**`( java.lang.String` **value** `)`
  - **Description**
    Sets the value of the "FontName" attribute. This is used in the constructor for java.awt.Font.
  - **Parameters**
    * `value` – a `String` which contains the "FontName" value

---

- *setFontSize*
  `public void` **setFontSize**`( int` **value** `)`
  - **Description**
    Sets the value of the "FontSize" attribute. This is used in the constructor for java.awt.Font.
  - **Parameters**

* `value` – an `int` “FontSize” value

---

- *setFontStyle*
  `public void` **setFontStyle**`( int` **value** `)`
  - **Description**
    Sets the value of the “FontStyle” attribute. This is used in the constructor for java.awt.Font.
  - **Parameters**
    * `value` – an `int` “FontStyle” value.

---

- *setGradient*
  `public void` **setGradient**`( java.awt.Color[]` **colorGradient** `)`
  - **Description**
    Sets the value of the “Gradient” attribute.
  - **Parameters**
    * `colorGradient` – is a `Color` array of length four, containing the colors at the lower left, lower right, upper right and upper left corners of the bounding box of the regions being filled. See setGradient(java.awt.Color,java.awt.Color,java.awt.Color,java.awt.Color) for details on the interpretation of these colors.

---

- *setGradient*
  `public void` **setGradient**`( java.awt.Color` **colorLL**`, java.awt.Color` **colorLR**`, java.awt.Color` **colorUR**`, java.awt.Color` **colorUL** `)`
  - **Description**
    Sets the value of the “Gradient” attribute.
  - **Parameters**
    * `colorLL` – `Color` value which specifies the color of the lower left corner.
    * `colorLR` – `Color` value which specifies the color of the lower right corner.
    * `colorUR` – `Color` value which specifies the color of the upper right corner.
    * `colorUL` – `Color` value which specifies the color of the upper left corner.
      This attribute defines a color gradient used to fill regions. Only two of the four colors given are actually used.
      If `colorLL==colorLR` and `colorUL==colorUR` then a vertical gradient is drawn.
      If `colorLL==colorUL` and `colorLR==colorUR` then a horizontal gradient is drawn.
      If `colorLR==null` and `colorUL==null` then a diagonal gradient is used.
      If `colorLL==null` and `colorUR==null` then a diagonal gradient is used.
      If none of these conditions is met then no gradient is drawn.

- *setGradient*
  `public void` **setGradient**`( java.lang.String` **colorLL**`, java.lang.String` **colorLR**`, java.lang.String` **colorUR**`, java.lang.String` **colorUL** `)`

  – **Description**
    Sets the value of the "Gradient" attribute using named colors.
  – **Parameters**
    * `colorLL` – `String` value which specifies the color of the lower left corner.
    * `colorLR` – `String` value which specifies the color of the lower right corner.
    * `colorUR` – `String` value which specifies the color of the upper right corner.
    * `colorUL` – `String` value which specifies the color of the upper left corner.
      This attribute defines a color gradient used to fill regions. Only two of the four colors given are actually used.
      If `colorLL==colorLR` and `colorUL==colorUR` then a vertical gradient is drawn.
      If `colorLL==colorUL` and `colorLR==colorUR` then a horizontal gradient is drawn.
      If `colorLR==null` and `colorUL==null` then a diagonal gradient is used.
      If `colorLL==null` and `colorUR==null` then a diagonal gradient is used.
      If none of these conditions is met then no gradient is drawn.

- *setHREF*
  `public void` **setHREF**`( java.lang.String` **value** `)`

  – **Description**
    Sets the value of the "HREF" attribute. The "HREF" attribute is used when client-side image maps are generated. A client-side image map has an entry for each node in which the chart attribute `HREF` is defined. The values of `HREF` attributes are URLs. Such regions treated by the browser as hyperlinks.
  – **Parameters**
    * `value` – "HREF" value.

- *setImage*
  `public void` **setImage**`( java.awt.Image` **value** `)`

  – **Description**
    Sets the value of the "Image" attribute. This function also loads the image, if necessary, using the java.awt.MediaTracker class. The component associated with this chart is redrawn after the image is loaded by MediaTracker.
    Note that `Image` objects are not serializable and their presence in the chart tree will make the entire chart non-serializable. javax.swing.ImageIcon objects are serializable.

– **Parameters**
  * `value` – `Image` value.

---

- *setImage*
  `public void` **setImage**`( javax.swing.ImageIcon` **value** `)`
  – **Description**
  Sets the value of the "Image" attribute.
  – **Parameters**
    * `value` – `ImageIcon` value.

---

- *setLabelType*
  `public void` **setLabelType**`( int` **type** `)`
  – **Description**
  Sets the value of the "LabelType" attribute. This indicates how a data point is to be labeled. The default is to not label data points.
  – **Parameters**
    * `type` – the `int` "LabelType" value

---

- *setLineColor*
  `public void` **setLineColor**`( java.awt.Color` **color** `)`
  – **Description**
  Sets the value of the "LineColor" attribute.
  – **Parameters**
    * `color` – the `LineColor` value

---

- *setLineColor*
  `public void` **setLineColor**`( java.lang.String` **color** `)`
  – **Description**
  Sets the value of the "LineColor" attribute.
  – **Parameters**
    * `color` – the `LineColor` value

---

- *setLineDashPattern*
  `public void` **setLineDashPattern**`( double[]` **value** `)`
  – **Description**
  Sets the value of the "LineDashPattern" attribute.
  – **Parameters**
    * `value` – `double` "LineDashPattern" value.

---

- *setLineWidth*
  `public void` **setLineWidth**`( double` **value** `)`
  - **Description**
    Sets the value of the “LineWidth” attribute.
  - **Parameters**
    * `value` – the `double` “LineWidth” value

---

- *setLocale*
  `public void` **setLocale**`( java.util.Locale` **value** `)`
  - **Description**
    Sets the value of the “Locale” attribute. This attribute controls how formatting is done.
  - **Parameters**
    * `value` – the `Locale` value

---

- *setMarkerColor*
  `public void` **setMarkerColor**`( java.awt.Color` **color** `)`
  - **Description**
    Sets the value of the “MarkerColor” attribute.
  - **Parameters**
    * `color` – a `Color` which contains the “MarkerColor” value

---

- *setMarkerColor*
  `public void` **setMarkerColor**`( java.lang.String` **color** `)`
  - **Description**
    Sets the value of the “MarkerColor” attribute to a color specified by name.
  - **Parameters**
    * `color` – `String` name of a color.

---

- *setMarkerDashPattern*
  `public void` **setMarkerDashPattern**`( double[]` **value** `)`
  - **Description**
    Sets the value of the “MarkerDashPattern” attribute.
  - **Parameters**
    * `value` – `double` array which contains the “MarkerDashPattern” value.

---

- *setMarkerSize*
  `public void` **setMarkerSize**`( double` **size** `)`

- **Description**
  Sets the value of the "MarkerSize" attribute. The default marker size is 1.0. If "MarkerSize" is 2.0 then markers are drawn twice as large as normal.
- **Parameters**
  * **`size`** – a `double` which specifies the "MarkerSize" value

---

- *setMarkerThickness*
  `public void` **setMarkerThickness**`( double` **width** `)`

  - **Description**
    Sets the value of the "MarkerThickness" attribute. This determines the line thickness used to draw the markers. The default marker width is 1.0. If "MarkerThickness" is 2.0 then markers are drawn twice as thick as normal.
  - **Parameters**
    * **`width`** – the `double` "MarkerThickness" value.

---

- *setMarkerType*
  `public void` **setMarkerType**`( int` **type** `)`

  - **Description**
    Sets the value of the "MarkerType" attribute. This indicates which marker is to be drawn.
  - **Parameters**
    * **`type`** – the `int` "MarkerType" value.

---

- *setName*
  `public void` **setName**`( java.lang.String` **value** `)`

  - **Description**
    Sets the value of the "Name" attribute. This the user-friendly name of the node.
  - **Parameters**
    * **`value`** – a `String` which contains the "Name" value

---

- *setNumber*
  `public void` **setNumber**`( int` **value** `)`

  - **Description**
    Sets the value of the "Number" attribute. This is the number of tick marks along an axis.
  - **Parameters**
    * **`value`** – the `int` "Number" value

---

- *setPaint*
  `public void` **setPaint**`( boolean` **value** `)`
  - **Description**
    Sets the value of the “Paint” attribute.
  - **Parameters**
    * `value` – the `boolean` “Paint” value. If false, this node and its children are not drawn.

---

- *setReference*
  `public void` **setReference**`( double` **value** `)`
  - **Description**
    Sets the value of the “Reference” attribute. This is used as the baseline in drawing area charts. It is also used as the angle (in degrees) of the first slice in a pie chart.
  - **Parameters**
    * `value` – the `double` “Reference” value

---

- *setScreenSize*
  `public void` **setScreenSize**`( java.awt.Dimension` **value** `)`
  - **Description**
    Sets the value of the “ScreenSize” attribute.
  - **Parameters**
    * `value` – the `Dimension` “ScreenSize” value.

---

- *setSize*
  `public void` **setSize**`( java.awt.Dimension` **value** `)`
  - **Description**
    Sets the value of the “Size” attribute.
  - **Parameters**
    * `value` – the `Dimension` “Size” value

---

- *setSkipWeekends*
  `public void` **setSkipWeekends**`( boolean` **skipWeekends** `)`
  - **Description**
    Sets the value of the “SkipWeekends” attribute. If this attribute is true and weekends are skipped on date axes. (A date axis is an Axis1D whose AxisLabel has a TextFormat value that extends java.text.DateFormat.)
    If this attribute is set to true, the attribute “AutoscaleMinimumTimeInterval” should also be set to value of a day or longer.

– **Parameters**
  * `skipWeekends` – the `boolean` value.

---

• *setTextAngle*
`public void` **setTextAngle**`( int` **value** `)`

– **Description**
  Sets the value of the "TextAngle" attribute. This indicates the angle, in degrees, at which text is to be drawn. Only multiples of 90 are allowed at this time.
– **Parameters**
  * `value` – an `int` "TextAngle" value

---

• *setTextColor*
`public void` **setTextColor**`( java.awt.Color` **color** `)`

– **Description**
  Sets the value of the "TextColor" attribute.
– **Parameters**
  * `color` – a `Color` which contains the "TextColor" value

---

• *setTextColor*
`public void` **setTextColor**`( java.lang.String` **color** `)`

– **Description**
  Sets the value of the "TextColor" attribute to a color specified by name.
– **Parameters**
  * `color` – `String` name of a color.

---

• *setTextFormat*
`public void` **setTextFormat**`( java.text.Format` **value** `)`

– **Description**
  Sets the value of the "TextFormat" attribute.
– **Parameters**
  * `value` – a `Format` which contains the "TextFormat" value

---

• *setTextFormat*
`public void` **setTextFormat**`( java.lang.String` **value** `)`

– **Description**
  Sets the value of the "TextFormat" attribute.

The TextFormat attribute is normally a java.text.Format object, but, as a convenience, it can be set as a String. The following special values are defined. In this table, "locale" is the value of the locale attribute.

| value | Attribute |
|---|---|
| "Date(SHORT)" | DateFormat.getDateInstance( DateFormat.SHORT, locale) |
| "Date(MEDIUM)" | DateFormat.getDateInstance( DateFormat.MEDIUM, locale) |
| "Date(LONG)" | DateFormat.getDateInstance( DateFormat.LONG, locale) |
| "Currency" | DateFormat.getCurrencyInstance(locale) |
| "Number" | DateFormat.getNumberInstance(locale) |
| "Percent" | DateFormat.getPercentInstance(locale) |

If the value does not match one of these special cases then an interpretation as a java.text.DecimalFormat object is attempted. If this fails then an interpretation as a java.text.SimpleDateFormat object is attempted.

- **Parameters**
  - `value` – a `String` which contains the "TextFormat" value

---

- *setTickLength*
  `public void` **setTickLength**`( double` **value** `)`

- **Description**
  Sets the value of the "TickLength" attribute. This scales the length of the tick mark lines. A value of 2.0 makes the tick marks twice as long as normal. A negative value causes the tick marks to be drawn pointing into the plot area.
- **Parameters**
  - `value` – a `double` which contains the "TickLength" value

---

- *setTitle*
  `public void` **setTitle**`( java.lang.String` **value** `)`

- **Description**
  Sets the value of the "Title" attribute.
- **Parameters**
  - `value` – a `String` which contains the "Title" value

---

- *setTitle*
  `public void` **setTitle**`( Text` **value** `)`

- **Description**
  Sets the value of the "Title" attribute.
- **Parameters**

* value – a Text which contains the "Title" value

---

- *setToolTip*
  public void **setToolTip**( java.lang.String **value** )

  - **Description**
    Sets the value of the "ToolTip" attribute.
  - **Parameters**
    * value – a String which contains the "ToolTip" value

---

- *setTransform*
  public void **setTransform**( int **value** )

  - **Description**
    Sets the value of the "Transform" attribute. This sets the axis to be linear, logarithmic or a custom transform.
  - **Parameters**
    * value – The "Transform" value. Legal values are TRANSFORM_LINEAR (the default), TRANSFORM_LOG and TRANSFORM_CUSTOM.

---

- *setViewport*
  public void **setViewport**( double[] **value** )

  - **Description**
    Sets the value of the "Viewport" attribute. The viewport is the subregion of the drawing surface where the plot is to be drawn. "Viewport" coordinates are [0,1] by [0,1] with (0,0) in the lower left corner. This attribute affects only Axis nodes, since they contain the mappings to device space.
  - **Parameters**
    * value – A double array of length 4 which contains the "Viewport" values for xmin, xmax, ymin, ymax. The value saved is a copy of the input array.

---

- *setViewport*
  public void **setViewport**( double **xmin**, double **xmax**, double **ymin**, double **ymax** )

  - **Description**
    Sets the value of the "Viewport" attribute.
  - **Parameters**
    * xmin – a double, the left side of the viewport
    * xmax – a double, the right side of the viewport
    * ymin – a double, the bottom side of the viewport
    * ymax – a double, the top side of the viewport

---

- *setX*
  `public void` **setX**`( java.lang.Object` **value** `)`

  – **Description**
    Sets the value of the "X" attribute.
  – **Parameters**
    * `value` – an `Object` which contains the "X" value

---

- *setY*
  `public void` **setY**`( java.lang.Object` **value** `)`

  – **Description**
    Sets the value of the "Y" attribute.
  – **Parameters**
    * `value` – the `Object` which contains the "Y" value

---

- *toString*
  `public java.lang.String` **toString**`( )`

  – **Description**
    Returns the name of this `ChartNode`
  – **Returns** – a `String`, the name of this `ChartNode`

## *class* Background

The background of a chart.

Grid is created by com.imsl.chart.Chart as its child. It can be retrieved using the method getBackground().

Fill attributes in this node control the drawing of the background.

### Declaration

| |
|---|
| public class com.imsl.chart.Background<br>**extends** com.imsl.chart.AxisXY (page 962) |

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paint this node. This is not normally called by a user program.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* ChartTitle

The main title of a chart.

ChartTitle is created by com.imsl.chart.Chart as its child. It can be retrieved using the method getChartTitle().

The chart title is the value of the "Title" attribute at this node. Text attributes in this node control the drawing of the title.

### Declaration

public class com.imsl.chart.ChartTitle
**extends** com.imsl.chart.AxisXY (page 962)

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* Legend

The chart legend.

Legend is created by com.imsl.chart.Chart as its child. It can be retrieved using the method getLegend().

By default the legend is not drawn. To have it drawn, set its “Paint” attribute to true.

com.imsl.chart.Data objects that have their “Title” attribute defined are automatically entered into the legend.

The drawing of the background of the legend box is controlled by the fill attributes in this node. Text attributes control the drawing of the text strings in the box.

### Declaration

| |
|---|
| public class com.imsl.chart.Legend<br>**extends** com.imsl.chart.AxisXY (page 962) |

### Constructor

---

- *Legend*
  `protected` **Legend**`( Chart` **chart** `)`

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* Grid

Draws the grid lines perpendicular to an axis.

Grid is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getGrid().

Line attributes in this node control the drawing of the grid lines.

### Declaration

| |
|---|
| public class com.imsl.chart.Grid<br>**extends** com.imsl.chart.ChartNode (page 920) |

### Methods

---

- *getType*
  `public int` **getType**`( )`
  - **Description**
    Returns the axis type.
  - **Returns** – an `int`, the axis type

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* **Axis**

The Axis node provides the mapping for all of its children from the user coordinate space to the device (screen) space.

### Declaration

| |
|---|
| public abstract class com.imsl.chart.Axis<br>**extends** com.imsl.chart.ChartNode (page 920) |

## Constructor

- *Axis*
  `public` **Axis**`(` `Chart` **chart** `)`
  - **Description**
    Contructs an `Axis` node. Its parent must be a `Chart` node. This node's "Axis" attribute has itself as a value, so that decendent nodes can easily obtain their controlling axis node.
  - **Parameters**
    * `chart` – a `Chart` object, the parent of this node

## Methods

- *mapDeviceToUser*
  `public abstract void` **mapDeviceToUser**`(` `int` **devX**, `int` **devY**, `double[]` **userXY** `)`
  - **Description**
    Maps the device coordinates to user coordinates.
  - **Parameters**
    * `devX` – an `int` which specifies the device x-coordinate
    * `devY` – an `int` which specifies the device y-coordinate
    * `userXY` – an `int`[2] array on input, on output, the user coordinates

---

- *mapUserToDevice*
  `public abstract void` **mapUserToDevice**`(` `double` **userX**, `double` **userY**, `int[]` **devXY** `)`
  - **Description**
    Maps the user coordinates (userX,userY) to the device coordinates devXY.
  - **Parameters**
    * `userX` – a `double` which specifies the user x-coordinate
    * `userY` – a `double` which specifies the user y-coordinate
    * `devXY` – an `int`[2] array on input, on output, the device coordinates

---

- *paint*
  `public void` **paint**`(` `Draw` **draw** `)`

– **Description**
  Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
– **Parameters**
  * `draw` – a `Draw` object which specifies the chart tree to be rendered on the screen

---

- *setupMapping*
  `public abstract void` **setupMapping**`( )`
  – **Description**
    Initializes the mappings between user and coordinate space. This must be called whenever the screen size, the window or the viewport may have changed. Generally, it is safest to call this each time the chart is repainted.

## *class* **AxisXY**

The axes for an x-y chart.

This node is used when the mapping to and from user and device space can be decomposed into an x and a y mapping. This is when the mapping map(userX,userY) = (deviceX,deviceY) can be written as map(userX,userY) = (mapX(userX),mapY(userY) = (deviceX,deviceY)

### Declaration

public class com.imsl.chart.AxisXY
**extends** com.imsl.chart.Axis (page 960)

### Constructor

---

- *AxisXY*
  `public` **AxisXY**`( Chart` **chart** `)`
  – **Description**
    Create an `AxisXY`. This also creates two `Axis1D` nodes as children of this node. They hold the decomposed mapping. The "Viewport" attributute for this node is set to [0.2,0.8] by [0.2,0.8].
  – **Parameters**
    * `chart` – the `Chart` parent of this node

### Methods

- *getAxisX*
  `public Axis1D` **getAxisX**`( )`
  - **Description**
    Return the x-axis node.
  - **Returns** – the `Axis1D` x-axis node

---

- *getAxisY*
  `public Axis1D` **getAxisY**`( )`
  - **Description**
    Return the y-axis node.
  - **Returns** – the `Axis1D` y-axis node

---

- *getCross*
  `public double[]` **getCross**`( )`
  - **Description**
    Returns the value of the "Cross" attribute.
  - **Returns** – a `double`[2] array containing the value of the "Cross" attribute, if defined. The value is the point where the X and Y axes intersect, (xcross,ycross). If "Cross" is not defined then `null` is returned.

---

- *mapDeviceToUser*
  `public void` **mapDeviceToUser**`( int` **devX**`, int` **devY**`, double[]` **userXY** `)`
  - **Description**
    Map the device coordinates to user coordinates.
  - **Parameters**
    * `devX` – an `int` which specifies the device x-coordinate
    * `devY` – an `int` which specifies the device y-coordinate
    * `userXY` – a `double`[2] array on input. On output, the user coordinates.

---

- *mapUserToDevice*
  `public void` **mapUserToDevice**`( double` **userX**`, double` **userY**`, int[]` **devXY** `)`
  - **Description**
    Map the user coordinates (userX,userY) to the device coordinates devXY.
  - **Parameters**

    * `userX` – a `double` which specifies the user x-coordinate
    * `userY` – a `double` which specifies the user y-coordinate
    * `devXY` – an `int`[2] array on input. On output, the device coordinates.

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setCross*
  `public void` **setCross**`( double[]` **cross** `)`
  - **Description**
    Sets the value of the "Cross" attribute. This defines the point where the X and Y axes intersect. If "Cross" is not defined then the attribute "Window" is used to determine the crossing point.
  - **Parameters**
    * `cross` – is a `double` of length two containing the x and y-coordinate where the axes cross

---

- *setCross*
  `public void` **setCross**`( double` **xcross**`, double` **ycross** `)`
  - **Description**
    Sets the value of the "Cross" attribute. This defines the point where the X and Y axes intersect. If "Cross" is not defined then the attribute "Window" is used to determine the crossing point.
  - **Parameters**
    * `xcross` – a `double` which specifies the x-coordinate where the axes cross
    * `ycross` – a `double` which specifies the y-coordinate where the axes cross

---

- *setupMapping*
  `public void` **setupMapping**`( )`
  - **Description**
    Initializes the mappings between user and coordinate space. This must be called whenever the screen size, the window or the viewport may have changed. Generally, it is safest to call this each time the chart is repainted.

---

- *setWindow*
  `public void` **setWindow**`( double[]` **value** `)`

– **Description**
Sets the window in user coordinates along an axis.
– **Parameters**
  * `value` – a `double` array which contains the minimum and maximum of the window along an axis

# *class* **Axis1D**

An x-axis or a y-axis.

Axis1D is created by com.imsl.chart.AxisXY as its child. It can be retrieved using the method getAxisX() or getAxisY().

It in turn creates the following child nodes: com.imsl.chart.AxisLine, com.imsl.chart.AxisLabel, com.imsl.chart.AxisTitle, com.imsl.chart.AxisUnit, com.imsl.chart.MajorTick, com.imsl.chart.MinorTick and com.imsl.chart.Grid.

The number of tick marks ("Number" attribute) is set to 5, but autoscaling can change this value.

## Declaration

public class com.imsl.chart.Axis1D
**extends** com.imsl.chart.ChartNode (page 920)

## Methods

---

- *getAxisLabel*
  `public AxisLabel` **getAxisLabel**`( )`
  – **Description**
  Returns the label node associated with this axis.
  – **Returns** – the `AxisLabel` node created as a child by this node

- *getAxisLine*
  `public AxisLine` **getAxisLine**`( )`
  – **Description**
  Returns the axis line node associated with this axis.

– **Returns** – the `AxisLine` node created as a child by this node

---

- *getAxisTitle*
  `public AxisTitle` **getAxisTitle**`( )`

  – **Description**
  Returns the title node associated with this axis.

  – **Returns** – the `AxisTitle` node created as a child by this node

---

- *getAxisUnit*
  `public AxisUnit` **getAxisUnit**`( )`

  – **Description**
  Returns the unit node associated with this axis.

  – **Returns** – the `AxisUnit` node created as a child by this node

---

- *getFirstTick*
  `public double` **getFirstTick**`( )`

  – **Description**
  Convenience routine to get the "FirstTick" attribute.

  – **Returns** – the `double` value of the "FirstTick" attribute, if defined. Otherwise, window[0] is returned.

---

- *getGrid*
  `public Grid` **getGrid**`( )`

  – **Description**
  Returns the grid node associated with this axis.

  – **Returns** – the `Grid` node created as a child by this node

---

- *getMajorTick*
  `public MajorTick` **getMajorTick**`( )`

  – **Description**
  Returns the major tick node associated with this axis.

  – **Returns** – the `MajorTick` node created as a child by this node

---

- *getMinorTick*
  `public MinorTick` **getMinorTick**`( )`

  – **Description**
  Returns the minor tick node associated with this axis.

  – **Returns** – the `MinorTick` node created as a child by this node

---

- *getTickInterval*
  `public double` **getTickInterval**`( )`
  - **Description**
    Retrieves the tick interval.
  - **Returns** – a `double` which specifies the tick interval

---

- *getTicks*
  `public double[]` **getTicks**`( )`
  - **Description**
    Returns the value of the "Ticks" attribute, if set. If not set, then computed tick values are returned.
  - **Returns** – the `double` value of the "Ticks" attribute, if defined. Otherwise, the computed tick values are returned.

---

- *getType*
  `public int` **getType**`( )`
  - **Description**
    Returns the axis type.
  - **Returns** – an `int` which specifies the node type; can be `AXIS_X`, `AXIS_Y`, `AXIS_X_TOP` or `AXIS_Y_RIGHT`

---

- *getWindow*
  `public double[]` **getWindow**`( )`
  - **Description**
    Returns the window for an Axis1D.
  - **Returns** – a `double` array of length two containing the range of this axis.

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    - `draw` – the `Draw` object to be painted

---

- *setFirstTick*
  `public void` **setFirstTick**`( double` **firstTick** `)`
  - **Description**
    Convenience routine to set the "FirstTick" attribute.

– **Parameters**
  * `firstTick` – a `double`, the location of the first tick

---

- *setTickInterval*
  `public void` **setTickInterval**`( double` **tickInterval** `)`
  – **Description**
    Sets the tick interval.
  – **Parameters**
    * `tickInterval` – a `double` which specifies a tick interval

---

- *setTicks*
  `public void` **setTicks**`( double[]` **ticks** `)`
  – **Description**
    Sets the value of the "Ticks" attribute. The attribute Number is set to the length of the array.
  – **Parameters**
    * `ticks` – an array of `doubles` which contain the location, in user coordinates, of the major tick marks. If set, this attribute overrides the automatic computation of the tick values.

---

- *setType*
  `public void` **setType**`( int` **type** `)`
  – **Description**
    Sets the type of this node.
  – **Parameters**
    * `type` – an `int` which specifies the node type; can be `AXIS_X`, `AXIS_Y`, `AXIS_X_TOP` or `AXIS_Y_RIGHT`

---

- *setWindow*
  `public void` **setWindow**`( double[]` **window** `)`
  – **Description**
    Sets the window for an Axis1D.
  – **Parameters**
    * `window` – is an array of length two containing the range of this axis.

---

- *setWindow*
  `public void` **setWindow**`( double` **min**`, double` **max** `)`
  – **Description**
    Sets the window for an Axis1D.

– **Parameters**

  * `min` – a `double` which specifies the value of the left/bottom end of the axis
  * `max` – a `double` which specifies the value of the right/top end of the axis

# *class* AxisLabel

The labels on an axis.

`AxisLabel` is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getAxisLabel().

Axis labels are placed at the tick mark locations. The number of tick marks is determined by the attribute "Number". Tick marks are evenly spaced. If the attribute "Labels" is defined then it is used to label the tick marks.

If "Labels" is not defined, the ticks are labeled numerically. The endpoint label values are obtained from the attribute "Window". The numbers are formatted using the attribute "TextFormat".

Text attributes in this node control the drawing of the axis labels.

## Declaration

public class com.imsl.chart.AxisLabel
**extends** com.imsl.chart.ChartNode (page 920)

## Methods

---

- *getLabels*
  `public Text[] getLabels( )`
  - **Description**
    Returns the "Labels" attribute.
  - **Returns** – a `String` array containing the axis labels, if set. Otherwise, `null` is returned.

---

- *paint*
  `public void paint( Draw draw )`

– **Description**
Paints this node and all of its children. This is normally called only by the paint method in this node's parent.

– **Parameters**
  * `draw` – the `Draw` object to be painted

---

- *setLabels*
`public void` **setLabels**`( java.lang.String[]` **value** `)`

  – **Description**
  Sets the axis label values for this node to be used instead of the default numbers. The attribute "Number" is also set to `value.length`.

  – **Parameters**
    * `value` – a `String` array containing the labels for the major tick marks

## *class* **AxisLine**

The axis line.

`AxisLine` is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getAxisLine().

Line attributes in this node control the drawing of the axis line.

### Declaration

public class com.imsl.chart.AxisLine
**extends** com.imsl.chart.ChartNode (page 920)

### Method

---

- *paint*
`public void` **paint**`( Draw` **draw** `)`

  – **Description**
  Paints this node and all of its children. This is normally called only by the paint method in this node's parent.

  – **Parameters**
    * `draw` – the `Draw` object to be painted

# *class* AxisTitle

The title on an axis.

`AxisTitle` is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getAxisTitle().

The axis title is the value of the "Title" attribute at this node. Text attributes in this node control the drawing of the axis title.

## Declaration

public class com.imsl.chart.AxisTitle
**extends** com.imsl.chart.ChartNode (page 920)

## Method

---

- *paint*
  `public void` **`paint`**`( Draw` **`draw`** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

# *class* AxisUnit

The unit title on an axis.

`AxisUnit` is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getAxisUnit().

The unit title is the value of the "Title" attribute at this node. Text attributes in this node control the drawing of the unit title.

## Declaration

public class com.imsl.chart.AxisUnit
**extends** com.imsl.chart.ChartNode (page 920)

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* MajorTick

The major tick marks.

MajorTick is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getMajorTick().

Line attributes in this node control the drawing of the major tick marks.

### Declaration

public class com.imsl.chart.MajorTick
**extends** com.imsl.chart.ChartNode (page 920)

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* **MinorTick**

The minor tick marks.

MinorTick is created by com.imsl.chart.Axis1D as its child. It can be retrieved using the method getMinorTick().

Line attributes in this node control the drawing of the minor tick marks.

### Declaration

| |
|---|
| public class com.imsl.chart.MinorTick<br>**extends** com.imsl.chart.ChartNode (page 920) |

### Method

---

- *paint*
  `public void` **`paint`**`( Draw` **`draw`** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    - `draw` – the `Draw` object to be painted

## *interface* **Transform**

Defines a custom transformation along an axis. Axis1D has built in support for linear and logarithmic transformations. Additional transformations can be specified by setting the "CustomTransform" attribute in an Axis1D to an object that implements this interface. The interface consists of two methods that must be implemented. Each method is the inverse of the other.

### Declaration

| |
|---|
| public interface com.imsl.chart.Transform |

### Methods

---

- *mapUnitToUser*
  `double` **mapUnitToUser**`( double` **unit** `)`
  - **Description**
    Maps points in the interval [0,1] to user coordinates.

- *mapUserToUnit*
  `double` **mapUserToUnit**`( double` **user** `)`
  - **Description**
    Maps user coordinate to the interval [0,1]. The user coordinate interval is specified by the "Window" attribute for the axis with which the transform is associated.

- *setupMapping*
  `void` **setupMapping**`( Axis1D` **axis1d** `)`
  - **Description**
    Initializes the mappings between user and coordinate space.

## *class* TransformDate

Defines a transformation along an axis that skips weekend dates.

### Declaration

public class com.imsl.chart.TransformDate
**extends** java.lang.Object
**implements** Transform

### Constructor

---

- *TransformDate*
  `public` **TransformDate**`( )`

## Methods

- *isWeekday*
`public boolean` **isWeekday**`( java.util.GregorianCalendar` **cal** `)`
  - **Description**
  Returns true if the specified date is a weekday.

---

- *mapUnitToUser*
`public double` **mapUnitToUser**`( double` **unit** `)`
  - **Description**
  Maps points in the interval [0,1] to user coordinates.

---

- *mapUserToUnit*
`public double` **mapUserToUnit**`( double` **user** `)`
  - **Description**
  Maps user coordinate to the interval [0,1]. The user coordinate interval is specified by the "Window" attribute for the axis with which the transform is associated.

---

- *setupMapping*
`public void` **setupMapping**`( Axis1D` **axis1d** `)`
  - **Description**
  Initializes the mappings between user and coordinate space.

# *class* AxisR

The R-axis in a polar plot.

`AxisR` is created by com.imsl.chart.Polar as its child. It can be retrieved using the method getAxisR().

It in turn creates the following child nodes: com.imsl.chart.AxisRLine, com.imsl.chart.AxisRLabel and com.imsl.chart.AxisRMajorTick.

The number of tick marks ("Number" attribute) is set to 4, but autoscaling can change this value.

## Declaration

public class com.imsl.chart.AxisR
**extends** com.imsl.chart.ChartNode (page 920)

## Methods

---

- *getAxisRLabel*
  `public AxisRLabel` **getAxisRLabel**`( )`
  - **Description**
    Returns the `AxisRLabel` node.

---

- *getAxisRLine*
  `public AxisRLine` **getAxisRLine**`( )`
  - **Description**
    Returns the `AxisRLine` node.

---

- *getAxisRMajorTick*
  `public AxisRMajorTick` **getAxisRMajorTick**`( )`
  - **Description**
    Returns the major tick node associated with this axis.
  - **Returns** – the `MajorTick` node created as a child by this node

---

- *getTickInterval*
  `public double` **getTickInterval**`( )`
  - **Description**
    Retrieves the tick interval.
  - **Returns** – a `double` which indicates the tick interval

---

- *getTicks*
  `public double[]` **getTicks**`( )`
  - **Description**
    Returns the value of the "Ticks" attribute, if set. If not set, then it computes and returns tick values, based on the attributes "Number" and "TickInterval".
  - **Returns** – the `double` values of the "Ticks" attribute, if defined. Otherwise, computed tick values are returned.

---

- *getWindow*
  `public double` **getWindow**`( )`
  - **Description**
    Returns the Window attribute.
  - **Returns** – a `double` which specifies the Window value

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setTickInterval*
  `public void` **setTickInterval**`( double` **tickInterval** `)`
  - **Description**
    Sets the tick interval.
  - **Parameters**
    * `tickInterval` – a `double` which specifies the tick interval

---

- *setWindow*
  `public void` **setWindow**`( double` **rmax** `)`
  - **Description**
    Sets the Window attribute. The R-axis always starts at 0. The Window attribute is the maximum value of R.
  - **Parameters**
    * `rmax` – a `double` specifying the radius at which the `AxisTheta` is drawn.

## *class* **AxisRLabel**

The labels on an axis.

`AxisRLabel` is created by com.imsl.chart.AxisR as its child. It can be retrieved using the method getAxisRLabel().

Axis labels are placed at the tick mark locations. The number of tick marks is determined by the attribute “Number”. Tick marks are evenly spaced. If the attribute “Labels” is defined then it is used to label the tick marks.

If “Labels” is not defined, the ticks are labeled numerically. The endpoint label values are obtained from the attribute “Window”. The numbers are formatted using the attribute “TextFormat”.

Text attributes in this node control the drawing of the axis labels.

### Declaration

public class com.imsl.chart.AxisRLabel
**extends** com.imsl.chart.ChartNode (page 920)

### Methods

---

- *getLabels*
  `public Text[] getLabels( )`
  - **Description**
    Returns the “Labels” attribute.
  - **Returns** – a `Text` array containing the axis labels and formatting information, if set. Otherwise, `null` is returned.

---

- *paint*
  `public void paint( Draw draw )`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node’s parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setLabels*
  `public void setLabels( java.lang.String[] value )`
  - **Description**
    Sets the axis label values for this node to be used instead of the default numbers. The attribute “Number” is also set to `value.length`.
  - **Parameters**
    * `value` – a `String` array containing the labels to be used to label the major tick marks

# *class* **AxisRLine**

The radius axis line in a polar plot.

`AxisRLine` is created by com.imsl.chart.AxisR as its child. It can be retrieved using the method getAxisRLine().

Line attributes in this node control the drawing of the axis line.

## Declaration

public class com.imsl.chart.AxisRLine
**extends** com.imsl.chart.ChartNode (page 920)

## Method

---

- *paint*
  `public void` **`paint`**`( Draw` **`draw`** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

# *class* **AxisRMajorTick**

The major tick marks for the radius axis in a polar plot.

`AxisRMajorTick` is created by com.imsl.chart.AxisR as its child. It can be retrieved using the method getAxisRMajorTick().

Line attributes in this node control the drawing of the major tick marks.

## Declaration

public class com.imsl.chart.AxisRMajorTick
**extends** com.imsl.chart.ChartNode (page 920)

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    - `draw` – the `Draw` object to be painted

## *class* AxisTheta

The angular axis in a polar plot.

`AxisTheta` is created by com.imsl.chart.Polar as its child. It can be retrieved using the method getAxisTheta().

The angles are labeled using the TextFormat attribute, which is set to `‘‘0.##\\u00b0’’`, where `\\u00b0` is the Unicode character for degrees. This labels the angles in degrees. More generally, TextFormat can be set to a NumberFormat object to format the angles in degrees.

TextFormat can also be set to a MessageFormat object. In this case, field {0} is the value in degrees, field {1} is the value in radians and field {2} is the value in radians/$\pi$. So, for labels like `1.5\\u03c0`, where `\\u03c0` is the Unicode character for $\pi$, set TextFormat to `new MessageFormat(‘‘{2,number,0.##\\u03c0}’’)`.

The number of tick marks ("Number" attribute) is set to 9, but autoscaling can change this value.

### Declaration

public class com.imsl.chart.AxisTheta
**extends** com.imsl.chart.ChartNode (page 920)

### Methods

---

- *getTicks*
  `public double[] getTicks( )`
  - **Description**
    Returns the value of the "Ticks" attribute, if set. If not set then computed tick values are returned. These are the positions at which the angles are labeled.
  - **Returns** – the `double` value of the "Ticks" attribute, if defined. Otherwise, computed tick values are returned. The ticks are in radians, not degrees.

---

- *getWindow*
  `public double[] getWindow( )`
  - **Description**
    Returns the window for an `AxisTheta`.
  - **Returns** – a `double` array of length two containing the angular range of the window.

---

- *paint*
  `public void paint( Draw draw )`
  - **Description**
    Paints this node and all of its children.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setWindow*
  `public void setWindow( double[] window )`
  - **Description**
    Sets the window for an `AxisTheta`.
  - **Parameters**
    * `window` – a `double` array of length two containing the angular range.

---

- *setWindow*
  `public void setWindow( double min, double max )`
  - **Description**
    Sets the window for an `AxisTheta`. The default Window is [0,2pi].
  - **Parameters**
    * `min` – a `double` which specifies the initial angular value, in radians.
    * `max` – a `double` which specifies the final angular value, in radians.

## *class* **GridPolar**

Draws the grid lines for a polar plot.

PolarGrid is created by com.imsl.chart.Polar as its child. It can be retrieved using the method getGridPolar().

Line attributes in this node control the drawing of the grid lines.

### Declaration

public class com.imsl.chart.GridPolar
**extends** com.imsl.chart.ChartNode (page 920)

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* **Data**

Draws a data node.

Drawing of a Data node is determined by the setting of the “DataType” attribute. Multiple bits can be set in “DataType”. If the DATA_TYPE_LINE bit is set, the line attributes are active. If the DATA_TYPE_MARKER bit is set, the marker attributes are active. If the DATA_TYPE_FILL bit is set, the fill attributes are active.

If the attribute “LabelType” is set to other than the default, then the data points are labeled. The contents of the labels are determined by the value of the “LabelType” attribute. See Chart Programmer’s Guide: Labels for details. The drawing of the labels is controlled by the text attributes.

## Declaration

public class com.imsl.chart.Data
**extends** com.imsl.chart.ChartNode (page 920)

## Constructors

---

- *Data*
  `public` **Data**`(` `ChartNode` **parent** `)`
  - **Description**
    Creates a data node.
  - **Parameters**
    - `parent` – the `ChartNode` parent of this data node

---

- *Data*
  `public` **Data**`(` `ChartNode` **parent**`,` `ChartFunction` **cf**`,` `double` **a**`,` `double` **b** `)`
  - **Description**
    Creates a data node with y values. The attribute "X" is set to the double array containing {0,1,...,y.length-1}.
  - **Parameters**
    - `parent` – the `ChartNode` parent of this data node
    - `cf` – a `ChartFunction` object that defines the function to be plotted
    - `a` – a `double`, the left endpoint
    - `b` – a `double`, the right endpoint

---

- *Data*
  `public` **Data**`(` `ChartNode` **parent**`,` `double[]` **y** `)`
  - **Description**
    Creates a data node with y values. The attribute "X" is set to the double array containing {0,1,...,y.length-1}.
  - **Parameters**
    - `parent` – the `ChartNode` parent of this data node
    - `y` – a `double` array containing the "Y" attribute in this node

---

- *Data*
  `public` **Data**`(` `ChartNode` **parent**`,` `double[]` **x**`,` `double[]` **y** `)`
  - **Description**
    Creates a data node with x and y values.

– **Parameters**
  * `parent` – the `ChartNode` parent of this data node
  * `x` – a `double` array which contains the value for the attribute “X” in this node
  * `y` – a `double` array which contains the value for the attribute “Y” in this node

### Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`
  – **Description**
  Update the data range. `range = {xmin,xmax,ymin,ymax}` The entries in range are updated to reflect the extent of the data in this node. Range is an input/output variable. Its value should be updated only if the data in this node is outside the range already in the array.
  – **Parameters**
    * `range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

- *formatLabel*
  `protected Text` **formatLabel**`( double` **x**`, double` **y** `)`

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  – **Description**
  Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  – **Parameters**
    * `draw` – the `Draw` object to be painted

### Example: Scatter Chart

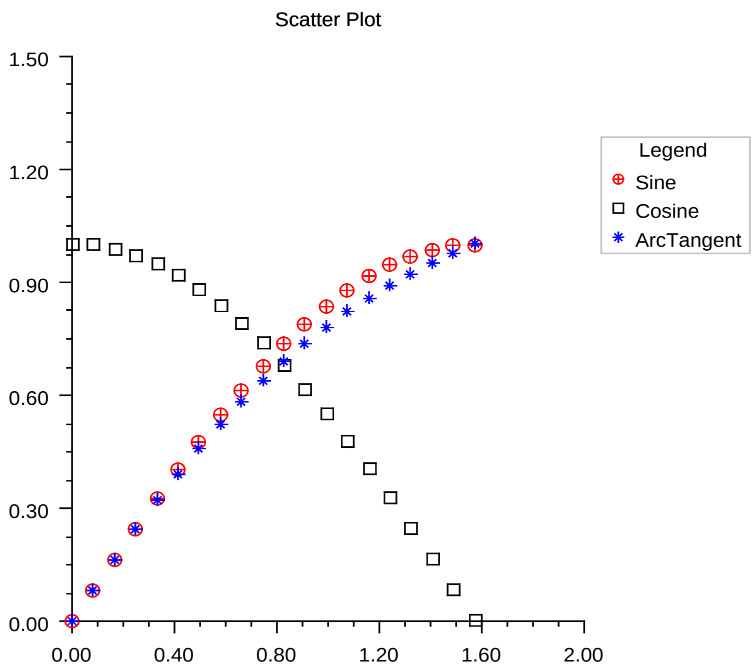
A scatter plot is constructed in this example. Three data sets are used and a legend is added to the chart. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import java.awt.Color;

public class ScatterEx1 extends javax.swing.JApplet {
```

```
private JPanelChart panel;

public void init() {
    Chart chart = new Chart(this);
    panel = new JPanelChart(chart);
    getContentPane().add(panel, java.awt.BorderLayout.CENTER);
    setup(chart);
}

static private void setup(Chart chart) {
    AxisXY axis = new AxisXY(chart);

    int npoints = 20;
    double dx = .5 * Math.PI/(npoints - 1);
    double x[] = new double[npoints];
    double y1[] = new double[npoints];
    double y2[] = new double[npoints];
    double y3[] = new double[npoints];

    // Generate some data
    for (int i = 0; i < npoints; i++){
        x[i] = i * dx;
        y1[i] = Math.sin(x[i]);
        y2[i] = Math.cos(x[i]);
        y3[i] = Math.atan(x[i]);
    }
    Data d1 = new Data(axis, x, y1);
    Data d2 = new Data(axis, x, y2);
    Data d3 = new Data(axis, x, y3);

    // Set Data Type to Marker
    d1.setDataType(d1.DATA_TYPE_MARKER);
    d2.setDataType(d2.DATA_TYPE_MARKER);
    d3.setDataType(d3.DATA_TYPE_MARKER);

    // Set Marker Types
    d1.setMarkerType(Data.MARKER_TYPE_CIRCLE_PLUS);
    d2.setMarkerType(Data.MARKER_TYPE_HOLLOW_SQUARE);
    d3.setMarkerType(Data.MARKER_TYPE_ASTERISK);

    // Set Marker Colors
    d1.setMarkerColor(Color.red);
```

```
        d2.setMarkerColor(Color.black);
        d3.setMarkerColor(Color.blue);

        // Set Data Labels
        d1.setTitle("Sine");
        d2.setTitle("Cosine");
        d3.setTitle("ArcTangent");

        // Add a Legend
        Legend legend = chart.getLegend();
        legend.setTitle(new Text("Legend"));
        chart.addLegendItem(2, chart);
        legend.setPaint(true);

        // Set the Chart Title
        chart.getChartTitle().setTitle("Scatter Plot");

    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        ScatterEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




## Example: Line Chart

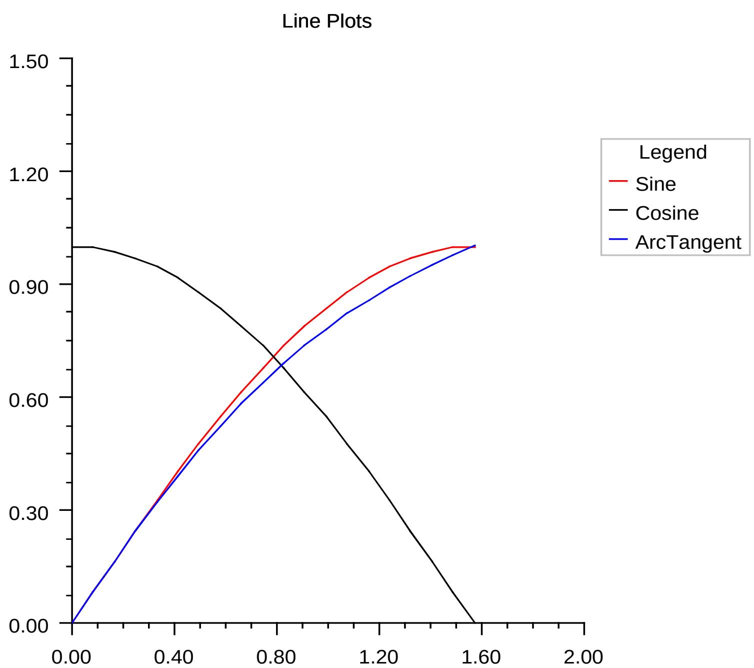
A simple line chart is constructed in this example. Three data sets are used and a legend is added to the chart. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import java.awt.Color;

public class LineEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }
```

```
static private void setup(Chart chart) {
    AxisXY axis = new AxisXY(chart);

    int npoints = 20;
    double dx = .5 * Math.PI/(npoints - 1);
    double x[] = new double[npoints];
    double y1[] = new double[npoints];
    double y2[] = new double[npoints];
    double y3[] = new double[npoints];

    // Generate some data
    for (int i = 0; i < npoints; i++){
        x[i] = i * dx;
        y1[i] = Math.sin(x[i]);
        y2[i] = Math.cos(x[i]);
        y3[i] = Math.atan(x[i]);
    }
    Data d1 = new Data(axis, x, y1);
    Data d2 = new Data(axis, x, y2);
    Data d3 = new Data(axis, x, y3);

    // Set Data Type to Line
    axis.setDataType(axis.DATA_TYPE_LINE);

    // Set Line Colors
    d1.setLineColor(Color.red);
    d2.setLineColor(Color.black);
    d3.setLineColor(Color.blue);

    // Set Data Labels
    d1.setTitle("Sine");
    d2.setTitle("Cosine");
    d3.setTitle("ArcTangent");

    // Add a Legend
    Legend legend = chart.getLegend();
    legend.setTitle(new Text("Legend"));
    chart.addLegendItem(1, chart);
    legend.setPaint(true);

    // Set the Chart Title
```

```
        chart.getChartTitle().setTitle("Line Plots");
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        LineEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




## Example: Picture Chart

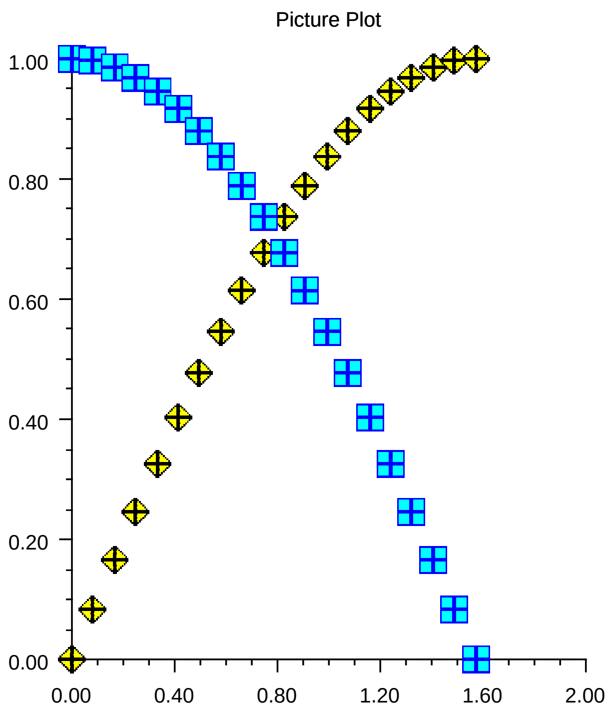
A picture plot is constructed in this example. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
```

```
import java.awt.Color;
import java.net.URL;
import javax.swing.ImageIcon;


public class PictureEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        AxisXY axis = new AxisXY(chart);

        int npoints = 20;
        double dx = .5 * Math.PI/(npoints - 1);
        double x[] = new double[npoints];
        double y1[] = new double[npoints];
        double y2[] = new double[npoints];

        // Generate some data
        for (int i = 0; i < npoints; i++){
            x[i] = i * dx;
            y1[i] = Math.sin(x[i]);
            y2[i] = Math.cos(x[i]);
        }
        Data d1 = new Data(axis, x, y1);
        Data d2 = new Data(axis, x, y2);

        // Load Images
        d1.setDataType(Data.DATA_TYPE_PICTURE);
        d1.setImage(loadImage("/com/imsl/example/chart/marker.gif"));
        d2.setDataType(Data.DATA_TYPE_PICTURE);
        d2.setImage(loadImage("/com/imsl/example/chart/marker2.gif"));

        // Set the Chart Title
        chart.getChartTitle().setTitle("Picture Plot");
    }
```

```
    static private java.awt.Image loadImage(String name) {
        return new ImageIcon(PictureEx1.class.getResource(name)).getImage();
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        PictureEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output



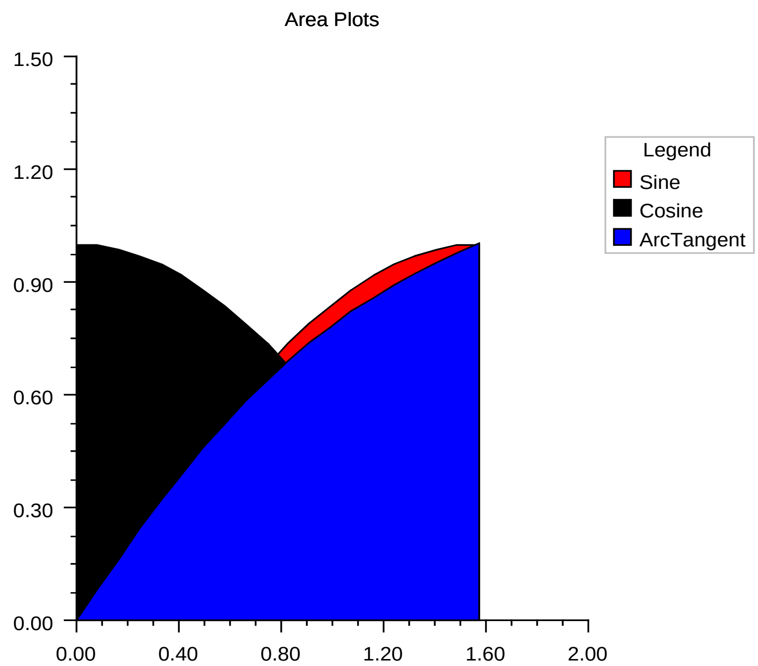
### Example: Area Chart

An area chart is constructed in this example. Three data sets are used and a legend is added to the chart. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import java.awt.Color;

public class AreaEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        AxisXY axis = new AxisXY(chart);

        int npoints = 20;
        double dx = .5 * Math.PI/(npoints - 1);
        double x[] = new double[npoints];
        double y1[] = new double[npoints];
        double y2[] = new double[npoints];
        double y3[] = new double[npoints];

        // Generate some data
        for (int i = 0;  i < npoints;  i++) {
            x[i] = i * dx;
            y1[i] = Math.sin(x[i]);
            y2[i] = Math.cos(x[i]);
            y3[i] = Math.atan(x[i]);
        }
        Data d1 = new Data(axis, x, y1);
        Data d2 = new Data(axis, x, y2);
        Data d3 = new Data(axis, x, y3);

        // Set Data Type to Fill Area
        axis.setDataType(d1.DATA_TYPE_FILL);

        // Set Line Colors
```

```
        d1.setLineColor(Color.red);
        d2.setLineColor(Color.black);
        d3.setLineColor(Color.blue);

        // Set Fill Colors
        d1.setFillColor(Color.red);
        d2.setFillColor(Color.black);
        d3.setFillColor(Color.blue);

        // Set Data Labels
        d1.setTitle("Sine");
        d2.setTitle("Cosine");
        d3.setTitle("ArcTangent");

        // Add a Legend
        Legend legend = chart.getLegend();
        legend.setTitle(new Text("Legend"));
        legend.setPaint(true);

        // Set the Chart Title
        chart.getChartTitle().setTitle("Area Plots");
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        AreaEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




# *interface* ChartFunction

An interface that allows a function to be plotted.

## Declaration

```
public interface com.imsl.chart.ChartFunction
```

## Method

---

- *f*
  `double f( double x )`

– **Description**
Function to be charted.

## *class* ChartSpline

Wrap a spline into a ChartFunction to be plotted.

### Declaration

public class com.imsl.chart.ChartSpline
**extends** java.lang.Object
**implements** ChartFunction

### Constructors

---

- *ChartSpline*
`public` **ChartSpline**`( com.imsl.math.Spline` **spline** `)`
  – **Description**
  Creates a ChartSpline from a Spline.
  – **Parameters**
    * `spline` – The Spline to be plotted.

---

- *ChartSpline*
`public` **ChartSpline**`( com.imsl.math.Spline` **spline**`, int` **ideriv** `)`
  – **Description**
  Creates a ChartSpline from the derivative of a Spline.
  – **Parameters**
    * `spline` – The Spline to be plotted.
    * `ideriv` – The derivative to be plotted. If zero, the function value is plotted. If one, the first derivative is plotted, etc.

### Method

---

- *f*
  `public double` **f**`( double` **x** `)`
  - **Description**
    Function to be charted.

## *class* Text

The value of the attribute "Title". A Title is a multi-line string with alignment information.

Line breaks are indicated by the newline character ('\n') within the string.

Titles are drawn relative to a reference point. Alignment determines the position of the reference point on the horizontally-aligned box that bounds the text.

### Declaration

public class com.imsl.chart.Text
**extends** java.lang.Object
**implements** java.io.Serializable

### Constructors

---

- *Text*
  `public` **Text**`( java.text.Format` **format**`, double` **value** `)`
  - **Description**
    Creates a text object by applying a java.text.Format to a double.
  - **Parameters**
    * `format` – a java.text.`Format`
    * `value` – the `double` to which the java.text.`Format` is to be applied.

---

- *Text*
  `public` **Text**`( java.lang.String` **string** `)`
  - **Description**
    Construct a `Text` object.
  - **Parameters**
    * `string` – a `String`

- *Text*
  `public` **Text**`( java.lang.String` **string**`, int` **alignment** `)`

  - **Description**
    Construct a `Text` object with specified alignment.
  - **Parameters**
    * `string` – a `String`
    * `alignment` – an `int` which specifies the alignment. The alignment determines the position of the reference point on the horizontally aligned box containing the drawn text. It is the bitwise combination of one of TEXT_X_LEFT, TEXT_X_CENTER, TEXT_X_RIGHT and one of TEXT_Y_BOTTOM, TEXT_Y_CENTER, TEXT_Y_TOP.

## Methods

- *getAlignment*
  `public int` **getAlignment**`( )`

  - **Description**
    Gets the alignment for this `Text` object.
  - **Returns** – the `int` which specifies the alignment for this `Text` object.

- *getOffset*
  `public double` **getOffset**`( )`

  - **Description**
    Returns the offset.

- *getString*
  `public java.lang.String` **getString**`( )`

  - **Description**
    Gets the string for this `Text` object.
  - **Returns** – the `String`

- *setAlignment*
  `public void` **setAlignment**`( int` **alignment** `)`

  - **Description**
    Sets the alignment for this `Text` object.
  - **Parameters**

* `alignment` – the `int` which specifies the alignment.

---

- *setDefaultAlignment*
  `public void` **setDefaultAlignment**`( int` **alignment** `)`
  - **Description**
    Sets the alignment to use, if it has not been set using setAlignment(int).
  - **Parameters**
    * `alignment` – the `int` which specifies the default alignment.

---

- *setDefaultOffset*
  `public void` **setDefaultOffset**`( double` **offset** `)`
  - **Description**
    Sets the default value of the offset. Offset is in units of the default marker size. Text drawn is offset in the direction of the alignment.

---

- *setOffset*
  `public void` **setOffset**`( double` **offset** `)`
  - **Description**
    Sets the offset. Offset is in units of the default marker size. Text drawn is offset in the direction of the alignment.

---

- *setString*
  `public void` **setString**`( java.lang.String` **string** `)`
  - **Description**
    Sets the string for this `Text` object.
  - **Parameters**
    * `string` – the `String`

## *class* **ToolTip**

A ToolTip for a chart element.

This class requires that the chart's component be a subclass of javax.swing.JComponent. The JComponent class can be subclassed to provide different behaviors for displaying ToolTips.

To use, create an instance of ToolTip to activate the ToolTips in a node and in the node's descendants. The ToolTip string is the value of a node's "ToolTip" attribute or, if it is null, the node's "Title" attribute.

## Declaration

public class com.imsl.chart.ToolTip
**extends** com.imsl.chart.ChartNode (page 920)
**implements** PickListener, java.awt.event.MouseMotionListener

## Constructor

---

- *ToolTip*
  `public` **ToolTip**`( ChartNode` **parent** `)`
  - **Description**
    Creates a ToolTip node that enables ToolTips on charts.
  - **Parameters**
    * `parent` – The `ChartNode` parent of this node. Do not use the root chart node for this argument, because it will normally select only the background node.

## Methods

---

- *mouseDragged*
  `public void` **mouseDragged**`( java.awt.event.MouseEvent` **e** `)`
  - **Description**
    Part of the MouseMotionListener interface.

- *mouseMoved*
  `public void` **mouseMoved**`( java.awt.event.MouseEvent` **event** `)`
  - **Description**
    Part of the MouseMotionListener interface.

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

- *pickPerformed*
  `public void` **pickPerformed**`(` `PickEvent` **event** `)`

  – **Description**
  Part of the PickListener interface.

# *class* FillPaint

A collection of methods to create Paint objects for fill areas. All of the Paint objects returned by the methods in this class are Serializable, unlike most Paint objects.

## Declaration

public class com.imsl.chart.FillPaint
**extends** java.lang.Object

## Methods

---

- *checkerboard*
  `public static java.awt.Paint` **checkerboard**`(` `int` **n**, `java.awt.Color` **colorA**, `java.awt.Color` **colorB** `)`

  – **Description**
  Returns a checkerboard pattern.

  – **Parameters**
    * `n` – is the size of the pattern in pixels.
    * `colorA` – is one of the colors.
    * `colorB` – is the other color.

  – **Returns** – a Paint containing the checkerboard pattern.

- *crosshatch*
  `public static java.awt.Paint` **crosshatch**`(` `int` **n**, `int` **p**, `java.awt.Color` **colorBackground**, `java.awt.Color` **colorLine** `)`

  – **Description**
  Returns a horizonal and vertical crosshatch pattern.

  – **Parameters**
    * `n` – is the size of the pattern in pixels.

    * `p` – is the number of pixels between the vertical lines.
    * `colorBackground` – is the background color.
    * `colorLine` – is the line color.
  - **Returns** – a Paint containing the pattern.

---

- *diagonal*
  `public static java.awt.Paint` **diagonal**`( int` **n**`, java.awt.Color` **colorA**`, java.awt.Color` **colorB** `)`
  - **Description**
    Returns a diagonal pattern.
  - **Parameters**
    * `n` – is the size of the pattern in pixels.
    * `colorA` – is one of the colors.
    * `colorB` – is the other color.
  - **Returns** – a Paint containing the checkerboard pattern.

---

- *diamond*
  `public static java.awt.Paint` **diamond**`( int` **n**`, int` **p**`, java.awt.Color` **colorBackground**`, java.awt.Color` **colorLine** `)`
  - **Description**
    Returns a diamond pattern (a checkerboard rotated 45 degrees).
  - **Parameters**
    * `n` – is the size of the pattern in pixels.
    * `p` – is the thickness of the line.
    * `colorBackground` – is the color of the background.
    * `colorLine` – is the color of the line.
  - **Returns** – a Paint containing the diamond pattern.

---

- *diamondHatch*
  `public static java.awt.Paint` **diamondHatch**`( int` **n**`, int` **p**`, java.awt.Color` **colorBackground**`, java.awt.Color` **colorLine** `)`
  - **Description**
    Returns a crosshatch on a 45 degree angle.
  - **Parameters**
    * `n` – is the size of the pattern in pixels.
    * `p` – is the number of pixels between the vertical lines.
    * `colorBackground` – is the background color.
    * `colorLine` – is the line color.
  - **Returns** – a Paint containing the pattern.

---

- *dot*

  `public static java.awt.Paint` **dot**`( int` **n**`, int` **r**`, java.awt.Color` **colorBackground**`, java.awt.Color` **colorCircle** `)`

  - **Description**

    Returns a pattern that is an array of circles.
  - **Parameters**
    * `n` – is the size of the pattern in pixels.
    * `r` – is the radius, in pixels, of the circles in the pattern.
    * `colorBackground` – is the background color.
    * `colorCircle` – is the color of the circles.
  - **Returns** – a Paint containing the pattern.

---

- *horizontalStripe*

  `public static java.awt.Paint` **horizontalStripe**`( int` **n**`, int` **p**`, java.awt.Color` **colorBackground**`, java.awt.Color` **colorLine** `)`

  - **Description**

    Returns a horizontally striped pattern.
  - **Parameters**
    * `n` – is the size of the pattern in pixels.
    * `p` – is the number of pixels between the vertical lines.
    * `colorBackground` – is the background color.
    * `colorLine` – is the line color.
  - **Returns** – a Paint containing the pattern.

---

- *image*

  `public static java.awt.Paint` **image**`( javax.swing.ImageIcon` **imageIcon** `)`

  - **Description**

    Returns a tiling of an image.
  - **Parameters**
    * `imageIcon` – is the image to be tiled.
  - **Returns** – a Paint containing the tiling of the image.

---

- *verticalStripe*

  `public static java.awt.Paint` **verticalStripe**`( int` **n**`, int` **p**`, java.awt.Color` **colorBackground**`, java.awt.Color` **colorLine** `)`

  - **Description**

    Returns a vertically striped pattern.
  - **Parameters**
    * `n` – is the size of the pattern in pixels.

- * `p` – is the number of pixels between the vertical lines.
  * `colorBackground` – is the background color.
  * `colorLine` – is the line color.
- **Returns** – a Paint containing the pattern.

## *class* Draw

Chart tree renderer.

Renders the chart tree to the screen.

### Declaration

public class com.imsl.chart.Draw
**extends** java.lang.Object

### Fields

- protected static final `double` **RADIAN**
- protected static final `int` **NONE**
- protected static final `int` **LINE**
- protected static final `int` **MARKER**
- protected static final `int` **FILL**
- protected static final `int` **TEXT**
- protected static final `int` **IMAGE**
- protected static final `int` **ERROR_BAR**
- protected `java.awt.Graphics2D` **graphics**
- protected `java.awt.geom.GeneralPath` **path**
- protected `ChartNode` **node**
- protected `int` **currentType**

- protected `boolean` **haveLineProperties**
- protected `java.awt.Color` **lineColor**
- protected `float` **lineWidth**
- protected `float[]` **lineDashPattern**
- protected `boolean` **haveMarkerProperties**
- protected `java.awt.Color` **markerColor**
- protected `float` **markerSize**
- protected `int` **markerType**
- protected `float` **markerThickness**
- protected `float[]` **markerDashPattern**
- protected `boolean` **haveFillProperties**
- protected `java.awt.Color` **fillColor**
- protected `java.awt.Color` **fillOutlineColor**
- protected `int` **fillType**
- protected `int` **fillOutlineType**
- protected `java.awt.Paint` **fillPaint**
- protected `boolean` **haveTextProperties**
- protected `java.awt.Font` **textFont**
- protected `java.awt.Color` **textColor**
- protected `int` **textAngle**
- protected `float` **scaleFont**
- protected `boolean` **haveImageProperties**
- protected `java.awt.Component` **imageObserver**
- protected `boolean` **haveErrorBarProperties**
- protected static final `int` **LAST**
  - Flag for the last data marker.

- protected static final `float` **MARKER_SCALE**
  - Normal marker size in pixels is screen width times MARKER_SCALE.
- protected static final `float[][][]` **outline**
  - Markers defined on a [-1,1] x [-1,1] grid. Each row is a continuous polyline, {x1,y1, x2,y2, x3,y3, etc.} If a row contains only a single number then that number is taken as the radius of a circle with center at (0,0).

## Constructor

---

- *Draw*
  `public` **Draw**`( java.awt.Graphics` **graphics**`, java.awt.Dimension` **bounds** `)`
  - **Description**
    Contructs a `Draw` object.
  - **Parameters**
    * `graphics` – is the graphics context in which to draw.
    * `bounds` – is the size of the chart to be drawn.

## Methods

---

- *check*
  `protected void` **check**`( int` **type** `)`
- *drawArc*
  `public void` **drawArc**`( int` **x**`, int` **y**`, int` **width**`, int` **height**`, int` **startAngle**`, int` **arcAngle** `)`
  - **Description**
    Draws the outline of a circular or elliptical arc covering the specified rectangle. The center of the arc is center of this rectangle.
  - **Parameters**
    * `x` – An `int` which specifies the x of the rectangle.
    * `y` – An `int` which specifies the y of the rectangle origin.
    * `width` – An `int` which specifies the width of the rectangle.
    * `height` – An `int` which specifies the height of the rectangle.
    * `startAngle` – An `int` which specifies the start angle in degrees. startAngle = 0 is equivalent to the 3-o'clock position.

* `arcAngle` – An `int` which specifies the arcAngle. drawArc draws the arc from startAngle to startAngle+arcAngle. A positive arcAngle indicates a counter-clockwise rotation. A negative arcAngle implies a clockwise rotation.

---

- *drawErrorBar*

  `public void` **`drawErrorBar`**`( int` **`x0`**`, int` **`y0`**`, int` **`x1`**`, int` **`y1`**`, int` **`flag`** `)`

  - **Description**

    Draw an error bar.

  - **Parameters**

    * `x0` – an `int` which specifies the x-coordinate of the beginning reference point
    * `y0` – an `int` which specifies the y-coordinate of the beginning reference point
    * `x1` – an `int` which specifies the x-coordinate of the ending reference point
    * `y1` – an `int` which specifies the y-coordinate of the ending reference point
    * `flag` – indicates which caps to draw (0=none, 1=bottom, 2=top, 3=both).

---

- *drawImage*

  `public void` **`drawImage`**`( java.awt.Image` **`image`**`, int` **`x`**`, int` **`y`** `)`

  - **Description**

    Draw an image.

  - **Parameters**

    * `image` – the `Image` object to be drawn
    * `x` – an `int` which specifies the x-coordinate of the reference point
    * `y` – an `int` which specifies the y-coordinate of the reference point

---

- *drawLine*

  `public void` **`drawLine`**`( int` **`x0`**`, int` **`y0`**`, int` **`x1`**`, int` **`y1`** `)`

  - **Description**

    Draw a line from (x0,y0) to (x1,y1).

  - **Parameters**

    * `x0` – an `int` which specifies the x0 of the line origin, (x0,y0)
    * `y0` – an `int` which specifies the y0 of the line origin, (x0,y0)
    * `x1` – an `int` which specifies the x1 of the line destination, (x1,y1)
    * `y1` – an `int` which specifies the y1 of the line destination, (x1,y1)

---

- *drawMarker*

  `public void` **`drawMarker`**`( int` **`x`**`, int` **`y`** `)`

– **Description**
  Draw a marker.
– **Parameters**
  * `x` – an `int` which specifies the x of the marker destination, (x,y)
  * `y` – an `int` which specifies the y of the marker destination, (x,y)

---

• *drawRotatedText*
`protected void` **drawRotatedText**`( Text` **text**`, int` **x**`, int` **y**`, float` **angle** `)`

– **Description**
  Draws a text object, at the specified angle, with its lower left point being at (x,y).

---

• *drawText*
`protected void` **drawText**`( java.awt.Graphics` **g**`, Text` **text** `)`

– **Description**
  Draws the text.

---

• *drawText*
`public java.awt.Dimension` **drawText**`( Text` **text**`, int` **x**`, int` **y** `)`

– **Description**
  Draws a text object.
– **Parameters**
  * `text` – the `Textobject to be drawn`
  * `x` – an `int` which specifies the abscissa of the (x,y) point at which to start drawing the text
  * `y` – an `int` which specifies the ordinate of the (x,y) point at which to start drawing the text

---

• *drawText*
`protected java.awt.Dimension` **drawText**`( Text` **text**`, int` **x**`, int` **y**`, boolean` **dimensionOnly** `)`

– **Description**
  Draws a text object. The angle of the string is given by textAngle. Consider the horizontally and vertically aligned bounding box around the string. The box below corresponds to textAngle == 45.

```
*--*--*
|    o|
|   l |
```

```
*  l  *
| e   |
|H    |
*--*--*
```

The reference point corresponds to one of the 8 starred points on the bounding box, as indicated by the "alignment" attribute" in the text object.

- **Parameters**
  - `text` – a `Text` object to be drawn.
  - `x` – an `int` which specifies the x-coordinate of the reference point.
  - `y` – an `int` which specifies the y-coordinate of the reference point.
  - `dimensionOnly` – a `boolean` which is true if only the bounding box is to be computed and no text actually drawn.
- **Returns** – the dimension of the bounding box.

---

- *endErrorBar*
  `public void` **`endErrorBar`**`( )`
  - **Description**
    Stop drawing an error bar.

---

- *endFill*
  `public void` **`endFill`**`( )`
  - **Description**
    Stop drawing a filled region.

---

- *endImage*
  `public void` **`endImage`**`( )`
  - **Description**
    Stop drawing an image.

---

- *endLine*
  `public void` **`endLine`**`( )`
  - **Description**
    Finish drawing lines.

---

- *endMarker*
  `public void` **`endMarker`**`( )`
  - **Description**
    Finish drawing markers.

---

- *endText*
  `public void` **endText**`( )`
  - **Description**
    Stop drawing text.

---

- *fillArc*
  `public void` **fillArc**`( int` **x**`, int` **y**`, int` **width**`, int` **height**`, int` **startAngle**`, int` **arcAngle** `)`
  - **Description**
    Fills a circular or elliptical arc covering the specified rectangle. The center of the arc is center of this rectangle.
  - **Parameters**
    * `x` – An `int` which specifies the x of the rectangle.
    * `y` – An `int` which specifies the y of the rectangle origin.
    * `width` – An `int` which specifies the width of the rectangle.
    * `height` – An `int` which specifies the height of the rectangle.
    * `startAngle` – An `int` which specifies the start angle in degrees. startAngle = 0 is equivalent to the 3-o'clock position.
    * `arcAngle` – An `int` which specifies the arcAngle.

---

- *fillPolygon*
  `public void` **fillPolygon**`( int[]` **xpoints**`, int[]` **ypoints**`, int` **npoints** `)`
  - **Description**
    Fill a polygon.
  - **Parameters**
    * `xpoints` – an `int` array which contains the abscissae of the points which define the polygon
    * `ypoints` – an `int` array which contains the ordinates of the points which define the polygon
    * `npoints` – an `int` which specifies the number of points

---

- *fillPolygon*
  `public void` **fillPolygon**`( java.awt.Polygon` **polygon** `)`
  - **Description**
    Fill a polygon defined by a `Polygon` object.
  - **Parameters**
    * `polygon` – a `Polygon` object which specifies the polygon to be filled

---

- *fillRectangle*
  `public void` **fillRectangle**`( int` **x**`, int` **y**`, int` **width**`, int` **height** `)`

– **Description**
Fill a rectangle.

– **Parameters**

* `x` – an `int` which specifies the abscissa of the origin of the rectangle
* `y` – an `int` which specifies the ordinate of the origin of the rectangle
* `width` – an `int` which specifies the width of the rectangle
* `height` – an `int` which specifies the height of the rectangle

---

- *getClipBounds*
`public java.awt.Rectangle` **getClipBounds**`( )`

– **Description**
Get the clipping rectangle.

– **Returns** – a `Rectangle` object which contains the clipping bounds

---

- *getDeviceMarkerSize*
`public float` **getDeviceMarkerSize**`( )`

– **Description**
Returns the marker size in device corrdinates.

---

- *getScaleFont*
`public double` **getScaleFont**`( )`

– **Description**
Returns the factor by which fonts are to be scaled.

---

- *getSize*
`protected java.awt.Dimension` **getSize**`( Text` **text** `)`

– **Description**
Returns the size of the bounding box for a text object. This does not take into account any rotation.

---

- *setClip*
`public void` **setClip**`( java.awt.Rectangle` **clip** `)`

– **Description**
Set the clipping rectangle.

– **Parameters**

* `clip` – a `Rectangle` object which contains the clipping bounds

---

- *setNode*
`public void` **setNode**`( ChartNode` **node** `)`

– **Description**
Set the current ChartNode. This is used to get drawing attributes from the tree.

– **Parameters**
  * `node` – a `ChartNode` object

---

- *setScaleFont*
`public void` **setScaleFont**`( double` **scaleFont** `)`

  – **Description**
  Set a factor by which fonts are to be scaled.

---

- *start*
`public void` **start**`( Chart` **chart** `)`

  – **Description**
  Called just before a chart is drawn.

---

- *startErrorBar*
`public void` **startErrorBar**`( )`

  – **Description**
  Start drawing an error bar.

---

- *startFill*
`public void` **startFill**`( )`

  – **Description**
  Start drawing a filled region.

---

- *startImage*
`public void` **startImage**`( )`

  – **Description**
  Start drawing an image.

---

- *startLine*
`public void` **startLine**`( )`

  – **Description**
  Start drawing lines.

---

- *startMarker*
`public void` **startMarker**`( )`

– **Description**
Start drawing markers.

---

- *startText*
`public void` **`startText`**`( )`

  – **Description**
  Start drawing text.

---

- *stop*
`public void` **`stop`**`( )`

  – **Description**
  Called when a chart is finished being drawn.

---

- *translate*
`public void` **`translate`**`( int x, int` **`y`** `)`

  – **Description**
  Translates the origin to the point (x,y)

  – **Parameters**
  * `x` – an `int` which specifies the x of the new origin
  * `y` – an `int` which specifies the y of the new origin

## *class* JFrameChart

JFrameChart is a JFrame that contains a chart. It is designed to allow simple charting applications to be quickly implemented. It contains a menu bar with "Print" and "Exit" menu items.

### Declaration

public class com.imsl.chart.JFrameChart
**extends** javax.swing.JFrame

### Constructors

---

- *JFrameChart*
`public` **`JFrameChart`**`( )`

– **Description**
Creates new JFrameChart to display a chart.

---

- *JFrameChart*
`public` **`JFrameChart`**`( Chart` **`chart`** `)`
  – **Description**
  Creates new JFrameChart to display a given chart.
  – **Parameters**
    * `chart` – is the Chart to be displayed

### Methods

---

- *getChart*
`public Chart` **`getChart`**`( )`
  – **Description**
  Return the Chart object.
  – **Returns** – the chart being displayed by this container

---

- *getPanel*
`public JPanelChart` **`getPanel`**`( )`
  – **Description**
  Returns the JPanelChart into which the chart is drawn.

---

- *setChart*
`public void` **`setChart`**`( Chart` **`chart`** `)`
  – **Description**
  Sets the chart to be handled.
  – **Parameters**
    * `chart` – is the new chart

## *class* **JPanelChart**

A Swing JPanel that contains a chart. This class causes the contained chart to be redrawn as necessary.

## Declaration

public class com.imsl.chart.JPanelChart
**extends** javax.swing.JPanel

## Field

---

- protected `Chart` **chart**
  - The embedded chart.

## Constructors

---

- *JPanelChart*
  `public` **JPanelChart**`( )`
  - **Description**
    Creates new JPanelChart. This creates a new Chart object.

---

- *JPanelChart*
  `public` **JPanelChart**`( Chart` **chart** `)`
  - **Description**
    Creates new JPanelChart using a given Chart object.
  - **Parameters**
    * `chart` – is the Chart to be displayed in this panel

## Methods

---

- *getChart*
  `public Chart` **getChart**`( )`
  - **Description**
    Return the Chart object.
  - **Returns** – the chart being displayed by this container

---

- *paintComponent*
  `public void` **paintComponent**`( java.awt.Graphics` **g** `)`

– **Description**
Calls the UI delegate's paint method, if the UI delegate is non-null. We pass the delegate a copy of the Graphics object to protect the rest of the paint code from irrevocable changes (for example, `Graphics.translate`). If you override this in a subclass you should not make permanent changes to the passed in `Graphics`). For example, you should not alter the clip Rectangle or modify the transform. If you need to do these operations you may find it easier to create a new `Graphics` from the passed in `Graphics` and manipulate it. Further, if you do not invoker super's implementation you must honor the opaque property, that is if this component is opaque, you must completely fill in the background in a non-opaque color. If you do not honor the opaque property you will likely see visual artifacts.

– **Parameters**
  * `g` – the Graphics for painting the chart.

---

- *print*
`public void` **print**`( )`

– **Description**
Print the chart, centered on a page.

---

- *setChart*
`public void` **setChart**`( Chart` **chart** `)`

– **Description**
Sets the Chart to be handled by this container.

– **Parameters**
  * `chart` – is the Chart to be displayed by this container

## *class* **DrawPick**

The DrawPick class.

### Declaration

| |
|---|
| public class com.imsl.chart.DrawPick<br>**extends** com.imsl.chart.Draw (page 1003) |

## Constructor

---

- *DrawPick*
  `public` **DrawPick**`(` `java.awt.event.MouseEvent` **event**, `java.awt.Graphics` **graphics**, `java.awt.Dimension` **bounds** `)`
  - **Description**
    Contructs a DrawPick object.
  - **Parameters**
    * `event` – is a `MouseEvent`
    * `graphics` – is the graphics context in which to draw.
    * `bounds` – is the size of the chart to be drawn.

## Methods

---

- *drawArc*
  `public void` **drawArc**`(` `int` **x**, `int` **y**, `int` **width**, `int` **height**, `int` **startAngle**, `int` **arcAngle** `)`
  - **Description**
    Draw an arc.
  - **Parameters**
    * `x` – An `int` which specifies the x of the rectangle origin, (x,y). The center of the arc is the center of this rectangle.
    * `y` – An `int` which specifies the y of the rectangle origin, (x,y). The center of the arc is the center of this rectangle.
    * `width` – An `int` which specifies the width of the rectangle.
    * `height` – An `int` which specifies the height of the rectangle.
    * `startAngle` – An `int` which specifies the start angle in degrees. startAngle = 0 is equivalent to the 3-o'clock position.
    * `arcAngle` – An `int` which specifies the arcAngle. drawArc draws the arc from startAngle to startAngle+arcAngle. A positive arcAngle indicates a counter-clockwise rotation. A negative arcAngle implies a clockwise rotation.

- *drawErrorBar*
  `public void` **drawErrorBar**`(` `int` **x0**, `int` **y0**, `int` **x1**, `int` **y1** `)`
  - **Description**
    Draw ErrorBar

– **Parameters**
  * `x0` – an `int` which specifies the x-coordinate of the beginning reference point
  * `y0` – an `int` which specifies the y-coordinate of the beginning reference point
  * `x1` – an `int` which specifies the x-coordinate of the ending reference point
  * `y1` – an `int` which specifies the y-coordinate of the ending reference point

---

- *drawImage*
  `public void` **`drawImage`**`( java.awt.Image` **`image`**`, int` **`x`**`, int` **`y`** `)`

  – **Description**
  Draw Image
  – **Parameters**
  * `image` – the `Image` object to be drawn
  * `x` – an `int` which specifies the x-coordinate of the reference point
  * `y` – an `int` which specifies the y-coordinate of the reference point

---

- *drawLine*
  `public void` **`drawLine`**`( int` **`x0`**`, int` **`y0`**`, int` **`x1`**`, int` **`y1`** `)`

  – **Description**
  Draw a line from (x0,y0) to (x1,y1).
  – **Parameters**
  * `x0` – an `int` which specifies the x0 of the line origin, (x0,y0)
  * `y0` – an `int` which specifies the y0 of the line origin, (x0,y0)
  * `x1` – an `int` which specifies the x1 of the line destination, (x1,y1)
  * `y1` – an `int` which specifies the y1 of the line destination, (x1,y1)

---

- *drawMarker*
  `public void` **`drawMarker`**`( int` **`x`**`, int` **`y`** `)`

  – **Description**
  Draw a marker.
  – **Parameters**
  * `x` – an `int` which specifies the x of the marker destination, (x,y)
  * `y` – an `int` which specifies the y of the marker destination, (x,y)

---

- *drawText*
  `public java.awt.Dimension` **`drawText`**`( Text` **`text`**`, int` **`x`**`, int` **`y`** `)`

  – **Description copied from Draw** (**page 1003**)
  Draws a text object.
  – **Parameters**

* `text` – the `Textobject to be drawn`
* `x` – an `int` which specifies the abscissa of the (x,y) point at which to start drawing the text
* `y` – an `int` which specifies the ordinate of the (x,y) point at which to start drawing the text

---

• *endErrorBar*
`public void` **endErrorBar**`( )`

– **Description**
End ErrorBar

---

• *endFill*
`public void` **endFill**`( )`

– **Description**
End fill

---

• *endImage*
`public void` **endImage**`( )`

– **Description**
End Image

---

• *endLine*
`public void` **endLine**`( )`

– **Description**
Finish drawing lines.

---

• *endMarker*
`public void` **endMarker**`( )`

– **Description**
Finish drawing markers.

---

• *endText*
`public void` **endText**`( )`

– **Description**
End Text

---

• *fillArc*
`public void` **fillArc**`( int` **x**`, int` **y**`, int` **width**`, int` **height**`, int` **startAngle**`, int` **arcAngle** `)`

– **Description**
Fills a circular or elliptical arc covering the specified rectangle. The center of the arc is center of this rectangle.

– **Parameters**

* `x` – An `int` which specifies the x of the rectangle.
* `y` – An `int` which specifies the y of the rectangle origin.
* `width` – An `int` which specifies the width of the rectangle.
* `height` – An `int` which specifies the height of the rectangle.
* `startAngle` – An `int` which specifies the start angle in degrees. `startAngle` = 0 is equivalent to the 3-o'clock position.
* `arcAngle` – An `int` which specifies the arcAngle. `drawArc` draws the arc from `startAngle` to `startAngle+arcAngle`. A positive arcAngle indicates a counter-clockwise rotation. A negative `arcAngle` implies a clockwise rotation.

---

• *fillPolygon*
`public void` **fillPolygon**`( int[]` **xpoints**`, int[]` **ypoints**`, int` **npoints** `)`

– **Description**
Fill a polygon.

– **Parameters**

* `xpoints` – an `int` array which contains the abscissae of the points which define the polygon
* `ypoints` – an `int` array which contains the ordinates of the points which define the polygon
* `npoints` – an `int` which specifies the number of points

---

• *fillPolygon*
`public void` **fillPolygon**`( java.awt.Polygon` **polygon** `)`

– **Description**
Fill a polygon defined by a `Polygon` object.

– **Parameters**

* `polygon` – a `Polygon` object which specifies the polygon to be filled

---

• *fillRectangle*
`public void` **fillRectangle**`( int` **x**`, int` **y**`, int` **width**`, int` **height** `)`

– **Description**
Fill a rectangle.

– **Parameters**

* `x` – an `int` which specifies the abscissa of the origin of the rectangle
* `y` – an `int` which specifies the ordinate of the origin of the rectangle
* `width` – an `int` which specifies the width of the rectangle

* `height` – an `int` which specifies the height of the rectangle

---

- *fire*
  `public void` **fire**`( )`
  - **Description**
    Fires the pickListeners for all of the picked nodes.

---

- *getTolerance*
  `public int` **getTolerance**`( )`
  - **Description**
    Get the minimum distance that an event can be from a point or a line and still be considered a hit.
  - **Returns** – an `int` which specifies the minimum distance that an event can be from a point or a line and still be considered a hit

---

- *pickNode*
  `protected void` **pickNode**`( )`
  - **Description**
    Register the currentNode as the "picked" node if the "PickListener" attribute is defined for the current node.

---

- *setNode*
  `public void` **setNode**`( ChartNode` **node** `)`
  - **Description**
    Set the current ChartNode. This is used to get drawing attributes from the tree.
  - **Parameters**
    * `node` – a `ChartNode` object

---

- *setTolerance*
  `public void` **setTolerance**`( int` **tolerance** `)`
  - **Description**
    Set the minimum distance that an event can be from a point or a line and still be considered a hit.
  - **Parameters**
    * `tolerance` – an `int` which specifies the minimum distance that an event can be from a point or a line and still be considered a hit

---

- *startErrorBar*
  `public void` **startErrorBar**`( )`

- **Description**
  Start ErrorBar

---

- *startFill*
  `public void` **`startFill`**`( )`

  - **Description**
    Fill

---

- *startImage*
  `public void` **`startImage`**`( )`

  - **Description**
    Start Image

---

- *startLine*
  `public void` **`startLine`**`( )`

  - **Description**
    Start drawing lines.

---

- *startMarker*
  `public void` **`startMarker`**`( )`

  - **Description**
    Start drawing markers.

---

- *startText*
  `public void` **`startText`**`( )`

  - **Description**
    Start drawing text

---

- *translate*
  `public void` **`translate`**`( int x, int y )`

  - **Description**
    Translates the origin to the point (x,y)
  - **Parameters**
    * `x` – an `int` which specifies the x of the new origin
    * `y` – an `int` which specifies the y of the new origin

# *class* **PickEvent**

An event that indicates that a chart element has been selected.

## Declaration

```
public class com.imsl.chart.PickEvent
extends java.awt.event.MouseEvent
```

## Constructors

- *PickEvent*
  `public` **PickEvent**`( java.awt.Component` **source**`, int` **id**`, long` **when**`, int` **modifiers**`, int` **x**`, int` **y**`, int` **clickCount**`, boolean` **popupTrigger** `)`
  - **Description**
    Construct a `PickEvent` object at point (x,y).
  - **Parameters**
    - `source` – the `Component` that originated the event
    - `id` – an `int` that identifies the event
    - `when` – a `long` that gives the time the event occurred
    - `modifiers` – an `int` that gives the modifier keys down during event (e.g. shift, ctrl, alt, meta)
    - `x` – an `int`, the x coordinate of the point (x,y)
    - `y` – an `int`, the y coordinate of the point (x,y)
    - `clickCount` – an `int` which specifies the number of mouse button clicks necessary to trigger the event
    - `popupTrigger` – is a `boolean`, true if this event is a trigger for a popup menu

- *PickEvent*
  `public` **PickEvent**`( java.awt.event.MouseEvent` **event** `)`
  - **Description**
    Construct a `PickEvent` object.
  - **Parameters**
    - `event` – a `MouseEvent`

## Methods

---

- *getNode*
  `public ChartNode` **getNode**`( )`
  - **Description**
    Gets this `ChartNode`.

---

- *pointToLine*
  `public static double` **pointToLine**`( int` **Px**`, int` **Py**`, int[]` **devA**`, int[]` **devB** `)`
  - **Description**
    Compute the distance from the point (Px,Py) to the line segment AB. If the closest point from P to the line AB is not between A and B then the distance to the closer of A and B is returned.
  - **Parameters**
    * `Px` – an `int`, the x coordinate of the point (Px,Py)
    * `Py` – an `int`, the y coordinate of the point (Px,Py)
    * `devA` – an `int` array which contains the point which defines the head of the line segment.
    * `devB` – an `int` array which contains the point which defines the tail of the line segment.
  - **Returns** – a `double`, the distance from the point (Px,Py) to the line segment AB.

---

- *setNode*
  `public void` **setNode**`( ChartNode` **node** `)`
  - **Description**
    Sets the `ChartNode`.
  - **Parameters**
    * `node` – the `ChartNode` to be set

# *interface* PickListener

The listener interface for receiving pick events.

### Declaration

public interface com.imsl.chart.PickListener
**implements** java.util.EventListener

### Method

---

- *pickPerformed*
  `void` **pickPerformed**`( PickEvent` **event** `)`
  - **Description**
    Public interface for `PickListener`.
  - **Parameters**
    - `event` – a `PickEvent`

## *class* JspBean

JspBean is used to refer to charts in a Java Server Page that are later rendered using the ChartServlet.

### Declaration

public class com.imsl.chart.JspBean
**extends** java.lang.Object
**implements** java.io.Serializable

### Constructor

---

- *JspBean*
  `public` **JspBean**`( )`
  - **Description**
    Creates a JspBean object.

## Methods

---

- *getChartServletName*
  `public java.lang.String` **getChartServletName**`( )`
  - **Description**
    Returns the URL of the servlet used to render the chart.

---

- *getCreateImageMap*
  `public boolean` **getCreateImageMap**`( )`
  - **Description**
    Returns true if a client-side imagemap is to be created.

---

- *getId*
  `public java.lang.String` **getId**`( )`
  - **Description**
    Returns the identifier number for the chart. It is assigned a unique random value by the constructor.

---

- *getImageMap*
  `public java.lang.String` **getImageMap**`( )`
  - **Description**
    Returns an HTML for the client-side imagemap. This HTML is to be inserted into the page being generated.
  - **Returns** – the HTML map tag. If no map is defined the empty string is returned.

---

- *getImageTag*
  `public java.lang.String` **getImageTag**`( )`
  - **Description**
    Returns an HTML image tag. This is normally inserted into the HTML being generated.
  - **Returns** – the HTML tag refering to the servlet-generated chart. If no image is defined the empty string is returned.

---

- *getMapName*
  `public java.lang.String` **getMapName**`( )`

– **Description**
Returns the name of the client-size imagemap. This is null if CreateImageMap is false.

---

• *getSize*
`public java.awt.Dimension` **getSize**`( )`

– **Description**
Returns the size of the generated image.

---

• *registerChart*
`public void` **registerChart**`( Chart` **chart**`,`
`javax.servlet.http.HttpServletRequest` **request** `)`

– **Description**
Saves the chart and sets the chart attribute "Size". The chart is saved using the saveChart method. If the ChartServletName has not been set, it is set to "*ContextPath*`/servlet/com.imsl.chart.ChartServlet`", where "*ContextPath* is the context path in the request.

– **Parameters**
  * `chart` – is the chart to be registered. The java.awt.Dimension-value attribute "Size" is set in the root node of the chart tree. The Size attribute is used by com.imsl.chart.ChartServlet.
  * `request` – from the Java Server Page.

---

• *saveChart*
`protected void` **saveChart**`( Chart` **chart**`,`
`javax.servlet.http.HttpServletRequest` **request** `)`

– **Description**
Saves the chart so that a servlet can later render it. The chart is saved in the HttpSession, associated with the request, under the key "`chart`*NNN*", where *NNN* is the value of the `id` property. This method can be overridden to change the mechanism by which the bean and the servlet correspond.

– **Parameters**
  * `chart` – is the chart to be registered.
  * `request` – from the Java Server Page. The chart is saved in its session object.

---

• *setChartServletName*
`public void` **setChartServletName**`( java.lang.String` **chartServletName**
`)`

– **Description**
Sets the URL of the servlet used to render the chart. Its initial value is null. It is usually set in the registerChart method. Since this is used only in the image tag, it can be specified relative to the URL of the page in which the image tag is used.

– **Parameters**

* `chartServletName` – is the location of the chart servlet to be used in the generated image tag.

---

- *setCreateImageMap*
`public void` **setCreateImageMap**`( boolean` **createImageMap** `)`

  – **Description**
  Sets a flag indicating if a client-size imagemap is to be generated. Its initial value is false. A client-side image map has an entry for each node in which the chart attribute `HREF` is defined. The values of `HREF` attributes are URLs. Such regions are treated by the browser as hyperlinks.

  – **Parameters**

  * `createImageMap` – is true if a client-side image map is to be generated.

---

- *setSize*
`public void` **setSize**`( java.awt.Dimension` **size** `)`

  – **Description**
  Sets the size of the generated image. Its initial value is `new Dimension(300,300)`.

  – **Parameters**

  * `size` – is the size of the generated image.

---

- *setSize*
`public void` **setSize**`( int` **width**`, int` **height** `)`

  – **Description**
  Sets the size of the generated image. Its initial value is `new Dimension(300,300)`.

  – **Parameters**

  * `width` – is the width of the generated image.
  * `height` – is the height of the generated image.

## *class* ChartServlet

The base class for chart servlets.

This class requires a servlet container.

The behavior of this class depends on the version of the Java runtime being used.

- *JDK1.4 or later.* Images are rendered using the standard class javax.imageio.ImageIO. This class can be used on a headless server. Java runs in a headless mode if the system property `java.awt.headless=true`. This class turns off caching in the ImageIO class (calls `javax.imageio.ImageIO.setUseCache(false)`).
- *JDK1.3 or earlier.* Since the ImageIO class does not exist in older versions of Java, this class requires the Java Advanced Imaging Toolkit (JAI) and a running windowing system to create images. It will not work on a "headless" server.

## Declaration

public class com.imsl.chart.ChartServlet
**extends** javax.servlet.http.HttpServlet

## Constructor

- *ChartServlet*
  `public` **ChartServlet**`( )`

## Methods

- *doGet*
  `protected void` **doGet**`( javax.servlet.http.HttpServletRequest` **request**`, javax.servlet.http.HttpServletResponse` **response** `) throws javax.servlet.ServletException, java.io.IOException`
  - **Description**
    Returns the chart as a PNG image. The HTTP request parameter "id" selects the chart.
  - **Parameters**
    * `request` – an `HttpServletRequest` object that contains the request the client has made of the servlet
    * `response` – an `HttpServletResponse` object that contains the response the servlet sends to the client

- *getChart*
  `protected Chart` **getChart**`( javax.servlet.http.HttpServletRequest` **request** `)`
  - **Description**
    Returns the chart found in the session saved with the key "chart"+id, where id is the value of the "id" parameter in the request. This method can be overridden to change how charts are communicated to this servlet.
  - **Parameters**
    * `request` – an `HttpServletRequest` object that contains the request the client has made of the servlet
  - **Returns** – the chart to be rendered or null if the chart cannot be found.

---

- *init*
  `public void` **init**`( ) throws javax.servlet.ServletException`

## *class* DrawMap

Creates an HTML client-side imagemap from a chart tree. Entries in the imagemap correspond to nodes that define the HREF attribute.

### Declaration

public class com.imsl.chart.DrawMap
**extends** com.imsl.chart.Draw (page 1003)

### Constructor

---

- *DrawMap*
  `public` **DrawMap**`( java.awt.Graphics` **graphics**`, java.awt.Dimension` **bounds** `)`
  - **Description**
    Contructs a DrawMap object.
  - **Parameters**
    * `graphics` – is the graphics context in which to draw.
    * `bounds` – is the size of the chart to be drawn.

## Methods

- *circle*
  `protected void` **`circle`**`( int` **`x`**`, int` **`y`**`, int` **`r`** `)`
  - **Description**
    Sets a circle as the target.
  - **Parameters**
    * `x` – is the x-coordinate of the center of the circle
    * `y` – is the y-coordinate of the center of the circle
    * `r` – is the radius of the circle

---

- *drawArc*
  `public void` **`drawArc`**`( int` **`x`**`, int` **`y`**`, int` **`width`**`, int` **`height`**`, int` **`startAngle`**`, int` **`arcAngle`** `)`
  - **Description**
    Draws the outline of a circular or elliptical arc covering the specified rectangle. The center of the arc is center of this rectangle.
  - **Parameters**
    * `x` – An `int` which specifies the x of the rectangle.
    * `y` – An `int` which specifies the y of the rectangle origin.
    * `width` – An `int` which specifies the width of the rectangle.
    * `height` – An `int` which specifies the height of the rectangle.
    * `startAngle` – An `int` which specifies the start angle in degrees. startAngle = 0 is equivalent to the 3-o'clock position.
    * `arcAngle` – An `int` which specifies the arcAngle. drawArc draws the arc from startAngle to startAngle+arcAngle. A positive arcAngle indicates a counter-clockwise rotation. A negative arcAngle implies a clockwise rotation.

---

- *drawErrorBar*
  `public void` **`drawErrorBar`**`( int` **`x0`**`, int` **`y0`**`, int` **`x1`**`, int` **`y1`**`, int` **`flag`** `)`
  - **Description**
    Draw an error bar.
  - **Parameters**
    * `x0` – an `int` which specifies the x-coordinate of the beginning reference point
    * `y0` – an `int` which specifies the y-coordinate of the beginning reference point
    * `x1` – an `int` which specifies the x-coordinate of the ending reference point

* `y1` – an `int` which specifies the y-coordinate of the ending reference point
* `flag` – an `int` that indicates which caps to draw (0=none, 1=bottom, 2=top, 3=both).

---

- *drawImage*

  `public void` **drawImage**`( java.awt.Image` **image**`, int` **x**`, int` **y** `)`

  - **Description**

    Draw Image
  - **Parameters**
    * `image` – the `Image` object to be drawn
    * `x` – an `int` which specifies the x-coordinate of the reference point
    * `y` – an `int` which specifies the y-coordinate of the reference point

---

- *drawLine*

  `public void` **drawLine**`( int` **x0**`, int` **y0**`, int` **x1**`, int` **y1** `)`

  - **Description**

    Draw a line from (x0,y0) to (x1,y1).
  - **Parameters**
    * `x0` – an `int` which specifies the x0 of the line origin, (x0,y0)
    * `y0` – an `int` which specifies the y0 of the line origin, (x0,y0)
    * `x1` – an `int` which specifies the x1 of the line destination, (x1,y1)
    * `y1` – an `int` which specifies the y1 of the line destination, (x1,y1)

---

- *drawMarker*

  `public void` **drawMarker**`( int` **x**`, int` **y** `)`

  - **Description**

    Draw a marker.
  - **Parameters**
    * `x` – an `int` which specifies the x of the marker destination, (x,y)
    * `y` – an `int` which specifies the y of the marker destination, (x,y)

---

- *drawText*

  `protected java.awt.Dimension` **drawText**`( Text` **text**`, int` **x**`, int` **y**`, boolean` **dimensionOnly** `)`

  - **Description copied from Draw** (**page 1003**)

    Draws a text object. The angle of the string is given by textAngle. Consider the horizontally and vertically aligned bounding box around the string. The box below corresponds to textAngle == 45.

```
*--*--*
|    o|
```

```
|   l |
*  l  *
| e   |
|H    |
*--*--*
```

The reference point corresponds to one of the 8 starred points on the bounding box, as indicated by the "alignment" attribute" in the text object.

- **Parameters**
  - `text` – a `Text` object to be drawn.
  - `x` – an `int` which specifies the x-coordinate of the reference point.
  - `y` – an `int` which specifies the y-coordinate of the reference point.
  - `dimensionOnly` – a `boolean` which is true if only the bounding box is to be computed and no text actually drawn.
- **Returns** – the dimension of the bounding box.

---

- *endErrorBar*
  `public void` **endErrorBar**`( )`
  - **Description copied from Draw** (**page 1003**)
    Stop drawing an error bar.

---

- *endFill*
  `public void` **endFill**`( )`
  - **Description copied from Draw** (**page 1003**)
    Stop drawing a filled region.

---

- *endImage*
  `public void` **endImage**`( )`
  - **Description copied from Draw** (**page 1003**)
    Stop drawing an image.

---

- *endLine*
  `public void` **endLine**`( )`
  - **Description copied from Draw** (**page 1003**)
    Finish drawing lines.

---

- *endMarker*
  `public void` **endMarker**`( )`
  - **Description copied from Draw** (**page 1003**)
    Finish drawing markers.

- *endText*
  `public void` **endText**`( )`
  - **Description copied from Draw** (**page 1003**)
    Stop drawing text.

- *fillArc*
  `public void` **fillArc**`( int` **x**`, int` **y**`, int` **width**`, int` **height**`, int` **startAngle**`, int` **arcAngle** `)`
  - **Description**
    Fills a circular or elliptical arc covering the specified rectangle. The center of the arc is center of this rectangle.
  - **Parameters**
    * `x` – An `int` which specifies the x of the rectangle.
    * `y` – An `int` which specifies the y of the rectangle origin.
    * `width` – An `int` which specifies the width of the rectangle.
    * `height` – An `int` which specifies the height of the rectangle.
    * `startAngle` – An `int` which specifies the start angle in degrees. `startAngle = 0` is equivalent to the 3-o'clock position.
    * `arcAngle` – An `int` which specifies the `arcAngle`. `drawArc` draws the arc from `startAngle` to `startAngle+arcAngle`. A positive `arcAngle` indicates a counter-clockwise rotation. A negative `arcAngle` implies a clockwise rotation.

- *fillPolygon*
  `public void` **fillPolygon**`( int[]` **xpoints**`, int[]` **ypoints**`, int` **npoints** `)`
  - **Description**
    Fill a polygon.
  - **Parameters**
    * `xpoints` – an `int` array which contains the abscissae of the points which define the polygon
    * `ypoints` – an `int` array which contains the ordinates of the points which define the polygon
    * `npoints` – an `int` which specifies the number of points

- *fillPolygon*
  `public void` **fillPolygon**`( java.awt.Polygon` **polygon** `)`
  - **Description**
    Fill a polygon defined by a `Polygon` object.
  - **Parameters**
    * `polygon` – a `Polygon` object which specifies the polygon to be filled

- *fillRectangle*
  `public void` **fillRectangle**`( int` **x**`, int` **y**`, int` **width**`, int` **height** `)`

  – **Description**
    Fill a rectangle.
  – **Parameters**
    * `x` – an `int` which specifies the abscissa of the origin of the rectangle
    * `y` – an `int` which specifies the ordinate of the origin of the rectangle
    * `width` – an `int` which specifies the width of the rectangle
    * `height` – an `int` which specifies the height of the rectangle

- *getALT*
  `protected java.lang.String` **getALT**`( )`

  – **Description**
    Returns the current ALT string.

- *getHREF*
  `protected java.lang.String` **getHREF**`( )`

  – **Description**
    Returns the current HREF string.

- *getMap*
  `public java.lang.String` **getMap**`( )`

  – **Description**
    Returns the body of the HTML imagemap.
  – **Returns** – the body of the HTML client-side imagemap. The actual <map>and </map>tags are not included, so that the client code can more easily add attributes to the <map>tag.

- *getTolerance*
  `public int` **getTolerance**`( )`

  – **Description**
    Get the minimum distance that an event can be from a point or a line and still be considered a hit.
  – **Returns** – an `int` which specifies the minimum distance that an event can be from a point or a line and still be considered a hit

- *poly*
  `protected void` **poly**`( int[]` **x**`, int[]` **y** `)`

- **Description**
  Sets a polygon as the target.
- **Parameters**
  * `x` – is an array containing the x-coordinates of the polygon.
  * `y` – is an array containing the y-coordinates of the polygon.

---

- *rect*
  `protected void rect( int x, int y, int w, int h )`
  - **Description**
    Sets a rectangle as the target.
  - **Parameters**
    * `x` – is the x-coordinate of the left edge of the rectangle
    * `y` – is the y-coordinate of the top edge of the rectangle
    * `w` – is the width of the rectangle
    * `h` – is the height of the rectangle

---

- *setNode*
  `public void setNode( ChartNode node )`
  - **Description**
    Set the current ChartNode. This is used to get drawing attributes from the tree.
  - **Parameters**
    * `node` – a `ChartNode` object

---

- *setTolerance*
  `public void setTolerance( int tolerance )`
  - **Description**
    Set the minimum distance that an event can be from a point or a line and still be considered a hit.
  - **Parameters**
    * `tolerance` – an `int` which specifies the minimum distance that an event can be from a point or a line and still be considered a hit

---

- *startErrorBar*
  `public void startErrorBar( )`
  - **Description copied from Draw** (**page 1003**)
    Start drawing an error bar.

---

- *startFill*
  `public void startFill( )`

– **Description copied from Draw** (**page 1003**)
Start drawing a filled region.

---

- *startImage*
`public void` **startImage**`( )`

  – **Description copied from Draw** (**page 1003**)
  Start drawing an image.

---

- *startLine*
`public void` **startLine**`( )`

  – **Description**
  Start drawing lines.

---

- *startMarker*
`public void` **startMarker**`( )`

  – **Description**
  Start drawing markers.

---

- *startText*
`public void` **startText**`( )`

  – **Description copied from Draw** (**page 1003**)
  Start drawing text.

---

- *translate*
`public void` **translate**`( int x, int y )`

  – **Description**
  Translates the origin to the point (x,y)

  – **Parameters**
    * `x` – an `int` which specifies the x of the new origin
    * `y` – an `int` which specifies the y of the new origin

## *class* BoxPlot

Draws a multiple-group Box plot.

For each group of observations, the box limits represent the lower quartile (25th percentile) and upper quartile (75th percentile). The median is displayed as a line across the box. Whiskers are drawn from the upper quartile to the upper adjacent value, and from the lower quartile to the lower adjacent value.

Optional notches may be displayed to show a 95 percent confidence interval about the median, at $\pm 1.58\ IRQ/\sqrt{n}$, where $IRQ$ is the interquartile range and $n$ is the number of observations. Outside and far outside values may be displayed as symbols. Outside values are outside the inner fence. Far out values are outside the outer fence.

The BoxPlot has several child nodes. Any of these nodes can be disabled by setting their "Paint" attribute to `false`.

- The "Bodies" node has the main body of the box plot elements. Its fill attributes determine the drawing of (notched) rectangle. Its line attributes determine the drawing of the median line. The width of the box is controlled by the "MarkerSize" attribute.
- The "Whiskers" node draws the lines to the upper and lower quartile. Its drawing is affected by the marker attributes.
- The "FarMarkers" node hold the far markers. Its drawing is affected by the marker attributes.
- The "OutsideMarkers" node hold the outside markers. Its drawing is affected by the marker attributes.

## Declaration

public class com.imsl.chart.BoxPlot
**extends** com.imsl.chart.Data (page 982)

## Inner Class

### *class* BoxPlot.Statistics

Computes the statistics for one set of observations in a `Boxplot`.

#### Declaration

public static class com.imsl.chart.BoxPlot.Statistics
**extends** java.lang.Object
**implements** java.io.Serializable

### Constructor

- *BoxPlot.Statistics*
  `public` **BoxPlot.Statistics**`( double[]` **obs** `)`
  - **Description**
    Creates a new instance of `BoxPlot.Statistics`.
  - **Parameters**
    * `obs` – a `double` array containing the set of observations. There must be at least 4 observations to compute the statistics.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if there are fewer than 4 observations.

### Methods

- *getFarMarkers*
  `public double[]` **getFarMarkers**`( )`
  - **Description**
    Returns the array of far markers.
  - **Returns** – a `double` array containing the far markers for this set

---

- *getLowerAdjacentValue*
  `public double` **getLowerAdjacentValue**`( )`
  - **Description**
    Returns the lower adjacent value.
  - **Returns** – a `double` which specifies the lower adjacent value

---

- *getLowerQuartile*
  `public double` **getLowerQuartile**`( )`
  - **Description**
    Returns the lower quartile value.
  - **Returns** – a `double` which specifies the lower quartile value (25th percentile)

---

- *getMaximumValue*
  `public double` **getMaximumValue**`( )`

- **Description**
  Returns the maximum value of the observations.
- **Returns** – a `double` which specifies the the maximum value of this set

---

- *getMedian*
  `public double` **getMedian**`( )`
  - **Description**
    Returns the median value.
  - **Returns** – a `double` which specifies the median value for the set of observations

---

- *getMedianLowerConfidenceInterval*
  `public double` **getMedianLowerConfidenceInterval**`( )`
  - **Description**
    Returns the lower confidence interval for the median.
  - **Returns** – a `double` which specifies the lower confidence interval for the median value of this set of observations

---

- *getMedianUpperConfidenceInterval*
  `public double` **getMedianUpperConfidenceInterval**`( )`
  - **Description**
    Returns the upper confidence interval for the median.
  - **Returns** – a `double` which specifies the upper confidence interval for the median value of this set of observations

---

- *getMinimumValue*
  `public double` **getMinimumValue**`( )`
  - **Description**
    Returns the minimum value of the observations.
  - **Returns** – a `double` which specifies the the minimum value of this set

---

- *getNumberObservations*
  `public int` **getNumberObservations**`( )`
  - **Description**
    Returns the number of observations.
  - **Returns** – an `int` which specifies the number of observations in this set

---

- *getOutsideMarkers*
  `public double[]` **getOutsideMarkers**`( )`

- **Description**
  Returns the array of outside markers.
- **Returns** – a `double` array containing the outside markers for this set

---

- *getUpperAdjacentValue*
  `public double` **getUpperAdjacentValue**`( )`
  - **Description**
    Returns the lower adjacent value.
  - **Returns** – a `double` which specifies the upper adjacent value

---

- *getUpperQuartile*
  `public double` **getUpperQuartile**`( )`
  - **Description**
    Returns the upper quartile value.
  - **Returns** – a `double` which specifies the upper quartile value (75th percentile)

### Fields

---

- public static final `int` **BOXPLOT_TYPE_HORIZONTAL**
  - Value for attribute "BoxPlotType" indicating that this is a horizontal box plot. Used in connection with `BoxPlot` nodes.
- public static final `int` **BOXPLOT_TYPE_VERTICAL**
  - Value for attribute "BoxPlotType" indicating that this is a horizontal box plot. Used in connection with `BoxPlot` nodes.

### Constructors

---

- *BoxPlot*
  `public` **BoxPlot**`( AxisXY` **axis**`, double[][]` **obs** `)`
  - **Description**
    Constructs a box plot chart.
  - **Parameters**
    * `axis` – an `AxisXY` object, the parent of this node

* obs – a double array which contains the observations. The length of each row in obs must be at least 4.

---

- *BoxPlot*
  `public` **BoxPlot**`( AxisXY` **axis**`, double[]` **x**`, BoxPlot.Statistics[]` **statistics** `)`
  - **Description**
    Constructs a box plot chart node with specified x values.
  - **Parameters**
    * axis – an AxisXY object, the parent of this node
    * x – a double array which contains the $x$ values
    * statistics – is an array of BoxPlot.Statistics objects. The number of BoxPlot.Statistics must equal the length of x.

---

- *BoxPlot*
  `public` **BoxPlot**`( AxisXY` **axis**`, double[]` **x**`, double[][]` **obs** `)`
  - **Description**
    Constructs a box plot chart node with specified x values.
  - **Parameters**
    * axis – an AxisXY object, the parent of this node
    * x – a double array which contains the $x$ values
    * obs – a double array which contains the observations for each $x$. The number of rows in obs must equal the length of x. The length of each row in obs must be at least 4.

## Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`
  - **Description**
    Overrides Data.dataRange.
  - **Parameters**
    * range – a double array which contains the new range

---

- *getBodies*
  `public ChartNode` **getBodies**`( )`
  - **Description**
    Returns a node containing the body elements in the Box plot.

– **Returns** – a `ChartNode` containing the bodies.

---

- *getBoxPlotType*
  `public int` **getBoxPlotType**`( )`
  – **Description**
    Returns the value of the "BoxPlotType" attribute.
  – **Returns** – an `int` which contains the "BoxPlotType". Legal values are `BOXPLOT_TYPE_VERTICAL` or `BOXPLOT_TYPE_HORIZONTAL`.

---

- *getFarMarkers*
  `public ChartNode` **getFarMarkers**`( )`
  – **Description**
    Returns the FarMarkers node.
  – **Returns** – a `ChartNode` containing the far markers

---

- *getNotch*
  `public boolean` **getNotch**`( )`
  – **Description**
    Gets the "Notch" attribute value. return a `boolean` which specifies whether the notches are to be displayed; `true` if so `false` otherwise

---

- *getOutsideMarkers*
  `public ChartNode` **getOutsideMarkers**`( )`
  – **Description**
    Returns the OutsideMarkers node.
  – **Returns** – a `ChartNode` containing the outside markers

---

- *getStatistics*
  `public BoxPlot.Statistics[]` **getStatistics**`( )`
  – **Description**
    Returns an array of `BoxPlot.Statistics` objects, one for each set of observations.
  – **Returns** – an array of `BoxPlot.Statistics` objects

---

- *getStatistics*
  `public BoxPlot.Statistics` **getStatistics**`( int` **iSet** `)`
  – **Description**
    Returns a `BoxPlot.Statistics` for a set of observations.
  – **Parameters**

* `iSet` – an `int` which specifies the index of a set whose statistics are to be returned

– **Returns** – a `BoxPlot.Statistics` object related to the `iSet` set of observations

---

• *getWhiskers*
`public ChartNode` **getWhiskers**`( )`

– **Description**
Returns the Whiskers node. return a `ChartNode` containing the whiskers

---

• *isProportionalWidth*
`public boolean` **isProportionalWidth**`( )`

– **Description**
Returns the value of the attribute "ProportionalWidth". The width of the narrowest box is determined by the "MarkerSize" attribute.

– **Returns** – a `boolean` which specifies whether the box widths are proportional. If `true` the box widths are proportional to the square root of the number of observations. If `false` all of the boxes have the same width.

---

• *paint*
`public void` **paint**`( Draw` **draw** `)`

– **Description**
Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.

– **Parameters**
  * `draw` – the `Draw` object to be painted

---

• *setBoxPlotType*
`public void` **setBoxPlotType**`( int` **value** `)`

– **Description**
Sets the "BoxPlotType" attribute value.

– **Parameters**
  * `value` – an `int` which specifies the "BoxPlotType" attribute. Legal values are `BOXPLOT_TYPE_VERTICAL` or `BOXPLOT_TYPE_HORIZONTAL`.

---

• *setLabels*
`public void` **setLabels**`( java.lang.String[]` **labels** `)`

– **Description**
Sets up an axis with labels. This turns off the tick marks and sets the "BoxPlotType" attribute. It also turns off autoscaling for the axis and sets its

"Window" and "Number" and "Ticks" attribute as appropriate for a labeled Box plot. The existing value of the "BoxPlotType" attribute is used to determine the axis to be modified.

- **Parameters**
  * `labels` – is an array of strings with which to label the axis. The number of labels must equal the number of items.

---

- *setLabels*

  `public void` **setLabels**`( java.lang.String[]` **labels**`, int` **type** `)`

  - **Description**

    Sets up an axis with labels. This turns off the tick marks and sets the "BoxPlotType" attribute. It also turns off autoscaling for the axis and sets its "Window" and "Number" and "Ticks" attribute as appropriate for a labeled Box plot.
  - **Parameters**
    * `labels` – an array of `Strings` with which to label the axis. The number of labels must equal the number of items.
    * `type` – an `int` which specifies the BoxPlotType. Legal values are `BOXPLOT_TYPE_VERTICAL` or `BOXPLOT_TYPE_HORIZONTAL`. This determines the axis to be modified.

---

- *setNotch*

  `public void` **setNotch**`( boolean` **value** `)`

  - **Description**

    Sets the attribute "Notch".
  - **Parameters**
    * `value` – a `boolean` which specifies whether notches are to be displayed; `true` if so `false` otherwise

---

- *setProportionalWidth*

  `public void` **setProportionalWidth**`( boolean` **proportionalWidth** `)`

  - **Description**

    Sets the value of the attribute "ProportionalWidth".
  - **Parameters**
    * `proportionalWidth` – a `boolean` which specifies whether the box widths are to be proportional. Is `true` if the box widths are to be proportional to the square root of the number of observations. If `false` all of the boxes have the same width. The default value is `false`.

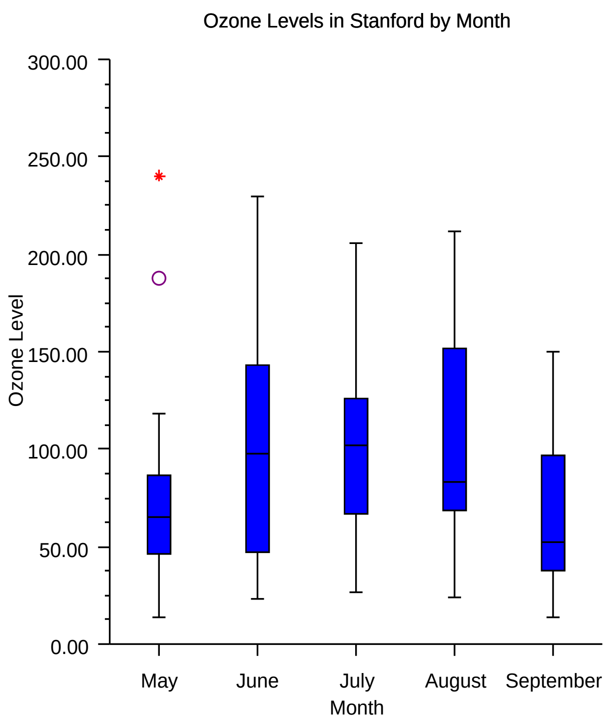
### Example: Box Plot Chart

A simple box plot chart is constructed in this example. Display of far and outside values is turned on.

```
import com.imsl.chart.*;

public class BoxPlotEx1 extends javax.swing.JApplet {

    public void init() {
        Chart chart = new Chart(this);
        JPanelChart panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        double obs[][]={{66.0, 52.0, 49.0, 64.0, 68.0, 26.0, 86.0, 52.0,
                         43.0, 75.0, 87.0, 188.0, 118.0, 103.0, 82.0,
                         71.0, 103.0, 240.0, 31.0, 40.0, 47.0, 51.0, 31.0,
                         47.0, 14.0, 71.0},

                        {61.0, 47.0, 196.0, 131.0, 173.0, 37.0, 47.0,
                         215.0, 230.0, 69.0, 98.0, 125.0, 94.0, 72.0,
                         72.0, 125.0, 143.0, 192.0, 122.0, 32.0, 114.0,
                         32.0, 23.0, 71.0, 38.0, 136.0, 169.0},

                        {152.0, 201.0, 134.0, 206.0, 92.0, 101.0, 119.0,
                         124.0, 133.0, 83.0, 60.0, 124.0, 142.0, 124.0, 64.0,
                         75.0, 103.0, 46.0, 68.0, 87.0, 27.0,
                         73.0, 59.0, 119.0, 64.0, 111.0},

                        {80.0, 68.0, 24.0, 24.0, 82.0, 100.0, 55.0, 91.0,
                         87.0, 64.0, 170.0, 86.0, 202.0, 71.0, 85.0, 122.0,
                         155.0, 80.0, 71.0, 28.0, 212.0, 80.0, 24.0,
                         80.0, 169.0, 174.0, 141.0, 202.0},

                        {113.0, 38.0, 38.0, 28.0, 52.0, 14.0, 38.0, 94.0,
                         89.0, 99.0, 150.0, 146.0, 113.0, 38.0, 66.0, 38.0,
                         80.0, 80.0, 99.0, 71.0, 42.0, 52.0, 33.0, 38.0,
                         24.0, 61.0, 108.0, 38.0, 28.0}
                       };
        double x[] = {1.0, 2.0, 3.0, 4.0, 5.0};
```

```
        String xLabels[] = {"May", "June", "July", "August", "September"};


        //  Create an instance of a BoxPlot Chart
        AxisXY axis = new AxisXY(chart);
        BoxPlot boxPlot = new BoxPlot(axis, obs);
        boxPlot.setLabels(xLabels);

        // Customize the fill color and the outside and far markers
        boxPlot.getBodies().setFillColor("blue");
        boxPlot.getOutsideMarkers().setMarkerType(boxPlot.MARKER_TYPE_HOLLOW_CIRCLE);
        boxPlot.getOutsideMarkers().setMarkerColor("purple");
        boxPlot.getFarMarkers().setMarkerType(boxPlot.MARKER_TYPE_ASTERISK);
        boxPlot.getFarMarkers().setMarkerColor("red");

        // Set titles
        chart.getChartTitle().setTitle("Ozone Levels in Stanford by Month");
        axis.getAxisX().getAxisTitle().setTitle("Month");
        axis.getAxisY().getAxisTitle().setTitle("Ozone Level");
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        BoxPlotEx1.setup(frame.getChart());
        frame.show();
    }
}
```

### Output




## *class* Contour

A Contour chart shows level curves of a two-dimensional function.

The function can be defined either as values on a rectangular grid or by scattered data points.

A set of ContourLevel objects are created as children of this node. The number of ContourLevels is one more than the number of level curves. If the level curve values are $c_0, \ldots, c_{n-1}$ then the $k$-th ContourLevel child corresponds to $c_{k-1} < z \le c_k$.

To change the look of the contour chart, change the line attributes and fill attributes in the ContourLevel nodes.

A Legend object is also created as a child of this node. It should be used instead of the usual chart legend. By default, this legend is not shown. To show it, set its paint method to true.

### Declaration

public class com.imsl.chart.Contour
**extends** com.imsl.chart.Data (page 982)

### Inner Class

### *class* Contour.Legend

A legend for a contour chart.

This legend should be used for contour charts, instead of usual chart legend.

#### Declaration

public class com.imsl.chart.Contour.Legend
**extends** com.imsl.chart.AxisXY (page 962)

#### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

### Constructors

---

- *Contour*
  `public` **Contour**`( AxisXY` **axis**`, double[]` **x**`, double[]` **y**`, double[]` **z** `)`
  - **Description**

Create a Contour chart from scattered data with computed contour levels. The contour chart is created by using a RadialBasis approximation to estimate the functions value on a rectangular grid. The contour chart is then computed as for gridded data.

– **Parameters**
  * `axis` – an `AxisXY` object, the parent of this node.
  * `x` – a `double` array which contains the x-values of the data points.
  * `y` – a `double` array which contains the y-values of the data points.
  * `z` – a `double` array which contains the z-values of the data points.

---

• *Contour*
`public` **Contour**`( AxisXY` **axis**`, double[]` **xGrid**`, double[]` **yGrid**`, double[][]` **zData** `)`

– **Description**
Create a Contour chart from rectangularly gridded data with computed contour levels. The contour levels are chosen to span the data and to be "nice" values.

– **Parameters**
  * `axis` – an `AxisXY` object, the parent of this node.
  * `xGrid` – a `double` array which contains the x-coordinate values of the grid.
  * `yGrid` – a `double` array which contains the y-coordinate values of the grid.
  * `zData` – a `double` rectangular matrix which contains the function values to be contoured. The value of the function at (xGrid[i],yGrid[j]) is given by zData[i][j]. The size of this matrix must be xGrid.length by yGrid.length.

---

• *Contour*
`public` **Contour**`( AxisXY` **axis**`, double[]` **xGrid**`, double[]` **yGrid**`, double[][]` **zData**`, double[]` **cLevel** `)`

– **Description**
Create a Contour chart from rectangularly gridded data.

– **Parameters**
  * `axis` – an `AxisXY` object, the parent of this node.
  * `xGrid` – a `double` array which contains the x-coordinate values of the grid.
  * `yGrid` – a `double` array which contains the y-coordinate values of the grid.
  * `zData` – a `double` rectangular matrix which contains the function values to be contoured. The value of the function at (xGrid[i],yGrid[j]) is given by zData[i][j]. The size of this matrix must be xGrid.length by yGrid.length.
  * `cLevel` – a `double` array which contains the values of the contour levels.

---

• *Contour*
`public` **Contour**`( AxisXY` **axis**`, double[]` **x**`, double[]` **y**`, double[]` **z**`, double[]` **cLevel**`, int` **nCenters** `)`

– **Description**
Create a Contour chart from scattered data. The contour chart is created by using a RadialBasis appoximation to estimate the functions value on a rectangular grid. The contour chart is then computed as for gridded data.

– **Parameters**
  * `axis` – an `AxisXY` object, the parent of this node.
  * `x` – a `double` array which contains the x-values of the data points.
  * `y` – a `double` array which contains the y-values of the data points.
  * `z` – a `double` array which contains the z-values of the data points.
  * `cLevel` – a `double` array which contains the values of the contour levels.
  * `nCenters` – is the number of centers to use for the radial basis approximation. The larger the number the closer, but noiser, the approximation.

## Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`

  – **Description**
  Update the data range. `range = {xmin,xmax,ymin,ymax}` The entries in range are updated to reflect the extent of the data in this node. Range is an input/output variable. Its value should be updated only if the data in this node is outside the range already in the array.

  – **Parameters**
    * `range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

---

- *getContourLegend*
  `public Contour.Legend` **getContourLegend**`( )`

  – **Description**
  Returns the contour chart legend.
  By default, the legend is not drawn because its “Paint” attribute is set to false. To show the legend set “Paint” to true, .i.e.,
  `contour.getContourLegend().setPaint(true);`

  – **Returns** – the Legend associated with the contour chart.

---

- *getContourLevel*
  `public ContourLevel[]` **getContourLevel**`( )`

– **Description**
Returns all of the contour levels.
– **Returns** – an array containing the contour levels.

---

- *getContourLevel*
`public ContourLevel` **getContourLevel**`( int` **k** `)`
  – **Description**
  Returns a ContourLevel. The k-th contour level contains the level curve equal to cLevel[k] in the constructor. It also contains the fill areas for the values in the interval (cLevel[k-1], cLevel[k]). The first contour level (k=0) contains the fill area for values less than cLevel[0] and the level curves lines where the function value equals cLevel[0]. The last contour level (k=cLevel.length) contains the fill area for values greater than cLevel[cLevel.length-1], but no level curve lines.

---

- *paint*
`public void` **paint**`( Draw` **draw** `)`
  – **Description copied from Data** (**page 982**)
  Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  – **Parameters**
    * `draw` – the `Draw` object to be painted

### Example: Contour Chart from Gridded Data

In the restricted three-body problem, two large objects (masses $M_1$ and $M_2$) a distance $a$ apart, undergoing mutual gravitational attraction, circle a common center-of-mass. A third small object (mass $m$) is assumed to move in the same plane as $M_1$ and $M_2$ and is assumed to be two small to affect the large bodies. For simplicity, we use a coordinate system that has the center of mass at the origin. $M_1$ and $M_2$ are on the $x$-axis at $x_1$ and $x_2$, respectively.

In the center-of-mass coordinate system, the effective potential energy of the system is given by

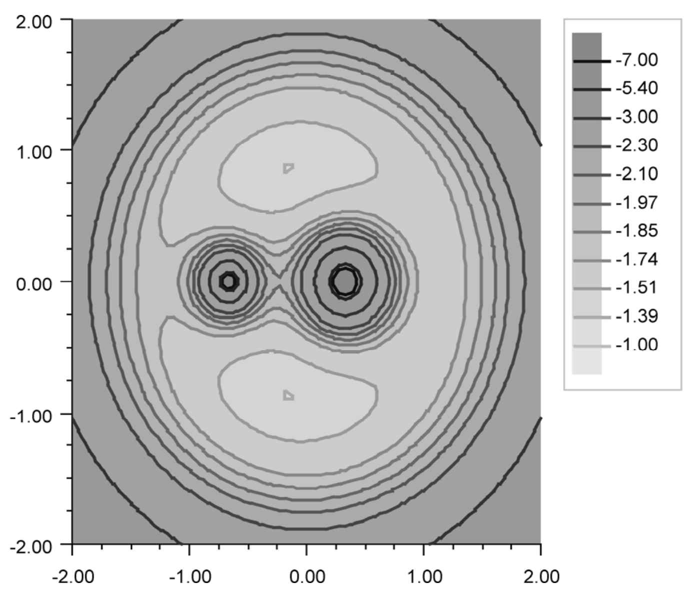
$$V = \frac{m(M_1+M_2)G}{a}\left[\frac{x_2}{\sqrt{(x-x_1)^2+y^2}} - \frac{x_1}{\sqrt{(x-x_2)^2+y^2}} - \frac{1}{2}\left(x^2+y^2\right)\right]$$

The universal gravitational constant is $G$. The following program plots the part of $V(x,y)$ inside of the square bracket. The factor $\frac{m(M_1+M_2)G}{a}$ is ignored because it just scales the plot.

```
import com.imsl.chart.*;

public class ContourEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        int nx = 80;
        int ny = 80;

        // Allocate space
        double xGrid[] = new double[nx];
        double yGrid[] = new double[ny];
        double zData[][] = new double[nx][ny];

        // Setup the grids points
        for (int i = 0;  i < nx;  i++) {
            xGrid[i] = -2 + 4.0*i/(double)(nx-1);
        }
        for (int j = 0;  j < ny;  j++) {
            yGrid[j] = -2 + 4.0*j/(double)(ny-1);
        }

        // Evaluate the function at the grid points
        for (int i = 0;  i < nx;  i++) {
            for (int j = 0;  j < ny;  j++) {
                double x = xGrid[i];
                double y = yGrid[j];
                double rm = 0.5;
                double x1 = rm / (1.0+rm);
                double x2 = x1 - 1.0;
                double d1 = Math.sqrt((x-x1)*(x-x1)+y*y);
                double d2 = Math.sqrt((x-x2)*(x-x2)+y*y);
                zData[i][j] = x2/d1 - x1/d2 - 0.5*(x*x+y*y);
            }
        }
```

```
        // Create the contour chart, with user-specified levels and a legend
        AxisXY axis = new AxisXY(chart);
        double cLevel[]
= {-7, -5.4, -3, -2.3, -2.1, -1.97, -1.85, -1.74, -1.51, -1.39, -1};
        Contour c = new Contour(axis, xGrid, yGrid, zData, cLevel);
        c.getContourLegend().setPaint(true);
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        ContourEx1.setup(frame.getChart());
        frame.show();
    }
}
```

### Output



### Example: Contour Chart from Scattered Data

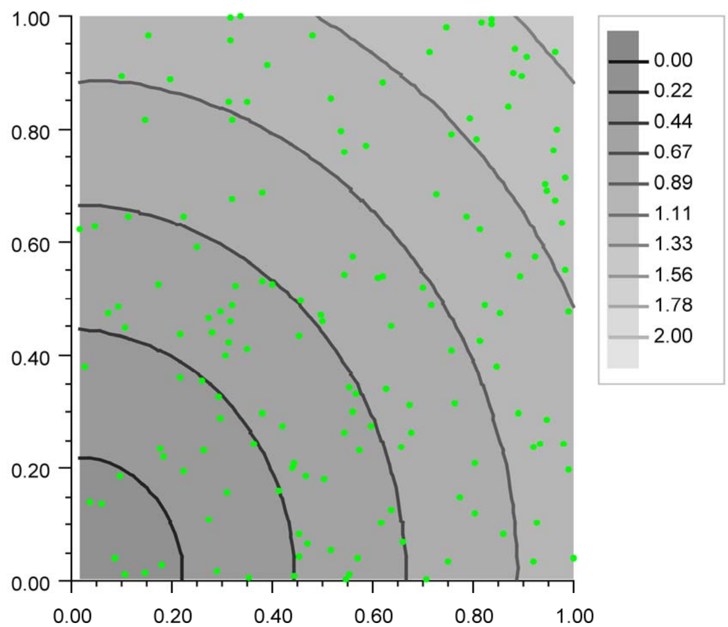
In this example, a contour chart is created from 150, randomly choosen, scattered data points. The function is $\sqrt{x^2 + y^2}$, so the level curve should be circles.

The input data is shown on top of the contours as small green circles. The chart data nodes are drawn in the order in which they are added, so the input data marker node has to be added to the axis after the contour, so that the markers are not hidden.

```
import com.imsl.chart.*;
import java.util.Random;

public class ContourEx2 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        JFrameChart jfc = new JFrameChart();
        int n = 150;

        // Allocate space
        double x[] = new double[n];
        double y[] = new double[n];
        double z[] = new double[n];

        // Evaluate the function at n random points
        Random random = new Random(123457);
        for (int k = 0;  k < n;  k++) {
            x[k] = random.nextDouble();
            y[k] = random.nextDouble();
            z[k] = Math.sqrt(x[k]*x[k] + y[k]*y[k]);
        }

        // Setup the contour plot and its legend
        AxisXY axis = new AxisXY(chart);
        Contour contour = new Contour(axis, x, y, z);
        contour.getContourLegend().setPaint(true);
```

```
        // Show the input data points as small green circles
        Data dataPoints = new Data(axis, x, y);
        dataPoints.setDataType(Data.DATA_TYPE_MARKER);
        dataPoints.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        dataPoints.setMarkerColor("green");
        dataPoints.setMarkerSize(0.5);
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        ContourEx2.setup(frame.getChart());
        frame.show();
    }
}
```

## Output



# *class* **ErrorBar**

Data points with error bars.

## Declaration

public class com.imsl.chart.ErrorBar
**extends** com.imsl.chart.Data (page 982)

## Fields

- public static final `int` **DATA_TYPE_ERROR_X**
  - Value for attribute "DataType" indicating that this is a horizontal error bar. Used in connection with ErrorBar nodes.
- public static final `int` **DATA_TYPE_ERROR_Y**
  - Value for attribute "DataType" indicating that this is a vertical error bar. Used in connection with ErrorBar nodes.

## Constructor

- *ErrorBar*
  `public` **ErrorBar**`( AxisXY` **axis**`, double[]` **x**`, double[]` **y**`, double[]` **low**`, double[]` **high** `)`
  - **Description**
    Creates a set of error bars centered at (x[k],y[k]) and with extents low[k],high[k]. If the attribute "DataType" has the bit DATA_TYPE_ERROR_X set then this is a horizontal error bar. If the bit DATA_TYPE_ERROR_Y is set then this is a vertical error bar. If neither bit is set then no error bar is drawn.
    A Data node with the same x and y values can be used to put markers at the center of each error bar.
  - **Parameters**
    * `axis` – an `Axis` object
    * `x` – a `double` array which contains the x coordinates of the points at which the error bars will be centered. This is used to set the "X" attribute.

* `y` – a `double` array which contains the y coordinates of the points at which the error bars will be centered. This is used to set the "Y" attribute.
* `low` – a `double` array which contains the values which define the minimum extent of the error bars. This is used to set the "Low" attribute.
* `high` – a `double` array which contains the values which define the maximum extent of the error bars. This is used to set the "High" attribute.

## Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`
  - **Description**
    Overrides Data.dataRange.
  - **Parameters**
    * `range` – a `double` array which contains the new range

---

- *getHigh*
  `public double[]` **getHigh**`( )`
  - **Description**
    Convenience routine to get the "High" attribute.
  - **Returns** – the `double` array which contains the value of the "High" attribute

---

- *getLow*
  `public double[]` **getLow**`( )`
  - **Description**
    Convenience routine to get the "Low" attribute.
  - **Returns** – the `double` array which contains the value of the "Low" attribute

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setHigh*
  `public void` **setHigh**`( double[]` **value** `)`

– **Description**
  Convenience routine to set the "High" attribute.
– **Parameters**
  * `value` – an `double` array which contains the "High" values.

---

- *setLow*
  `public void` **setLow**`( double[]` **value** `)`

  – **Description**
    Convenience routine to set the "Low" attribute.
  – **Parameters**
    * `value` – an `double` array which contains the "Low" values.

### Example: ErrorBar Chart

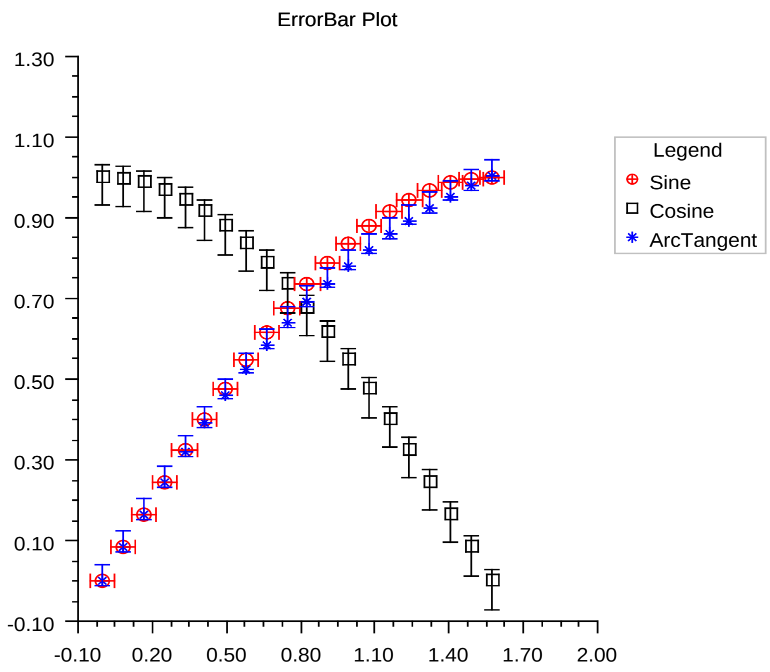
An ErrorBar chart is constructed in this example. Three data sets are used and a legend is added to the chart. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import java.awt.Color;

public class ErrorBarEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        AxisXY axis = new AxisXY(chart);

        int npoints = 20;
        double dx = .5 * Math.PI/(npoints - 1);
        double x[] = new double[npoints];
        double y1[] = new double[npoints];
        double y2[] = new double[npoints];
        double y3[] = new double[npoints];
        double low1[] = new double[npoints];
        double low2[] = new double[npoints];
        double low3[] = new double[npoints];
```

```
double hi1[] = new double[npoints];
double hi2[] = new double[npoints];
double hi3[] = new double[npoints];

// Generate some data
for (int i = 0; i < npoints; i++){
    x[i] = i * dx;
    y1[i] = Math.sin(x[i]);
    low1[i] = x[i] - .05;
    hi1[i] = x[i] + .05;
    y2[i] = Math.cos(x[i]);
    low2[i] = y2[i] - .07;
    hi2[i] = y2[i] + .03;
    y3[i] = Math.atan(x[i]);
    low3[i] = y3[i] - .01;
    hi3[i] = y3[i] + .04;
}

Data d1 = new Data(axis, x, y1);
Data d2 = new Data(axis, x, y2);
Data d3 = new Data(axis, x, y3);

// Set Data Type to Marker
d1.setDataType(d1.DATA_TYPE_MARKER);
d2.setDataType(d2.DATA_TYPE_MARKER);
d3.setDataType(d3.DATA_TYPE_MARKER);

// Set Marker Types
d1.setMarkerType(Data.MARKER_TYPE_CIRCLE_PLUS);
d2.setMarkerType(Data.MARKER_TYPE_HOLLOW_SQUARE);
d3.setMarkerType(Data.MARKER_TYPE_ASTERISK);

// Set Marker Colors
d1.setMarkerColor(Color.red);
d2.setMarkerColor(Color.black);
d3.setMarkerColor(Color.blue);

// Create an instances of ErrorBars
ErrorBar ebar1 = new ErrorBar(axis, x, y1, low1, hi1);
ErrorBar ebar2 = new ErrorBar(axis, x, y2, low2, hi2);
ErrorBar ebar3 = new ErrorBar(axis, x, y3, low3, hi3);
```

```
        // Set Data Type to Error_X
        ebar1.setDataType(ebar1.DATA_TYPE_ERROR_X);
        // Set Data Type to Error_Y
        ebar2.setDataType(ebar2.DATA_TYPE_ERROR_Y);
        ebar3.setDataType(ebar3.DATA_TYPE_ERROR_Y);

        // Set Marker Colors
        ebar1.setMarkerColor(Color.red);
        ebar2.setMarkerColor(Color.black);
        ebar3.setMarkerColor(Color.blue);

        // Set Data Labels
        d1.setTitle("Sine");
        d2.setTitle("Cosine");
        d3.setTitle("ArcTangent");

        // Add a Legend
        Legend legend = chart.getLegend();
        legend.setTitle(new Text("Legend"));
        chart.addLegendItem(0, chart);
        legend.setPaint(true);

        // Set the Chart Title
        chart.getChartTitle().setTitle("ErrorBar Plot");
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        ErrorBarEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




# *class* **HighLowClose**

High-low-close plot of stock data.

## Declaration

public class com.imsl.chart.HighLowClose
**extends** com.imsl.chart.Data (page 982)

## Field

- public static final `long` **DAY**

– Milliseconds per day

## Constructors

---

- *HighLowClose*
  `public` **HighLowClose**`(` `AxisXY` **axis**, `java.util.Date` **start**, `double[]` **high**, `double[]` **low**, `double[]` **close** `)`

  – **Description**
  Constructs a high-low-close chart node beginning with specified start date.

  – **Parameters**
  * `axis` – an `Axis` object, the parent of this node.
  * `start` – a `date` object which contains the first date.
  * `high` – a `double` array which contains the stock's high prices. This is used to set the "High" attribute.
  * `low` – a `double` array which contains the stock's low prices. This is used to set the "Low" attribute.
  * `close` – a `double` array which contains the stock's closing prices. This is used to set the "Close" attribute.

- *HighLowClose*
  `public` **HighLowClose**`(` `AxisXY` **axis**, `java.util.Date` **start**, `double[]` **high**, `double[]` **low**, `double[]` **close**, `double[]` **open** `)`

  – **Description**
  Constructs a high-low-close-open chart node beginning with specified start date.

  – **Parameters**
  * `axis` – an `Axis` object, the parent of this node.
  * `start` – a `date` object which contains the first date.
  * `high` – a `double` array which contains the stock's high prices. This is used to set the "High" attribute.
  * `low` – a `double` array which contains the stock's low prices. This is used to set the "Low" attribute.
  * `close` – a `double` array which contains the stock's closing prices. This is used to set the "Close" attribute.
  * `open` – a `double` array which contains the stock's opening prices. This is used to set the "Open" attribute.

- *HighLowClose*
  `public` **HighLowClose**`(` `AxisXY` **axis**, `double[]` **x**, `double[]` **high**, `double[]` **low**, `double[]` **close** `)`

- **Description**
  Constructs a high-low-close chart node at specified axis points.
- **Parameters**
  * `axis` – an `Axis` object, the parent of this node.
  * `x` – a `double` array which contains the axis points. This is used to set the "X" attribute.
  * `high` – a `double` array which contains the stock's high prices. This is used to set the "High" attribute.
  * `low` – a `double` array which contains the stock's low prices. This is used to set the "Low" attribute.
  * `close` – a `double` array which contains the stock's closing prices. This is used to set the "Close" attribute.

---

- *HighLowClose*
  `public` **HighLowClose**`( AxisXY` **axis**`, double[]` **x**`, double[]` **high**`,`
  `double[]` **low**`, double[]` **close**`, double[]` **open** `)`

  - **Description**
    Constructs a high-low-close-open chart node at specified axis points.
  - **Parameters**
    * `axis` – an `Axis` object, the parent of this node.
    * `x` – a `double` array which contains the axis points. This is used to set the "X" attribute.
    * `high` – a `double` array which contains the stock's high prices. This is used to set the "High" attribute.
    * `low` – a `double` array which contains the stock's low prices. This is used to set the "Low" attribute.
    * `close` – a `double` array which contains the stock's closing prices. This is used to set the "Close" attribute.
    * `open` – a `double` array which contains the stock's opening prices This is used to set the "Open" attribute.

## Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`

  - **Description**
    Overrides Data.dataRange.
  - **Parameters**
    * `range` – a `double` array which contains the new range

---

- *getClose*
  `public double[]` **getClose**`( )`

  – **Description**
    Gets the value of the attribute “Close”. return a `double` array of closing stock prices.

---

- *getHigh*
  `public double[]` **getHigh**`( )`

  – **Description**
    Convenience routine to get the “High” attribute.
  – **Returns** – the `double` array of high stock prices.

---

- *getLow*
  `public double[]` **getLow**`( )`

  – **Description**
    Convenience routine to get the “Low” attribute.
  – **Returns** – the `double` array of low stock prices.

---

- *getOpen*
  `public double[]` **getOpen**`( )`

  – **Description**
    Gets the value of the attribute “Open”. return a `double` array of opening stock prices.

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`

  – **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node’s parent.
  – **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setClose*
  `public void` **setClose**`( double[]` **value** `)`

  – **Description**
    Sets the attribute “Close”.
  – **Parameters**
    * `value` – a `double` array of closing stock prices.

---

- *setDateAxis*
  `public void` **setDateAxis**`( java.lang.String` **labelFormat** `)`

  – **Description**
  Sets up the x-axis for high-low-close plot. This turns off autoscaling on the x-axis and sets the Window attribute depending on the number of dates being plotted. The Number attribute determines the number of intervals along the x-axis.

  – **Parameters**
    * `labelFormat` – is used to format the date axis labels. It sets the TextFormat attribute in the AxisLabel node.

---

- *setHigh*
  `public void` **setHigh**`( double[]` **value** `)`

  – **Description**
  Convenience routine to set the "High" attribute.

  – **Parameters**
    * `value` – an `double` array of high stock prices.

---

- *setLow*
  `public void` **setLow**`( double[]` **value** `)`

  – **Description**
  Convenience routine to set the "Low" attribute.

  – **Parameters**
    * `value` – an `double` array of low stock prices.

---

- *setOpen*
  `public void` **setOpen**`( double[]` **value** `)`

  – **Description**
  Sets the attribute "Open".

  – **Parameters**
    * `value` – a `double` array of opening stock prices.

### Example: High-Low-Close Chart

A simple high-low-close chart is constructed in this example.

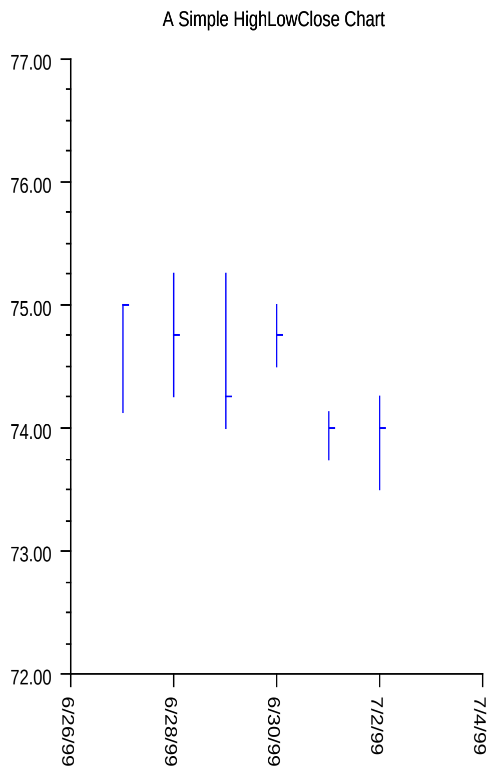
Autoscaling does not properly handle time data, so autoscaling is turned off for the $x$ (time) axis and the axis limits are set explicitly.

```
import com.imsl.chart.*;
```

```
import java.awt.Color;
import java.text.DateFormat;
import java.util.Date;
import java.util.GregorianCalendar;

public class HiLoEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        AxisXY axis = new AxisXY(chart);

        // Date is June 27, 1999
        Date date =
        new GregorianCalendar(1999, GregorianCalendar.JUNE, 27).getTime();

        double high[] = {75., 75.25, 75.25, 75., 74.125, 74.25};
        double low[] = {74.125, 74.25, 74., 74.5, 73.75, 73.50};
        double close[] = {75., 74.75, 74.25, 74.75, 74., 74.0};

        // Create an instance of a HighLowClose Chart
        HighLowClose hilo = new HighLowClose(axis, date, high, low, close);
        hilo.setMarkerColor("blue");

        // Set the HighLowClose Chart Title
        chart.getChartTitle().setTitle("A Simple HighLowClose Chart");

        // Configure the x-axis
        hilo.setDateAxis("Date(SHORT)");
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        HiLoEx1.setup(frame.getChart());
        frame.show();
```

```
    }
}
```

### Output




## *class* **Candlestick**

Candlestick plot of stock data.

Two nodes are created as children of this node. One for the up days and one for the down days.

## Declaration

public class com.imsl.chart.Candlestick
**extends** com.imsl.chart.HighLowClose (page 1061)

## Constructors

---

- *Candlestick*
  `public` **Candlestick**`(` `AxisXY` **axis**, `java.util.Date` **start**, `double[]` **high**, `double[]` **low**, `double[]` **close**, `double[]` **open** `)`
  - **Description**
    Constructs a candlestick chart node beginning with specified start date.
  - **Parameters**
    * `axis` – an `Axis` object, the parent of this node
    * `start` – a `date` object which contains the first date
    * `high` – a `double` array which contains the stock's high prices This is used to set the "High" attribute.
    * `low` – a `double` array which contains the stock's low prices This is used to set the "Low" attribute.
    * `close` – a `double` array which contains the stock's closing prices This is used to set the "Close" attribute.
    * `open` – a `double` array which contains the stock's opening prices This is used to set the "Open" attribute.

- *Candlestick*
  `public` **Candlestick**`(` `AxisXY` **axis**, `double[]` **x**, `double[]` **high**, `double[]` **low**, `double[]` **close**, `double[]` **open** `)`
  - **Description**
    Constructs a candlestick chart node at specified axis points.
  - **Parameters**
    * `axis` – an `Axis` object, the parent of this node
    * `x` – a `double` array which contains the axis points. This is used to set the "X" attribute.
    * `high` – a `double` array which contains the stock's high prices. This is used to set the "High" attribute.
    * `low` – a `double` array which contains the stock's low prices. This is used to set the "Low" attribute.
    * `close` – a `double` array which contains the stock's closing prices. This is used to set the "Close" attribute.
    * `open` – a `double` array which contains the stock's opening prices This is used to set the "Open" attribute.

### Methods

---

- *getDown*
  `public CandlestickItem` **getDown**`( )`
  - **Description**
    Returns the CandlestickItem for down days.

---

- *getUp*
  `public CandlestickItem` **getUp**`( )`
  - **Description**
    Returns the CandlestickItem for up days.

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* CandlestickItem

A candlestick for the up days or the down days.

CandlestickItem's are created by Candlestick; one for up days and one for down days.

### Declaration

public class com.imsl.chart.CandlestickItem
**extends** com.imsl.chart.Data (page 982)

### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`

– **Description**
Paints this node and all of its children. This is normally called only by the paint method in this node's parent.

– **Parameters**
  * `draw` – the `Draw` object to be painted

# *class* SplineData

A data set created from a Spline.

## Declaration

public class com.imsl.chart.SplineData
**extends** com.imsl.chart.Data (page 982)

## Constructor

• *SplineData*
`public` **SplineData**`( ChartNode` **parent**`, com.imsl.math.Spline` **spline** `)`

– **Description**
Creates a data node from Spline values.

– **Parameters**
  * `parent` – the `ChartNode` parent of this data node
  * `spline` – the `Spline` to be plotted

## Example: SplineData Chart

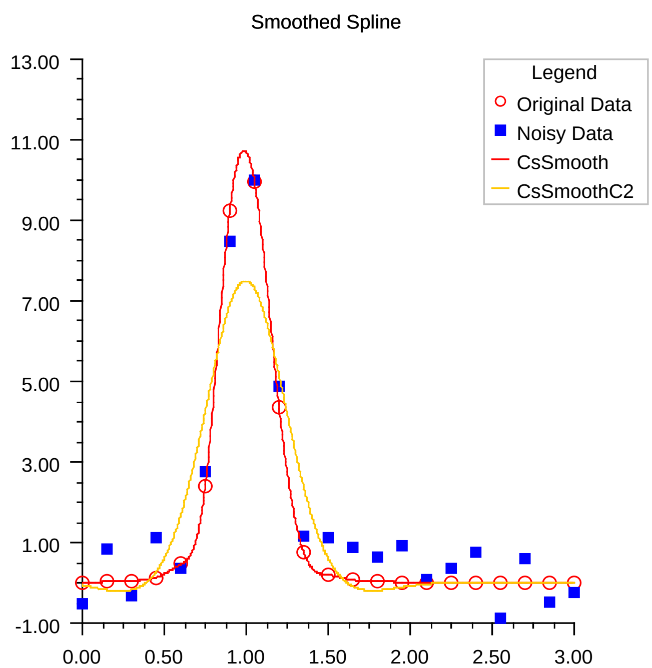
This example makes use of the SplineData class as well as the two spline smoothing classes in the package com.imsl.math. This class can be used either as an applet or as an application.

```
import com.imsl.math.*;
import com.imsl.chart.*;
import com.imsl.stat.Random;
import java.awt.Color;
```

```
public class SplineDataEx1 extends javax.swing.JApplet {
    static private final int nData = 21;
    static private final int nSpline = 100;

    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        chart.getChartTitle().setTitle(new Text("Smoothed Spline"));

        Legend legend = chart.getLegend();
        legend.setTitle(new Text("Legend"));
        legend.setViewport(0.7, 0.9, 0.1, 0.3);
        legend.setPaint(true);

        // Original data
        double xData[] = grid(nData);
        double yData[] = new double[nData];
        for (int k = 0;  k < nData;  k++) {
            yData[k] = f(xData[k]);
        }
        AxisXY axis = new AxisXY(chart);
        Data data = new Data(axis, xData, yData);
        data.setDataType(data.DATA_TYPE_MARKER);
        data.setMarkerType(Data.MARKER_TYPE_HOLLOW_CIRCLE);
        data.setMarkerColor(Color.red);
        data.setTitle("Original Data");

        // Noisy data
        Random random = new Random(123457);
        double yNoisy[] = new double[nData];
        for (int k = 0;  k < nData;  k++) {
            yNoisy[k] = yData[k] + (2.*random.nextDouble()-1.);
        }
        data = new Data(axis, xData, yNoisy);
        data.setDataType(data.DATA_TYPE_MARKER);
```

```
        data.setMarkerType(Data.MARKER_TYPE_FILLED_SQUARE);
        data.setMarkerSize(0.75);
        data.setMarkerColor(Color.blue);
        data.setTitle("Noisy Data");

        chartSpline(axis, new CsSmooth(xData, yData), Color.red, "CsSmooth");
        chartSpline(axis, new CsSmoothC2(xData, yData, nData),
            Color.orange, "CsSmoothC2");
    }


    static private void chartSpline(AxisXY axis, Spline spline,
    Color color, String title) {
        Data data = new SplineData(axis, spline);
        data.setDataType(data.DATA_TYPE_LINE);
        data.setLineColor(color);
        data.setTitle(title);
    }


    static private double[] grid(int nData) {
        double xData[] = new double[nData];
        for (int k = 0;  k < nData;  k++) {
            xData[k] = 3.0*k / (double)(nData-1);
        }
        return xData;
    }

    static private double f(double x) {
        return 1.0/(0.1+Math.pow(3.0*(x-1.0),4));
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        SplineDataEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




# *class* Bar

A bar chart.

The class `Bar` has children of class com.imsl.chart.BarItem. The attribute "BarItem" in class `Bar` is set to the `BarItem` array of children.

## Declaration

public class com.imsl.chart.Bar
**extends** com.imsl.chart.Data (page 982)

## Constructors

---

- *Bar*
  `public` **Bar**`(` `AxisXY` **axis** `)`
  - **Description**
    Constructs a bar chart.
  - **Parameters**
    * `axis` – the `AxisXY` parent of this node

---

- *Bar*
  `public` **Bar**`(` `AxisXY` **axis**`,` `double[]` **y** `)`
  - **Description**
    Constructs a simple bar chart using supplied y data.
  - **Parameters**
    * `axis` – the `AxisXY` parent of this node
    * `y` – a `double` array which contains the y data for the simple bar chart

---

- *Bar*
  `public` **Bar**`(` `AxisXY` **axis**`,` `double[][]` **y** `)`
  - **Description**
    Constructs a grouped bar chart using supplied x and y data.
  - **Parameters**
    * `axis` – the `AxisXY` parent of this node
    * `y` – a `double` array which contains the y data for the grouped bar chart. The first index refers to the group and the second refers to the x position.

---

- *Bar*
  `public` **Bar**`(` `AxisXY` **axis**`,` `double[][][]` **y** `)`
  - **Description**
    Constructs a stacked, grouped bar chart using supplied y data.
  - **Parameters**
    * `axis` – the `AxisXY` parent of this node
    * `y` – a `double` array which contains the y data for the stacked, grouped bar chart. The first index refers to the stack, the second refers to the group and the third refers to the x position.

---

- *Bar*
  `public` **Bar**`(` `AxisXY` **axis**`,` `double[]` **x**`,` `double[]` **y** `)`

– **Description**
Constructs a simple bar chart using supplied x and y data.

– **Parameters**
  * `axis` – the `AxisXY` parent of this node
  * `x` – a `double` array which contains the x data for the simple bar chart
  * `y` – a `double` array which contains the y data for the simple bar chart

---

- *Bar*
`public` **`Bar`**`( AxisXY` **`axis`**`, double[]` **`x`**`, double[][]` **`y`** `)`

  – **Description**
  Constructs a grouped bar chart using supplied x and y data.

  – **Parameters**
    * `axis` – the `AxisXY` parent of this node
    * `x` – a `double` array which contains the x data for the grouped bar chart
    * `y` – a `double` array which contains the y data for the grouped bar chart. The first index refers to the group and the second refers to the x position.

---

- *Bar*
`public` **`Bar`**`( AxisXY` **`axis`**`, double[]` **`x`**`, double[][][]` **`y`** `)`

  – **Description**
  Constructs a stacked, grouped bar chart using supplied x and y data.

  – **Parameters**
    * `axis` – the `AxisXY` parent of this node
    * `x` – a `double` array which contains the x data for the stacked, grouped bar chart
    * `y` – a `double` array which contains the y data for the stacked, grouped bar chart. The first index refers to the "stack", the second refers to the group and the third refers to the x position.

## Methods

---

- *dataRange*
`public void` **`dataRange`**`( double[]` **`range`** `)`

  – **Description**
  Overrides `Data.dataRange`.

  – **Parameters**
    * `range` – a `double` array which contains the new range

---

- *getBarData*
  `public double[][][]` **getBarData**`( )`
  - **Description**
    Returns the “BarData” attribute.
  - **Returns** – a `BarData`[][][] value

---

- *getBarSet*
  `public BarSet[][]` **getBarSet**`( )`
  - **Description**
    Returns the `BarSet` object.
  - **Returns** – a `BarSet`[][] value

---

- *getBarSet*
  `public BarSet` **getBarSet**`( int` **group** `)`
  - **Description**
    Returns the `BarSet` object. The group index is assumed to be zero. This method is most useful for charts with only a single group.
  - **Parameters**
    * `group` – an `int` which specifies the group index
  - **Returns** – a `BarSet`[][] value

---

- *getBarSet*
  `public BarSet` **getBarSet**`( int` **stack**`, int` **group** `)`
  - **Description**
    Returns the `BarSet` object.
  - **Parameters**
    * `stack` – an `int` which specifies the stack index
    * `group` – an `int` which specifies the group index
  - **Returns** – a `BarSet`[][] value

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setBarData*
  `public void` **setBarData**`( double[][][]` **value** `)`
  - **Description**
    Convenience routine to set the “BarData” attribute.
  - **Parameters**
    * `value` – a `BarData`[][][] array of objects that make up this bar chart. The first index refers to the “stack”, the second refers to the group and the third refers to the x position.

---

- *setLabels*
  `public void` **setLabels**`( java.lang.String[]` **labels** `)`
  - **Description**
    Sets up an axis with bar labels. This turns off the tick marks and sets the BarType attribute. It also turns off autoscaling for the axis and sets its Window and Number and Ticks attribute as appropriate for a labeled bar chart. The existing value of the BarType attribute is used to determine the axis to be modified.
  - **Parameters**
    * `labels` – a `String` array with which to label the axis. The number of labels must equal the number of items.

---

- *setLabels*
  `public void` **setLabels**`( java.lang.String[]` **labels**`, int` **type** `)`
  - **Description**
    Sets up an axis with bar labels. This turns off the tick marks and sets the “BarType” attribute. It also turns off autoscaling for the axis and sets its “Window”, “Number” and “Ticks” attributes as appropriate for a labeled bar chart.
  - **Parameters**
    * `labels` – a `String` array with which to label the axis. The number of labels must equal the number of items.
    * `type` – an `int` which specifies the BarType. Legal values are `BAR_TYPE_VERTICAL` or `BAR_TYPE_HORIZONTAL`. This determines the axis to be modified.

### Example: Stacked Bar Chart

A stacked bar chart is constructed in this example. Bar labels and colors are set and axis labels are set. This class can be used either as an applet or as an application.

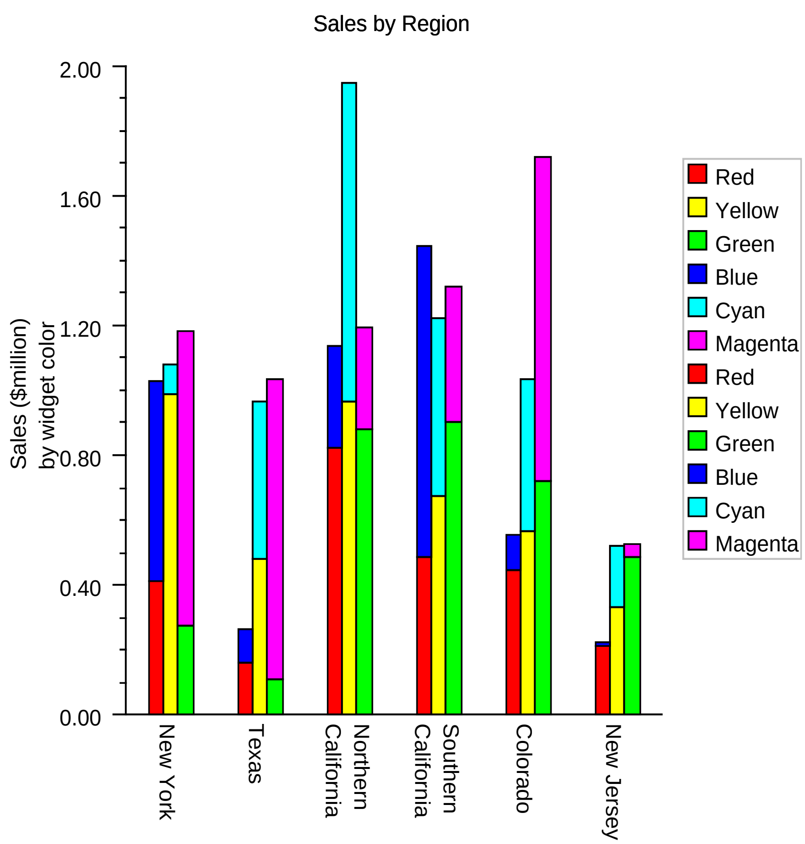
```
import com.imsl.chart.*;
```

```
import com.imsl.stat.Random;
import java.awt.Color;

public class BarEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        AxisXY axis = new AxisXY(chart);

        int nStacks = 2;
        int nGroups = 3;
        int nItems = 6;

        // Generate some random data
        Random r = new Random(123457);
        double x[] = new double[nItems];
        double y[][][] = new double[nStacks][nGroups][nItems];
        double dx = 0.5*Math.PI/(x.length-1);
        for (int istack = 0; istack < y.length; istack++) {
            for (int jgroup = 0; jgroup < y[istack].length; jgroup++) {
                for (int kitem = 0; kitem < y[istack][jgroup].length;
                kitem++) {
                    y[istack][jgroup][kitem] = r.nextDouble();
                }
            }
        }

        // Create an instance of a Bar Chart
        Bar bar = new Bar(axis, y);

        // Set the Bar Chart Title
        chart.getChartTitle().setTitle("Sales by Region");

        // Set the fill outline type;
        bar.setFillOutlineType(Bar.FILL_TYPE_SOLID);
```

```
        // Set the Bar Item fill colors
        bar.getBarSet(0,0).setFillColor(Color.red);
        bar.getBarSet(0,1).setFillColor(Color.yellow);
        bar.getBarSet(0,2).setFillColor(Color.green);
        bar.getBarSet(1,0).setFillColor(Color.blue);
        bar.getBarSet(1,1).setFillColor(Color.cyan);
        bar.getBarSet(1,2).setFillColor(Color.magenta);

        chart.getLegend().setPaint(true);
        bar.getBarSet(0,0).setTitle("Red");
        bar.getBarSet(0,1).setTitle("Yellow");
        bar.getBarSet(0,2).setTitle("Green");
        bar.getBarSet(1,0).setTitle("Blue");
        bar.getBarSet(1,1).setTitle("Cyan");
        bar.getBarSet(1,2).setTitle("Magenta");

        // Setup the vertical axis for a labeled bar chart.
        String labels[] = {
            "New York",
            "Texas",
            "Northern\nCalifornia",
            "Southern\nCalifornia",
            "Colorado",
            "New Jersey"
        };
        bar.setLabels(labels, bar.BAR_TYPE_VERTICAL);

        // Set the text angle
        axis.getAxisX().getAxisLabel().setTextAngle(270);

        // Set the Y axis title
        axis.getAxisY().getAxisTitle().setTitle("Sales ($million)\nby " +
        "widget color");
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        BarEx1.setup(frame.getChart());
        frame.show();
    }
```

}

## Output




# *class* **BarItem**

A single bar in a bar chart.

## Declaration

public class com.imsl.chart.BarItem
**extends** com.imsl.chart.Data (page 982)

### Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`
  - **Description**
    Overides `Data.dataRange`.
  - **Parameters**
    * `range` – a `double` array which contains the new range

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

## *class* BarSet

A set of bars in a bar chart.

A `BarSetis created by Bar and contains a collection of BarItems. Bar creates a BarSet for each stack-group combination. Each BarSet contains the BarItems for that combination. Normally all of the BarItems in a BarSet have the same color, title, etc.`

### Declaration

public class com.imsl.chart.BarSet
**extends** com.imsl.chart.ChartNode (page 920)

### Methods

---

- *dataRange*
  `public void` **dataRange**`( double[]` **range** `)`

- *getBarItem*
  `public BarItem[]` **getBarItem**`( )`

  – **Description**
    Returns an array of `BarItems`. This is the collection of all `BarItems` contained in this bar group.
  – **Returns** – a `BarItem` array

---

- *getBarItem*
  `public BarItem` **getBarItem**`( int` **index** `)`

  – **Description**
    Returns the `BarItem` given the index.
  – **Parameters**
    * `index` – an `int` which specifies the index
  – **Returns** – a `BarItem` associated with the specified index

---

- *paint*
  `public abstract void` **paint**`( Draw` **draw** `)`

  – **Description copied from ChartNode** (**page 920**)
    Paints this node and all of its children.
  – **Parameters**
    * `draw` – the `Draw` object to be painted

# *class* Pie

A pie chart.

The angle of the first slice is determined by the attribute "Reference".

The Pie class is an Axis, because it defines its own mapping to device space.

## Declaration

public class com.imsl.chart.Pie
**extends** com.imsl.chart.Axis (page 960)

## Constructors

- *Pie*
  `public` **Pie**`( Chart` **chart** `)`
  - **Description**
    Constructs a Pie chart object. The “Viewport” attribute for this node is set to [0.2,0.8] by [0.2,0.8].
  - **Parameters**
    * `chart` – the `Chart` parent of this node

- *Pie*
  `public` **Pie**`( Chart` **chart**`, double[]` **y** `)`
  - **Description**
    Constructs a Pie chart object with a specified number of slices. An array of y.length PieSlice nodes are created as children of this node and this array is used to define the attribute “PieSlice” in this node. The “Viewport” attribute for this node is set to [0.2,0.8] by [0.2,0.8].
  - **Parameters**
    * `chart` – the `Chart` parent of this node
    * `y` – a `double` array which contains the values for the pie chart

## Methods

- *getPieSlice*
  `public PieSlice[]` **getPieSlice**`( )`
  - **Description**
    Returns the PieSlice objects.
  - **Returns** – a `PieSlice` array of PieSlice objects

- *getPieSlice*
  `public PieSlice` **getPieSlice**`( int` **index** `)`
  - **Description**
    Returns a specified PieSlice.
  - **Parameters**
    * `index` – an `int`, the 0-based index of the pie slice to return

– **Returns** – a `PieSlice` array of PieSlice objects

---

- *mapDeviceToUser*
  `public void` **mapDeviceToUser**`( int` **devX**`, int` **devY**`, double[]` **userXY** `)`

  – **Description**
  Maps the device coordinates to user coordinates.

  – **Parameters**
  * `devX` – an `int` which specifies the device x-coordinate
  * `devY` – an `int` which specifies the device y-coordinate
  * `userXY` – an `int`[2] array in which the the user coordinates are returned.

---

- *mapUserToDevice*
  `public void` **mapUserToDevice**`( double` **userX**`, double` **userY**`, int[]` **devXY** `)`

  – **Description**
  Maps the user coordinates (userX,userY) to the device coordinates devXY.

  – **Parameters**
  * `userX` – a `double` which specifies the user x-coordinate
  * `userY` – a `double` which specifies the user y-coordinate
  * `devXY` – an `int`[2] array in which the device coordinates are returned.

---

- *setData*
  `public PieSlice[]` **setData**`( double[]` **y** `)`

  – **Description**
  Changes the data in a Pie chart object.

  – **Parameters**
  * `y` – a `double` array which contains the values for the pie chart.

  – **Returns** – A `PieSlice` array containing the updated PieSlice. If the number of slices is unchanged then the existing pie slice array, defined by the attribute "PieSlice" in this node, is reused. If the number is different, a new array is allocated, using the existing PieSlice elements to initialize the new array.

---

- *setupMapping*
  `public void` **setupMapping**`( )`

  – **Description**
  Initializes the mappings between user and coordinate space. This must be called whenever the screen size, the window or the viewport may have changed. Generally, it is safest to call this each time the chart is repainted.

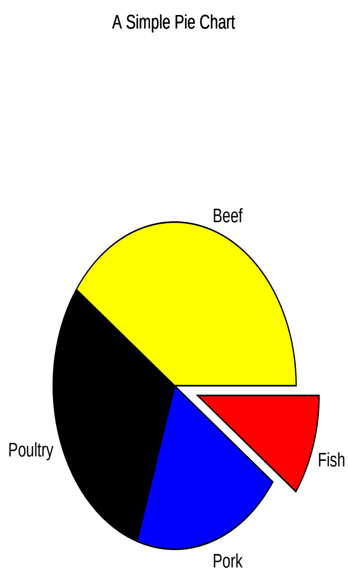
### Example: Pie Chart

A simple Pie chart is constructed in this example. Pie slice labels and colors are set and one pie slice is exploded from the center. This class extends JFrameChart, which manages the window.

```
import com.imsl.chart.*;
import java.awt.Color;
import java.applet.Applet;

public class PieEx1 extends javax.swing.JApplet {
    private JPanelChart panel;

    public void init() {
        Chart chart = new Chart(this);
        panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        // Create an instance of a Pie Chart
        double y[] = {10., 20., 30., 40.};
        Pie pie = new Pie(chart, y);

        // Set the Pie Chart Title
        chart.getChartTitle().setTitle("A Simple Pie Chart");

        // Set the colors of the Pie Slices
        PieSlice[] slice = pie.getPieSlice();
        slice[0].setFillColor(Color.red);
        slice[1].setFillColor(Color.blue);
        slice[2].setFillColor(Color.black);
        slice[3].setFillColor(Color.yellow);

        // Set the Pie Slice Labels
        pie.setLabelType(pie.LABEL_TYPE_TITLE);
        slice[0].setTitle("Fish");
        slice[1].setTitle("Pork");
        slice[2].setTitle("Poultry");
        slice[3].setTitle("Beef");

        // Explode a Pie Slice
```

```
        slice[0].setExplode(0.2);
    }


    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        PieEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




## *class* PieSlice

One wedge of a pie chart.

com.imsl.chart.Pie creates PieSlice objects as its children, one per pie wedge. A specific slice can be retrieved using the method getPieSlice(int). All of the slices can be retrieved using the method getPieSlice().

The drawing of the slice is controlled by the fill attributes in this node.

## Declaration

public class com.imsl.chart.PieSlice
**extends** com.imsl.chart.Data (page 982)

## Methods

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.

---

- *setAngles*
  `protected void` **setAngles**`( double` **angleA**`, double` **angleB** `)`
  - **Description**
    Sets the angles, in degrees, that determine the extent of this slice.
  - **Parameters**
    * `angleA` – is the angle, in degrees, at which the slice begins
    * `angleB` – is the angle, in degrees, at which the slice ends

# *class* Polar

This Axis node is used for polar charts. In a polar plot, the (x,y) coordinates in Data nodes are interpreted as (r,theta) values.

## Declaration

public class com.imsl.chart.Polar
**extends** com.imsl.chart.Axis (page 960)

## Constructor

- *Polar*
  `public` **Polar**`( Chart` **chart** `)`
  - **Description**
    Create an AxisPolar.
  - **Parameters**
    * `chart` – a `Chart` object, the parent of this node

## Methods

- *getAxisR*
  `public AxisR` **getAxisR**`( )`
  - **Description**
    Return the radius axis node.
  - **Returns** – the `AxisR` radius axis node

- *getAxisTheta*
  `public AxisTheta` **getAxisTheta**`( )`
  - **Description**
    Return the angular axis node.
  - **Returns** – the `AxisTheta` axis node

- *getGridPolar*
  `public GridPolar` **getGridPolar**`( )`
  - **Description**
    Returns the grid.
  - **Returns** – the grid, a `GridPolar` object

- *mapDeviceToUser*
  `public void` **mapDeviceToUser**`( int` **devX**`, int` **devY**`, double[]` **userRT** `)`
  - **Description**
    Map the device coordinates to polar coordinates.
  - **Parameters**

* `devX` – an `int`, the device x-coordinate
* `devY` – an `int`, the device y-coordinate
* `userRT` – a `double`[2] array in which the user coordinates, (radius,theta), are returned.

---

- *mapUserToDevice*
  `public void` **mapUserToDevice**`( double` **userRadius**`, double` **userTheta**`, int[]` **devXY** `)`

  - **Description**
    Map the polar coordinates (userRadius,userAngle) to the device coordinates devXY.
  - **Parameters**
    * `userRadius` – a `double`, the user radius coordinate
    * `userTheta` – a `double`, the user angle coordinate
    * `devXY` – an `int`[2] array in which the device coordinates are returned.

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`

  - **Description**
    Paints this node and all of its children. This is normally called only by the paint method in this node's parent.
  - **Parameters**
    * `draw` – the `Draw` object to be painted

---

- *setupMapping*
  `public void` **setupMapping**`( )`

  - **Description**
    Initializes the mappings between user and coordinate space. This must be called whenever the screen size, the window or the viewport may have changed.

## *class* Heatmap

`Heatmap` creates a chart from a two-dimensional array of double precision values or java.awt.Color values. Optionally, each cell in the heatmap can be labeled.

If the input is a two-dimensional array of `double` values then a `Colormap` object is used to map the real values to colors.

## Declaration

public class com.imsl.chart.Heatmap
**extends** com.imsl.chart.Data (page 982)

## Inner Class

### *class* Heatmap.Legend

A legend for use with a heatmap.

This `Legend` should be used with `heatmap`s, rather than the usual chart legend.

#### Declaration

public class com.imsl.chart.Heatmap.Legend
**extends** com.imsl.chart.AxisXY (page 962)

#### Method

---

- *paint*
  `public void` **paint**`( Draw` **draw** `)`
  - **Description**
    Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.
  - **Parameters**
    * `draw` – The `Draw` object to be painted.

## Constructors

---

- *Heatmap*
  `public` **Heatmap**`( AxisXY` **axis**`, double` **xmin**`, double` **xmax**`, double` **ymin**`, double` **ymax**`, java.awt.Color[][]` **color** `)`
  - **Description**
    Creates a `Heatmap` from an array of `Color` values.
  - **Parameters**

* `axis` – An `AxisXY` object, the parent of this node.
* `xmin` – The minimum $x$-value of the color data.
* `xmax` – The maximum $x$-value of the color data.
* `ymin` – The minimum $y$-value of the color data.
* `ymax` – The maximum $y$-value of the color data.
* `color` – A two-dimensional `Color` array of the color values. The value of `color[0][0]` is the color of the cell whose lower left corner is (`xmin`, `ymin`).

---

- *Heatmap*

  `public` **Heatmap**`( AxisXY` **axis**`, double` **xmin**`, double` **xmax**`, double` **ymin**`, double` **ymax**`, double` **zmin**`, double` **zmax**`, double[][]` **data**`, Colormap` **colormap** `)`

  – **Description**

  Creates a `Heatmap` from an array of `double` values and a `Colormap`.

  – **Parameters**

  * `axis` – An `AxisXY` object, the parent of this node.
  * `xmin` – The minimum $x$-value of the color data.
  * `xmax` – The maximum $x$-value of the color data.
  * `ymin` – The minimum $y$-value of the color data.
  * `ymax` – The maximum $y$-value of the color data.
  * `zmin` – The data value that corresponds to the initial ($t$=0) value in the `Colormap`.
  * `zmax` – The data value that corresponds to the final ($t$=1) value in the `Colormap`.
  * `data` – A two-dimensional `double` array containing the data values. The $x$-interval (`xmin` , `xmax`) is uniformly divided and mapped into the first index of `data`. The $y$-interval (`ymin`, `ymax`) is uniformly divided and mapped into the second index of `data`. So, the value of `data[0][0]` is used to determine the color of the cell whose lower left corner is (`xmin`,`ymin` ).
  * `colormap` – Maps the values in `data` to colors. If a cell has a data value equal to $t$ then its color is the value of the colormap at $s$, where $s = \frac{t-\mathrm{zmin}}{\mathrm{zmax}-\mathrm{zmin}}$.

## Methods

---

- *dataRange*

  `public void` **dataRange**`( double[]` **range** `)`

  – **Description**

  Update the data range. `range = {xmin,xmax,ymin,ymax}` The entries in `range` are updated to reflect the extent of the data in this node. `range` is an input/output variable. Its value should be updated only if the data in this node is outside the range already in the array.

– **Parameters**
  * `range` – A array containing the updated `range = {xmin,xmax,ymin,ymax}`.

---

• *getColormap*
`public Colormap` **getColormap**`( )`

– **Description**
Returns the value of the "Colormap" attribute. This is the `Colormap` associated with this `Heatmap`.

– **Returns** – The `Colormap` value of the "Colormap" attribute, if defined. Otherwise, `null` is returned.

---

• *getHeatmapLabels*
`public Text[][]` **getHeatmapLabels**`( )`

– **Description**
Returns the value of the "HeatmapLabels" attribute.

– **Returns** – A two-dimensional array of com.imsl.chart.Text objects that are the value of the "HeatmapLabels" attribute, if defined. Otherwise, `null` is returned.

---

• *getHeatmapLegend*
`public Heatmap.Legend` **getHeatmapLegend**`( )`

– **Description**
Returns the heatmap legend.
By default, the legend is not drawn because its "Paint" attribute is set to `false`. To show the legend set "Paint" to `true`, .i.e.,
`contour.getContourLegend().setPaint(true);`

– **Returns** – The `Legend` object associated with the `Heatmap`.

---

• *paint*
`public void` **paint**`( Draw` **draw** `)`

– **Description**
Paints this node and all of its children. This is normally called only by the `paint` method in this node's parent.

– **Parameters**
  * `draw` – The `Draw` object to be painted.

---

• *setColormap*
`public void` **setColormap**`( Colormap` **colorMap** `)`

– **Description**
Sets the value of the "Colormap" attribute. This is the `Colormap` associated with this `Heatmap`.

– **Parameters**
  * `colorMap` – The `Colormap` object's "ColorMap" value.

---

- *setHeatmapLabels*
`public void` **setHeatmapLabels**`( java.lang.String[][]` **labels** `)`

  – **Description**
  Sets the value of the "HeatmapLabels" attribute. The value of the "HeatmapLabels" attribute is a two dimensional array of `Text` objects. Each `Text` object is created from the corresponding label value with `TEXT_X_CENTER|TEXT_Y_CENTER` alignment.

  – **Parameters**
    * `labels` – A two-dimensional array of `String` objects used to create the two dimensional array of `Text` objects that is the value of the attribute. The array of labels and the array of `Text` objects have the same shape.

---

- *setHeatmapLabels*
`public void` **setHeatmapLabels**`( Text[][]` **labels** `)`

  – **Description**
  Sets the value of the "HeatmapLabels" attribute.

  – **Parameters**
    * `labels` – A two-dimensional array of com.imsl.chart.Text objects that are used to set the "HeatmapLabels" attribute.

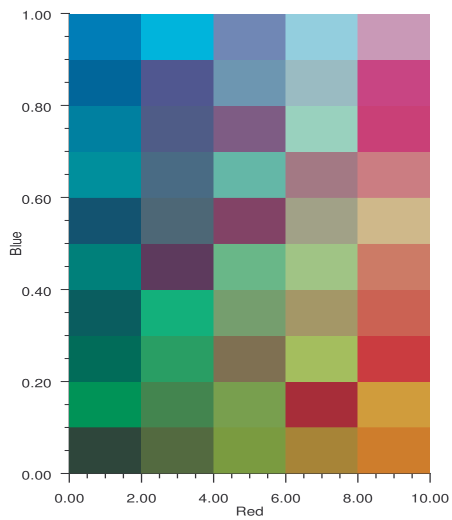
### Example: Heatmap from Color array

A 5 by 10 array of `Color` objects is created by linearly interpolating red along the x-axis, blue along the y-axis and mixing in a random amount of green. The data range is set to [0,10] by [0,1].

```
import com.imsl.chart.*;
import java.awt.Color;
import java.util.Random;

public class HeatmapEx1 extends javax.swing.JApplet  {

    public void init() {
        Chart chart = new Chart(this);
        JPanelChart panel = new JPanelChart(chart);
```

```
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        JFrameChart jfc = new JFrameChart();
        AxisXY axis = new AxisXY(chart);

        double xmin = 0.0;
        double xmax = 10.0;
        double ymin = 0.0;
        double ymax = 1.0;

        int nxRed = 5;
        int nyBlue = 10;
        Random random = new Random(123457L);
        Color color[][] = new Color[nxRed][nyBlue];
        for (int i = 0;  i < nxRed;  i++) {
            for (int j = 0;  j < nyBlue;  j++) {
                int r = (int)(255.*i/nxRed);
                int g = random.nextInt(255);
                int b = (int)(255.*j/nyBlue);
                color[i][j] = new Color(r,g,b);
            }
        }
        Heatmap heatmap = new Heatmap(axis, xmin, xmax, ymin, ymax, color);
        axis.getAxisX().getAxisTitle().setTitle("Red");
        axis.getAxisY().getAxisTitle().setTitle("Blue");
    }

    public static void main(String argv[]) throws Exception {
        JFrameChart frame = new JFrameChart();
        HeatmapEx1.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




## Example: Heatmap from Color array

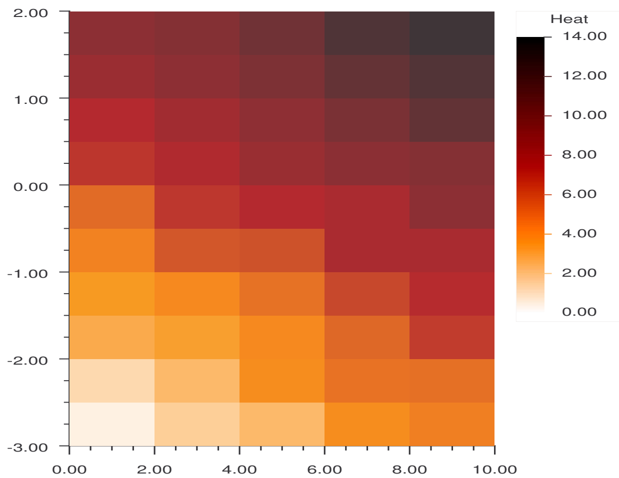
A 5 by 10 data array is created by linearly interpolating from the lower left corner to the upper right corner and adding in a uniform random variable. A red temperature color map is used. This maps the minimum data value to light green and the maximum data value to dark green.

The legend is enabled by setting its paint attribute to true.

```
import com.imsl.chart.*;
import java.awt.Color;
import java.util.Random;

public class HeatmapEx2 extends javax.swing.JApplet  {

    public void init() {
        Chart chart = new Chart(this);
        JPanelChart panel = new JPanelChart(chart);
```

```
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        JFrameChart jfc = new JFrameChart();
        AxisXY axis = new AxisXY(chart);

        int nx = 5;
        int ny = 10;
        double xmin = 0.0;
        double xmax = 10.0;
        double ymin = -3.0;
        double ymax = 2.0;
        double fmin = 0.0;
        double fmax = nx + ny - 1;

        double data[][] = new double[nx][ny];
        Random random = new Random(123457L);
        for (int i = 0;  i < nx;  i++) {
            for (int j = 0;  j < ny;  j++) {
                data[i][j] = i + j + random.nextDouble();
            }
        }
        Heatmap heatmap = new Heatmap(axis, xmin, xmax, ymin, ymax, 0.0, fmax,
            data, Colormap.RED_TEMPERATURE);
        heatmap.getHeatmapLegend().setPaint(true);
        heatmap.getHeatmapLegend().setTitle("Heat");
    }

    public static void main(String argv[]) throws Exception {
        JFrameChart frame = new JFrameChart();
        HeatmapEx2.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




## Example: Heatmap with Labels

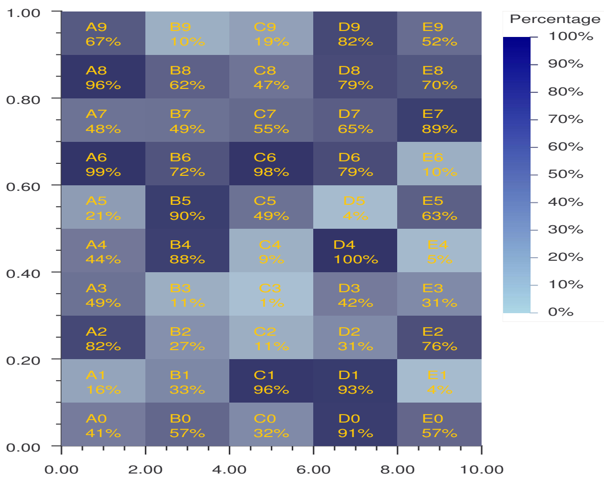
A 5 by 10 array of random data is created and a similarly sized array of strings is also created. These labels contain spreadsheet-like indices and the random data value expressed as a percentage.

The legend is enabled by setting its paint attribute to true. The tick marks in the legend are formatted using the percentage `NumberFormat` object. A title is also set in the legend.

```
import com.imsl.chart.*;
import java.awt.Color;
import java.text.NumberFormat;
import java.util.Random;

public class HeatmapEx3 extends javax.swing.JApplet  {

    public void init() {
        Chart chart = new Chart(this);
        JPanelChart panel = new JPanelChart(chart);
```

```
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    static private void setup(Chart chart) {
        JFrameChart jfc = new JFrameChart();
        AxisXY axis = new AxisXY(chart);

        double xmin = 0.0;
        double xmax = 10.0;
        double ymin = 0.0;
        double ymax = 1.0;

        NumberFormat format = NumberFormat.getPercentInstance();

        int nx = 5;
        int ny = 10;
        double data[][] = new double[nx][ny];
        String labels[][] = new String[nx][ny];
        Random random = new Random(123457L);
        for (int i = 0;  i < nx;  i++) {
            for (int j = 0;  j < ny;  j++) {
                data[i][j] = random.nextDouble();
                labels[i][j] = "ABCDE".charAt(i) + Integer.toString(j) + "\n"
                    + format.format(data[i][j]);
            }
        }
        Heatmap heatmap = new Heatmap(axis, xmin, xmax, ymin, ymax, 0.0, 1.0,
            data, Colormap.BLUE);
        heatmap.setHeatmapLabels(labels);
        heatmap.setTextColor("orange");
        heatmap.getHeatmapLegend().setPaint(true);
        heatmap.getHeatmapLegend().setTextFormat(format);
        heatmap.getHeatmapLegend().setTitle("Percentage");
    }

    public static void main(String argv[]) throws Exception {
        JFrameChart frame = new JFrameChart();
        HeatmapEx3.setup(frame.getChart());
        frame.show();
    }
}
```

## Output




# *interface* Colormap

`Colormap`s are mappings from the unit interval to `Color`s. They are a one-dimensional parameterized path through the color cube.

## Declaration

```
public interface com.imsl.chart.Colormap
```

## Fields

---

- `Colormap` **RED**
  - Linear red colormap.

- `Colormap` **GREEN**
  - Linear green colormap.
- `Colormap` **BLUE**
  - Linear blue colormap.
- `Colormap` **BW_LINEAR**
  - Black and white (grayscale) colormap.
- `Colormap` **BLUE_WHITE**
  - Blue/white colormap.
- `Colormap` **GREEN_RED_BLUE_WHITE**
  - Green/red/blue/white colormap.
- `Colormap` **RED_TEMPERATURE**
  - Red temperature colormap.
- `Colormap` **BLUE_GREEN_RED_YELLOW**
  - Blue/green/red/yellow colormap.
- `Colormap` **STANDARD_GAMMA**
  - Standard gamma colormap.
- `Colormap` **PRISM**
  - Prism colormap.
- `Colormap` **RED_PURPLE**
  - Red/purple colormap.
- `Colormap` **GREEN_WHITE_LINEAR**
  - Linear green/white colormap.
- `Colormap` **GREEN_WHITE_EXPONENTIAL**
  - Exponential green/white colormap.
- `Colormap` **GREEN_PINK**
  - Green/pink colormap.
- `Colormap` **BLUE_RED**
  - Blue/red colormap.
- `Colormap` **SPECTRAL**

– Spectral colormap.

- `Colormap` **WHITE_BLUE_LINEAR**
  – Linear blue/white colormap.

### Method

- *color*
  `java.awt.Color` **color**`( double t )`
  - **Description**
    Maps the parameterization interval [0,1] into `Colors`.
  - **Parameters**
    * `t` – A parameter value in the interval [0,1].
  - **Returns** – A `Color` value corresponding to `t`.
  - **Throws**
    * `java.lang.IllegalArgumentException` – is thrown if `t` is outside of the range [0,1]

# Chapter 25

# Neural Nets

Classes

## Usage Notes

### Neural Networks - An Overview

Today, neural networks are used to solve a wide variety of problems, some of which have been solved by existing statistical methods, and some of which have not. These applications fall into one of the following three categories:

- *Forecasting*: predicting one or more quantitative outcomes from both quantitative and categorical input data,
- *Classification*: classifying input data into one of two or more categories, or
- *Statistical pattern recognition*: uncovering patterns, typically spatial or temporal, among a set of variables.

Forecasting, pattern recognition and classification problems are not new. They existed years before the discovery of neural network solutions in the 1980's. What is new is that neural networks provide a single framework for solving so many traditional problems and, in some cases, extend the range of problems that can be solved.

Traditionally, these problems have been solved using a variety of well known statistical methods:

- linear regression and general least squares,
- logistic regression and discrimination,
- principal component analysis,
- discriminant analysis,
- $k$-nearest neighbor classification, and
- ARMA and non-linear ARMA time series forecasts.

In many cases, simple neural network configurations yield the same solution as many traditional statistical applications.For example, a single-layer, feed-forward neural network with linear activation for its output perceptron is equivalent to a general linear regression fit. Neural networks can provide more accurate and robust solutions for problems where traditional methods do not completely apply.

Mandic and Chambers (2001) point out that traditional methods for time series forecasting are unsuitable when a time series:

- is non-stationary,
- has large amounts of noise, such as a biomedical series, or
- is too short.

ARIMA and other traditional time series approaches can produce poor forecasts when one or more of the above problems exist. The forecasts of ARMA and non-linear ARMA (NARMA) depend heavily upon key assumptions about the model or underlying relationship between the output of the series and its patterns.

Neural networks, on the other hand, adapt to changes in a non-stationary series and can produce reliable forecasts even when the series contains a good deal of noise or when only a short series is available for training. Neural networks provide a single tool for solving many problems traditionally solved using a wide variety of statistical tools and for solving problems when traditional methods fail to provide an acceptable solution.

Although neural network solutions to forecasting, pattern recognition, and classification problems can be very different, they are always the result of computations that proceed from the network inputs to the network outputs. The network inputs are referred to as *patterns*, and outputs are referred to as *classes*. Frequently the flow of these computations is in one direction, from the network input patterns to its outputs. Networks with forward-only flow are referred to as feed-forward networks.

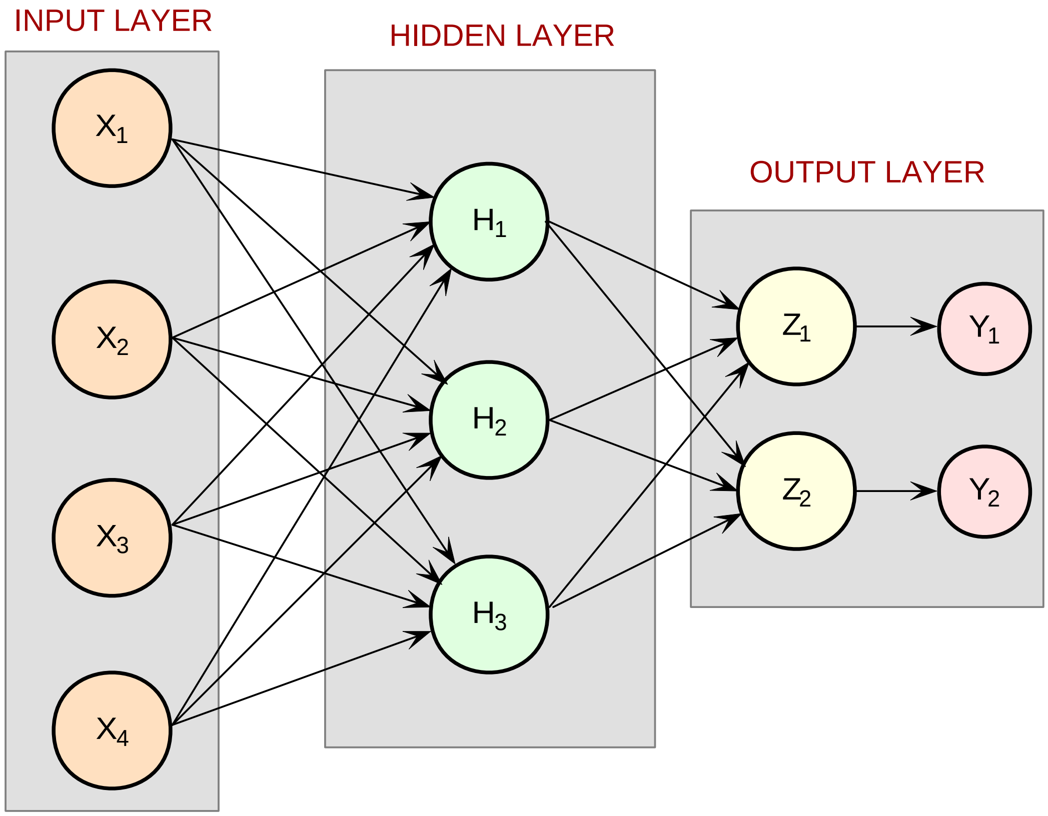



**Figure 1. A 2-layer, Feed-Forward Network with 4 Inputs and 2 Outputs**

Other networks, such as recurrent neural networks, allow data and information to flow in both directions, see Mandic and Chambers (2001).

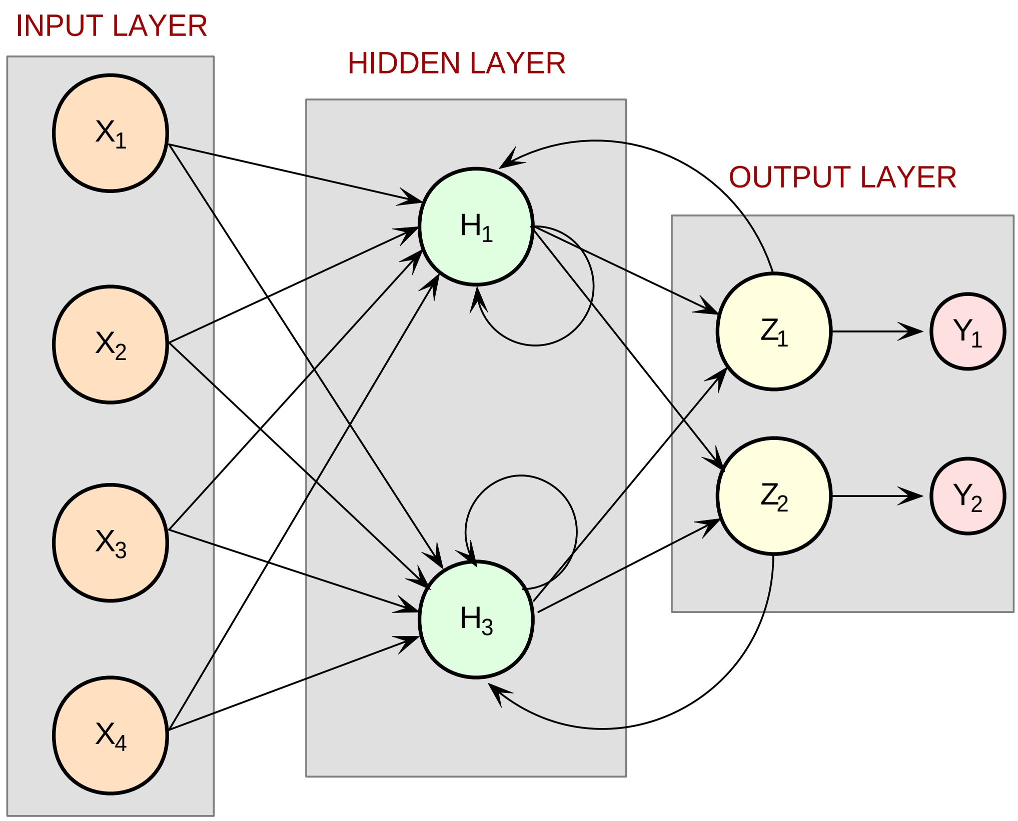



**Figure 2. A Recurrent Neural Network with 4 Inputs and 2 Outputs**

A neural network is defined not only by its architecture and flow, or interconnections, but also by computations used to transmit information from one node or input to another node. These computations are determined by network weights. The process of fitting a network to existing data to determine these weights is referred to as *training* the network, and the data used in this process are referred to as *patterns*. Individual network inputs are referred to as *attributes* and outputs are referred to as *classes*. Many terms used to describe neural networks are synonymous to common statistical terminology.

**Table 1. Synonyms between Neural Network and Common Statistical Terminology**

| Neural Network Terminology | Traditional Statistical Terminology | Description |
|---|---|---|
| Training | Model Fitting | Estimating unknown parameters or coefficients in the analysis. |
| Patterns | Cases or Observations | A single observation of all input and output variables. |
| Attributes | Independent variables | Inputs to the network or model. |
| Classes | Dependent variables | Outputs from the network or model calculations. |

## Neural Networks – History and Terminology

### The Threshold Neuron

McCulloch and Pitts (1943) wrote one of the first published works on neural networks. In their paper, they describe the threshold neuron as a model for how the human brain stores and processes information.

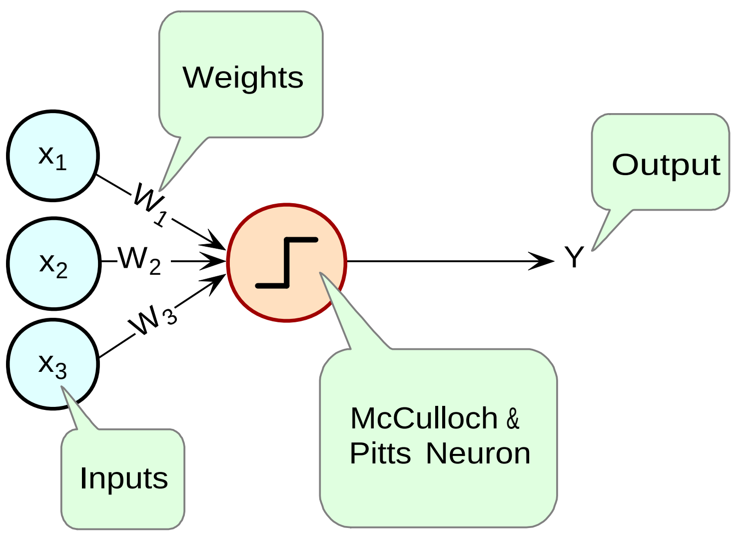



**Figure 3. The McCulloch and Pitts Threshold Neuron**

All inputs to a threshold neuron are combined into a single number, Z, using the following weighted sum:

$$Z = \sum_{i=1}^{m} w_i x_i - \mu$$

where $w_i$ is the weight associated with the *i*th input (attribute) $x_i$. The term $\mu$ in this calculation is referred to as the *bias term*. In traditional statistical terminology, it might be referred to as the *intercept*. The weights and bias terms in this calculation are estimated during network training.

In McCulloch and Pitt's description of the threshold neuron, the neuron does not respond to its inputs unless Z is greater than zero. If Z is greater than zero then the output from this neuron is set equal to 1. If Z is less than zero the output is zero:

$$Y = \begin{cases} 1 & \text{if} Z > 0 \\ 0 & \text{if} Z \leq 0 \end{cases}$$

where Y is the neuron's output.

For years following their 1943 paper, interest in the McCulloch and Pitts neural network was limited to theoretical discussions, such as those of Hebb (1949), about learning, memory, and the brain's structure.

## The Perceptron

The McCulloch and Pitts neuron is also referred to as a threshold neuron since it abruptly changes its output from 0 to 1 when its potential, Z, crosses a threshold. Mathematically, this behavior can be viewed as a step function that maps the neuron's potential, Z, to the neuron's output, Y.

Rosenblatt (1958) extended the McCulloch and Pitts threshold neuron by replacing this step function with a continuous function that maps Z to Y. The Rosenblatt neuron is referred to as the perceptron, and the continuous function mapping Z to Y makes it easier to train a network of perceptrons than a network of threshold neurons.

Unlike the threshold neuron, the perceptron produces analog output rather than the threshold neuron's purely binary output. Carefully selecting the analog function makes Rosenblatt's perceptron differentiable, whereas the threshold neuron is not. This simplifies the training algorithm.

Like the threshold neuron, Rosenblatt's perceptron starts by calculating a weighted sum of its inputs, $Z = \sum_{i=1}^{m} w_i x_i - \mu$. This is referred to as the perceptron's *potential*.

Rosenblatt's perceptron calculates its analog output from its potential. There are many choices for this calculation. The function used for this calculation is referred to as the activation function in Figure 4 below.

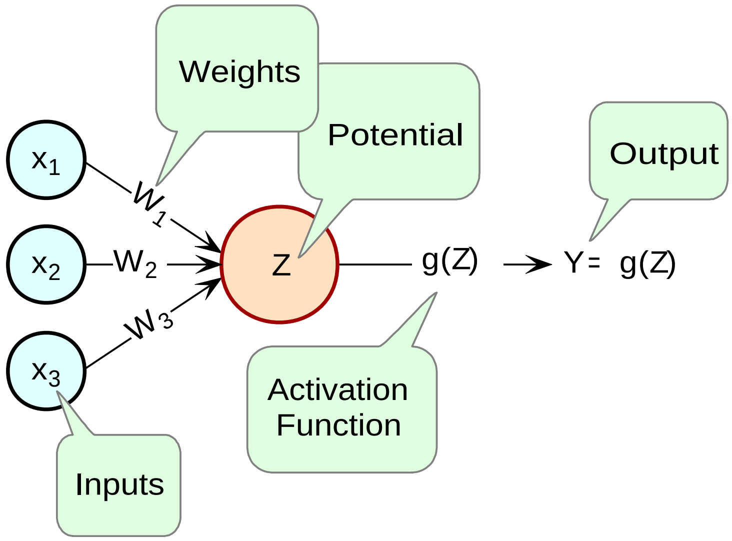



**Figure 4. The Perceptron**

As shown in Figure 4, perceptrons consist of the following five components:

| **Component** | **Example** |
|---|---|
| *Inputs* | $X_1$,$X_2$,$X_3$, |
| *Input Weights* | $W_1$,$W_2$,$W_3$, |
| *Potential* | $Z = \sum_{i=1}^{3} W_i X_i - \mu$, where $\mu$ is a bias correction. |
| *Activation Function* | $g(Z)$ |
| *Output* | $g(Z)$ |

Like threshold neurons, perceptron inputs can be either the initial raw data inputs or the output from another perceptron. The primary purpose of the network training is to estimate the weights associated with each perceptron's potential. The activation function maps this potential to the perceptron's output.

## The Activation Function

Although theoretically any differential function can be used as an activation function, the identity and sigmoid functions are the two most commonly used.

The *identity activation* function, also referred to as a *linear activation* function, is a flow-through mapping of the perceptron's potential to its output:

$$g(Z) = Z$$

Output perceptrons in a forecasting network often use the identity activation function.

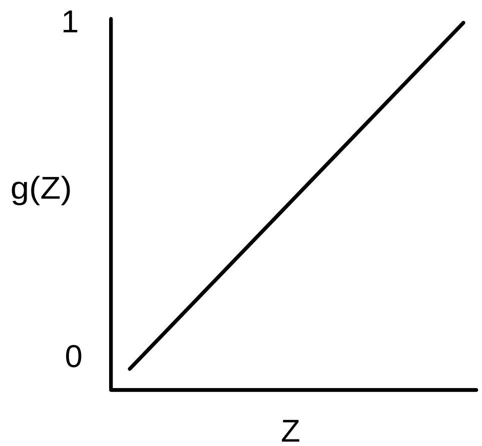



**Figure 5. An Identity (Linear) Activation Function**

If the identity activation function is used throughout the network, then it is easily shown that the network is equivalent to fitting a linear regression model of the form $Y_i = \beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k$, where $x_1, x_2, \cdots, x_k$ are the k network inputs, $Y_i$ is the $i$th network output and $\beta_0, \beta_1, \cdots, \beta_k$ are the coefficients in the regression equation. As a result, it is uncommon to find a neural network with identity activation used in all its perceptrons.

*Sigmoid activation* functions are differentiable functions that map the perceptron's potential to a range of values, such as 0 to 1, i.e., $\mathbb{R}^K \rightarrow \mathbb{R}$ where $K$ is the number of preceptron inputs.

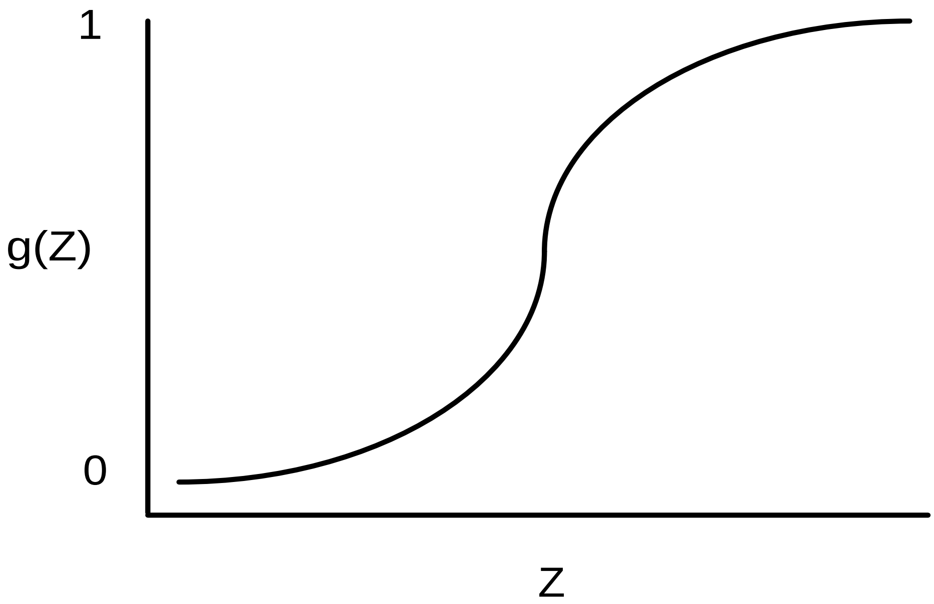



**Figure 6. A Sigmoid Activation Function**

In practice, the most common sigmoid activation function is the logistic function that maps the potential into the range 0 to 1:

$$g(Z) = \frac{1}{1 + e^{-Z}}$$

Since $0 < g(Z) < 1$, the logistic function is very popular for use in networks that output probabilities.

Other popular sigmoid activation functions include:

1. the hyperbolic-tangent $g(Z) = \tanh(Z) = \frac{e^{\alpha Z} - e^{-\alpha Z}}{e^{\alpha Z} + e^{-\alpha Z}}$
2. the arc-tangent $g(Z) = \frac{2}{\pi} \arctan\left(\frac{\pi Z}{2}\right)$, and
3. the squash activation function (Elliott (1993)) $g(Z) = \frac{Z}{1+|Z|}$

It is easy to show that the hyperbolic-tangent and logistic activation functions are linearly related. Consequently, forecasts produced using logistic activation should be close to those produced using hyperbolic-tangent activation. However, one function may be preferred over the other when training performance is a concern. Researchers report that the training time using the hyperbolic-tangent activation function is shorter than using the logistic activation function.

## Network Applications

### Forecasting using Neural Networks

There are many good statistical forecasting tools. Most require assumptions about the relationship between the variables being forecasted and the variables used to produce the forecast, as well as the distribution of forecast errors. Such statistical tools are referred to as *parametric methods*. ARIMA time series models, for example, assume that the time series is stationary, that the errors in the forecasts follow a particular ARIMA model, and that the probability distribution for the residual errors is Gaussian, see Box and Jenkins (1970). If these assumptions are invalid, then ARIMA time series forecasts can be very poor.

Neural networks, on the other hand, require few assumptions. Since neural networks can approximate highly non-linear functions, they can be applied without an extensive analysis of underlying assumptions.

Another advantage of neural networks over ARIMA modeling is the number of observations needed to produce a reliable forecast. ARIMA models generally require 50 or more equally spaced, sequential observations in time. In many cases, neural networks can also provide adequate forecasts with fewer observations by incorporating exogenous, or external, variables in the network's input.

For example, a company applying ARIMA time series analysis to forecast business expenses would normally require each of its departments, and each sub-group within each department to prepare its own forecast. For large corporations this can require fitting hundreds or even thousands of ARIMA models. With a neural network approach, the department and sub-group information could be incorporated into the network as exogenous variables. Although this can significantly increase the network's training time, the result would be a single model for predicting expenses within all departments and sub-departments.

Linear least squares models are also popular statistical forecasting tools. These methods range from simple linear regression for predicting a single quantitative outcome to logistic regression for estimating probabilities associated with categorical outcomes. It is easy to show that simple linear least squares forecasts and logistic regression forecasts are equivalent to a feed-forward network with a single layer. For this reason, single-layer feed-forward networks are rarely used for forecasting. Instead multilayer networks are used.

Hutchinson (1994) and Masters (1995) describe using multilayer feed-forward neural networks for forecasting. Multilayer feed-forward networks are characterized by the forward-only flow of information in the network. The flow of information and computations in a feed-forward network is always in one direction, mapping an M-dimensional vector of inputs to a C-dimensional vector of outputs, i.e., $\mathbb{R}^M \to \mathbb{R}^C$ .

There are many other types of networks without this feed-forward requirement. Information and computations in a recurrent neural network, for example, flows in both directions. Output from one level of a recurrent neural network can be fed back, with some delay, as input into the same network, see Figure 2. Recurrent networks are very useful for time series prediction, see Mandic and Chambers (2001).

### Pattern Recognition using Neural Networks

Neural networks are also extensively used in statistical pattern recognition. Pattern recognition applications that make wide use of neural networks include:

- natural language processing: Manning and Schtze (1999)
- speech and text recognition: Lippmann (1989)
- face recognition: Lawrence, et al. (1997)
- playing backgammon, Tesauro (1990)
- classifying financial news, Calvo (2001).

The interest in pattern recognition using neural networks has stimulated the development of important variations of feed-forward networks. Two of the most popular are:

- Self-Organizing Maps, also called Kohonen Networks, Kohonen (1995),
- and Radial Basis Function Networks, Bishop (1995).

Good mathematical descriptions of the neural network methods underlying these applications are given by Bishop (1995), Ripley (1996), Mandic and Chambers (2001), and Abe (2001). An excellent overview of neural networks, from a statistical viewpoint, is also found in Warner and Misra (1996).

### Neural Networks for Classification

Classifying observations using prior concomitant information is possibly the most popular application of neural networks. Data classification problems abound in business and research. When decisions based upon data are needed, they can often be treated as a neural network data classification problem. Decisions to buy, sell, hold or do nothing with a stock, are decisions involving four choices. Classifying loan applicants as good or bad credit risks, based upon their application, is a classification problem involving two choices. Neural networks are powerful tools for making decisions or choices based upon data.

These same tools are ideally suitable for automatic selection or decision-making. Incoming email, for example, can be examined to separate spam from important email using a neural network trained for this task. A good overview of solving classification problems using multilayer feed-forward neural networks is found in Abe (2001) and Bishop (1995).

There are two popular methods for solving data classification problems using multilayer feed-forward neural networks, depending upon the number of choices (classes) in the classification problem. If the classification problem involves only two choices, then it can be solved using a neural network with one logistic output. This output estimates the probability that the input data belong to one of the two choices.

For example, a multilayer feed-forward network with a single logistic output can be used to determine whether a new customer is credit-worthy. The network's input would consist of information on the applicants credit application, such as age, income, etc. If the network output probability is above some threshold value (such as 0.5 or higher) then the applicant's credit application is approved.

This is referred to as binary classification using a multilayer feed-forward neural network. If more than two classes are involved then a different approach is needed. A popular approach is to assign logistic output perceptrons to each class in the classification problem. The network assigns each input pattern to the class associated with the output perceptron that has the highest probability for that input pattern. However, this approach produces invalid probabilities since the sum of the individual class probabilities for each input is not equal to one, which is a requirement for any valid multivariate probability distribution.

To avoid this problem, the softmax activation function, see Bridle (1990), applied to the network outputs ensures that the outputs conform to the mathematical requirements of multivariate classification probabilities. If the classification problem has C categories, or classes, then each category is modeled by one of the network outputs. If $\mathrm{Z}_i$ is the weighted sum of products between its weights and inputs for the $i$th output, i.e., $Z_i = \sum_j w_{ji} y_{ji}$.

$$\mathrm{softmax_i} = \frac{\mathrm{e}^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}}$$

The softmax activation function ensures that the outputs all conform to the requirements for multivariate probabilities. That is,

$$0 < \mathrm{softmax}_i < 1, \quad \text{for all } i = 1, 2, \ldots, C$$

and

$$\sum_{i=1}^{C} \mathrm{softmax}_i = 1$$

A pattern is assigned to the $i$th classification when $\mathrm{softmax}_i$is the largest among all C classes.

However, multilayer feed-forward neural networks are only one of several popular methods for solving classification problems. Others include:

- Support Vector Machines (SVM Neural Networks), Abe (2001),
- Classification and Regression Trees (CART), Breiman, et al. (1984),
- Quinlan's classification algorithms C4.5 and C5.0, Quinlan (1993), and
- Quick, Unbiased and Efficient Statistical Trees (QUEST), Loh and Shih (1997).

Support Vector Machines are simple modifications of traditional multilayer feed-forward neural networks (MLFF) configured for pattern classification.

### Multilayer Feed-Forward Neural Networks

A multilayer feed-forward neural network is an interconnection of perceptrons in which data and calculations flow in a single direction, from the input data to the outputs. The number of layers in a neural network is the number of layers of perceptrons. The simplest neural network is one with a single input layer and an output layer of perceptrons. The network in Figure 7 illustrates this type of network. Technically this is referred to as a one-layer feed-forward network with two outputs because the output layer is the only layer with an activation calculation.

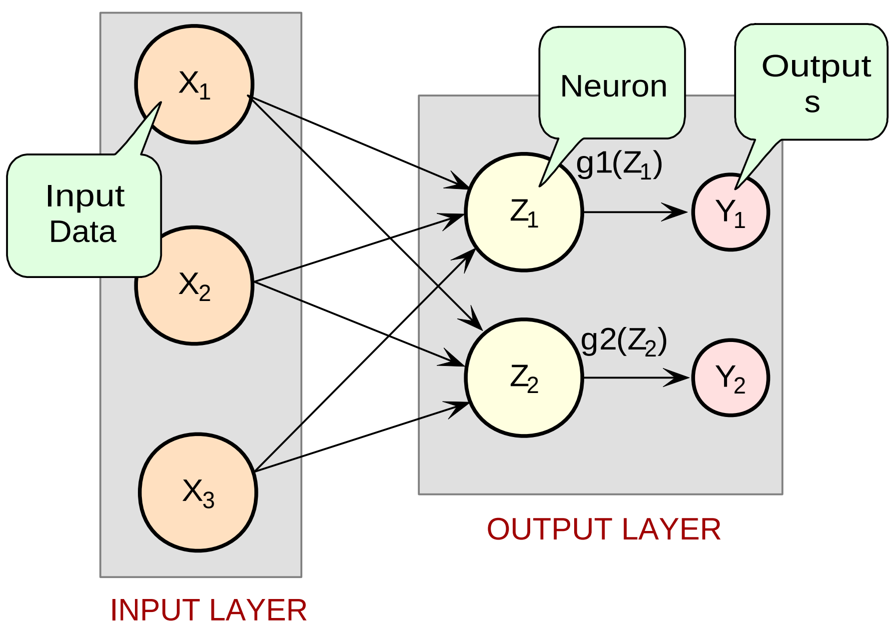



**Figure 7. A Single-Layer Feed-Forward Neural Net**

In this single-layer feed-forward neural network, the networks inputs are directly connected to the output layer perceptrons, $Z_1$ and $Z_2$.

The output perceptrons use activation functions, $g_1$ and $g_2$, to produce the outputs $Y_1$ and $Y_2$

Since

$$Z_1 = \sum_{i=1}^{3} W_{1,i} X_i - \mu_1$$

and

$$Z_2 = \sum_{i=1}^{3} W_{2,i} X_i - \mu_2$$

$$Y_1 = g_1(Z_1) = g_1(\sum_{i=1}^{3} W_{1,i} X_i - \mu_1)$$

and

$$Y_2 = g_2(Z_2) = g_2(\sum_{i=1}^{3} W_{2,i} X_i - \mu_2)$$

When the activation functions $g_1$ and $g_2$ are identity activation functions, a single-layer neural net is equivalent to a linear regression model. Similarly, if $g_1$ and $g_2$ are logistic activation functions, then the single-layer neural net is equivalent to logistic regression. Because of this correspondence between single-layer neural networks and linear and logistic regression, single-layer neural networks are rarely used in place of linear and logistic regression.

The next most complicated neural network is one with two layers. This extra layer is referred to as a hidden layer. In general there is no restriction on the number of hidden layers. However, it has been shown mathematically that a two-layer neural network, such as shown in Figure 1, can accurately reproduce any differentiable function, provided the number of perceptrons in the hidden layer is unlimited.

However, increasing the number of neurons increases the number of weights that must be estimated in the network, which in turn increases the execution time for this network. Instead of increasing the number of perceptrons in the hidden layers to improve accuracy, it is sometimes better to add additional hidden layers, which typically reduces both the total number of network weights and the computational time. However, in practice, it is uncommon to see neural networks with more than two or three hidden layers.

## Neural Network Error Calculations

### Error Calculations for Forecasting

The error calculations used to train a neural network are very important. Many error calculations have been researched, trying to find a calculation with a short training time that is appropriate for the network's application. Typically error calculations are very different depending primarily on the network's application.

For forecasting, the most popular error function is the sum-of-squared errors, or one of its scaled versions. This is analogous to using the minimum least squares optimization criterion in linear regression. Like least squares, the sum-of-squared errors is calculated by looking at the squared difference between what the network predicts for each training pattern and the target value, or observed value, for that pattern. Formally, the equation is the same as one-half the traditional least squares error:

$$E = \tfrac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij} - \hat{t}_{ij}\right)^2$$

where N is the total number of training cases, C is equal to the number of network outputs, $t_{ij}$ is the observed output for the $i$th training case and the $j$th network output, and $\hat{t}_{ij}$ is the network's forecast for that case.

Common practice recommends fitting a different network for each forecast variable. That is, the recommended practice is to use C=1 when using a multilayer feed-forward neural network for forecasting. For classification problems with more than two classes, it is common to associate one output with each classification category, i.e., C=number of classes.

Notice that in ordinary least squares, the sum-of-squared errors is not multiplied by one-half. Although this has no impact on the final solution, it significantly reduces the number of computations required during training.

Also note that as the number of training patterns increases, the sum-of-squared errors increases. As a result, it is often useful to use the root-mean-square (RMS) error instead of the unscaled sum-of-squared errors:

$$E^{RMS} = \frac{\sum\limits_{i=1}^{N}\sum\limits_{j=1}^{C}\left(t_{ij} - \hat{t}_{ij}\right)^2}{\sum\limits_{i=1}^{N}\sum\limits_{j=1}^{C}\left(t_{ij} - \bar{t}\right)^2}$$

where $\bar{t}$ is the average output:

$$\bar{t} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{C} t_{ij}}{N \cdot C}$$

Unlike the unscaled sum-of-squared errors, $E^{RMS}$ does not increase as N increases. The smaller the value of $E^{RMS}$ the closer the network is predicting its targets during training. A value of $E^{RMS} = 0$ indicates that the network is able to predict every pattern exactly. A value of $E^{RMS} = 1$ indicates that the network is predicting the training cases only as well as using the mean of the training cases for forecasting.

Notice that the root-mean-squared error is related to the sum-of-squared error by a simple scale factor:

$$E^{RMS} = \frac{2}{\bar{\bar{t}}} \cdot E$$

Another popular error calculation for forecasting from a neural network is the Minkowski-R error. The sum-of-squared error, $E$, and the root-mean-squared error, $E^{RMS}$, are both theoretically motivated by assuming the noise in the target data is Gaussian. In many cases, this assumption is invalid. A generalization of the Gaussian distribution to other distributions gives the following error function, referred to as the Minkowski-R error:

$$E^{R} = \sum_{i=1}^{N}\sum_{j=1}^{C} \left| t_{ij} - \hat{t}_{ij} \right|^{R}.$$

Notice that $E^R = 2E$ when R=2.

A good motivation for using $E^R$ instead of $E$ is to reduce the impact of outliers in the training data. The usual error measures, $E$ and $E^{RMS}$, emphasize larger differences between the training data and network forecasts since they square those differences. If outliers are expected, then it is better to de-emphasize larger differences. This can be done by using the Minkowski-R error with R=1. When R=1, the Minkowski-R error simplifies to the sum of absolute differences:

$$L = E^{1} = \sum_{i=1}^{N}\sum_{j=1}^{C} \left| t_{ij} - \hat{t}_{ij} \right|.$$

$L$ is also referred to as the Laplacian error. Its name is derived from the fact that it can be theoretically justified by assuming the noise in the training data follows a Laplacian rather than Gaussian distribution.

Of course, similar to $E$, $L$ generally increases when the number of training cases increases. Similar to $E^{RMS}$, a scaled version of the Laplacian error can be calculated using the following formula:

$$L^{RMS} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{C}\left|t_{ij} - \hat{t}_{ij}\right|}{\sum_{i=1}^{N}\sum_{j=1}^{C}\left|t_{ij} - \bar{t}\right|}$$

## Cross-Entropy Error for Binary Classification

As previously mentioned, multilayer feed-forward neural networks can be used for both forecasting and classification applications. Training a forecasting network involves finding the network weights that minimize either the Gaussian or Laplacian distributions, $E$ or $L$ respectively, or equivalently their scaled versions, $E^{RMS}$ or $L^{RMS}$. Although these error calculations can be adapted for use in classification by setting the target classification variable to zeros and ones, this is not recommended. Use of the sum-of-squared and Laplacian error calculations is based on the assumption that the target variable is continuous. In classification applications, the target variable is a discrete random variable with C possible values, where C=number of classes.

A multilayer feed-forward neural network for classifying patterns into one of only two categories is referred to as a binary classification network. It has a single output: the estimated probability that the input pattern belongs to one of the two categories. The probably that it belongs to the other category is equal to one minus this probability, i.e.,

$$P(C_2) = P(\text{not C}_1) = 1 - P(C_1)$$

Binary classification applications are very common. Any problem requiring *yes*/*no* classification is a binary classification application. For example, deciding to sell or buy a stock is a binary classification problem. Deciding to approve a loan application is also a binary classification problem. Deciding whether to approve a new drug or to provide one of two medical treatments are binary classification problems.

For binary classification problems, only a single output is used, C=1. This output represents the probability that the training case should be classified as *yes*. A common choice for the activation function of the output of a binary classification networks is the logistic activation function, which always results in an output in the range 0 to 1, regardless of the perceptron's potential.

One choice for training binary classification network is to use sum-of-squared errors with the class value of *yes* patterns coded as a 1 and the *no* classes coded as a 0, i.e.:

$$t_{ij} = \begin{cases} 1 & \text{if training pattern } i\text{=}yes \\ 0 & \text{if the training pattern } i\text{=}no \end{cases}$$

However, using either the sum-of-squared or Laplacian errors for training a network with these target values assumes that the noise in the training data are Gaussian. In binary classification, the zeros and ones are not Gaussian. They follow the Bernoulli distribution:

$$P(t_i = t) = p^t(1-p)^{1-t}$$

where $p$ is equal to the probability that a randomly selected case belongs to the *yes* class.

Modeling the binary classes as Bernoulli observations leads to the use of the cross-entropy error function described by Hopfield (1987) and Bishop (1995):

$$E^C = -\sum_{i=1}^{N} \left\{ t_i \ln(\hat{t}_i) + (1-t_i)\ln(1-\hat{t}_i) \right\}.$$

where N is the number of training patterns, $t_i$ is the target value for the $i$th case (either 1 or 0), and $\hat{t}_i$ is the network's output for the $i$th case. This is equal to the neural network's estimate of the probability that the $i$th case should be classified as *yes*.

For situations in which the target variable is a probability in the range $0 < t_{ij} < 1$, the value of the cross-entropy at the networks optimum is equal to:

$$E^C_{\text{min}} = -\sum_{i=1}^{N} \{ t_i \ln(t_i) + (1-t_i)\ln(1-t_i) \}$$

Subtracting this from $E^C$ gives an error term bounded below by zero, i.e., $E^{CE} \geq 0$ where:

$$E^{CE} = E^C - E^C_{\text{min}} = -\sum_{i=1}^{N} \left\{ t_i \ln\left[\frac{\hat{t}_i}{t_i}\right] + (1-t_i)\ln\left[\frac{1-\hat{t}_i}{1-t_i}\right] \right\}$$

This adjusted cross-entropy is normally reported when training a binary classification network where $0 < t_{ij} < 1$. Otherwise $E^C$, the non-adjusted cross-entropy error, is used. Small values, values near zero, would indicate that the training resulted in a network with a low error rate and that patterns are being classified correctly most of the time.

## Back-Propagation in Multilayer Feed-Forward Neural Network

Sometimes a multilayer feed-forward neural network is referred to incorrectly as a back-propagation network. The term back-propagation does not refer to the structure or architecture of a network. Back-propagation refers to the method used during network training. More specifically, back-propagation refers to a simple method for calculating the gradient of the network, that is the first derivative of the weights in the network.

The primary objective of network training is to estimate an appropriate set of network weights based upon a training dataset. There are many ways that have been researched for estimating these weights, but they all involve minimizing some error function. In forecasting, the most commonly used error function is the sum of squared errors:

$$E = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij} - \hat{t}_{ij}\right)^2$$

Training uses one of several possible optimization methods to minimize this error term. Some of the more common are: steepest descent, quasi-Newton, conjugant gradient, and many various modifications of these optimization routines.

Back-propagation is a method for calculating the first derivative, or gradient, of the error function required by some optimization methods.It is certainly not the only method for estimating the gradient. However, it is the most efficient. In fact, some will argue that the development of this method by Werbos (1974), Parket (1985), and Rumelhart, Hinton and Williams (1986) contributed to the popularity of neural network methods by significantly reducing the network training time and making it possible to train networks consisting of a large number of inputs and perceptrons.

Simply stated, back-propagation is a method for calculating the first derivative of the error function with respect to each network weight. Bishop (1995) derives and describes these calculations for the two most common forecasting error functions, the sum of squared errors and Laplacian error functions. Abe (2001) gives the description for the classification error function, the cross-entropy error function. For all of these error functions, the basic formula for the first derivative of the network weight $w_{ji}$ at the $i$th perceptron applied to the output from the $j$th perceptron

$$\frac{\partial E}{\partial w_{ji}} = \delta_j Z_i,$$

where $Z_i = g(a_i)$ is the output from the $i$th perceptron after activation, and

$$\frac{\partial E}{\partial w_{ji}}$$

is the derivative for a single output and a single training pattern. The overall estimate of the first derivative of $w_{ji}$ is obtained by summing this calculation over all N training patterns and C network outputs.

The term back-propagation gets its name from the way the term $\delta_j$ in the back-propagation formula is calculated:

$$\delta_j = g'(a_j) \cdot \sum_k w_{kj}\delta_k,$$

where the summation is over all perceptrons that use the activation from the $j$th perceptron, $g(a_j)$.

The derivative of the activation functions, $g'(a)$, varies among these functions, see the following table:

**Table 2. Activation Functions and Their Derivatives**

| **Activation Function** | $g(a)$ | $g'(a)$ |
|---|---|---|
| Linear | $g(a) = a$ | $g'(a) = 1$ (where $a$ is a constant) |
| Logistic | $g(a) = \frac{1}{1+e^{-a}}$ | $g'(a) = g(a)(1 - g(a))$ |
| Hyperbolic-tangent | $g(a) = \tanh(a)$ | $g'(a) = \operatorname{sech}^2(a) = 1 - \tanh^2(a)$ |
| Squash | $g(a) = \frac{a}{1+\lvert a\rvert}$ | $g'(a) = \frac{1}{(1+\lvert a\rvert)^2}$ |

## Creating a Feed Forward Network

The following code fragment creates the feed forward neural network shown in the following figure:

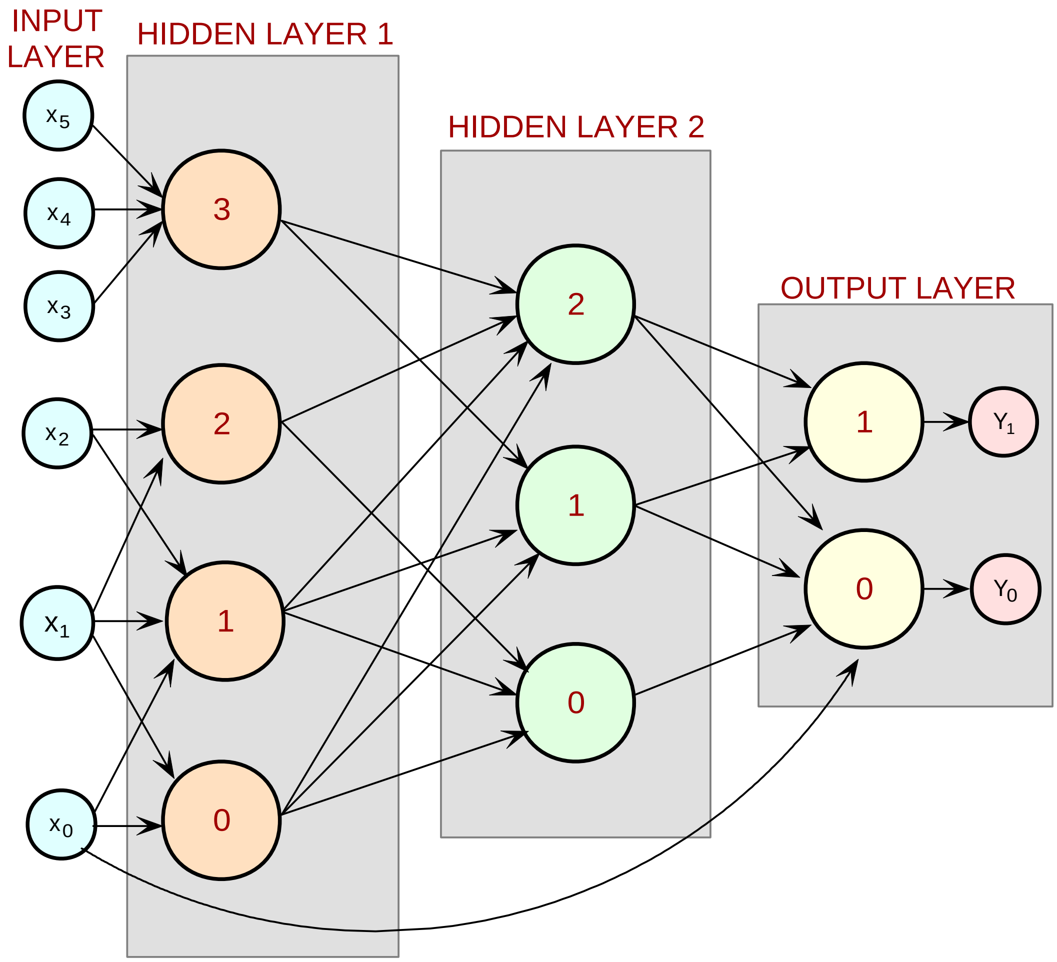



**Figure 8. A Three-Layer Feed-Forward Neural Net**

Notice that this network is more complex than the typical feed-forward network in which all nodes from each layer are connected to every node in the next layer. This network has 6 input nodes, and they are not all connected to every node in the 1st hidden layer.

Note also that the 4 perceptrons in the 1st hidden layer are not connected to every node in the 2nd hidden layer, and the perceptrons in the 2nd hidden layer are not all connected to the two outputs.

```
// ***************************************************************
// EXAMPLE CODE FOR CREATING LINKS AMONG NETWORK NODES
// ***************************************************************
   import com.imsl.datamining.neural.*;
```

```
   FeedForwardNetwork network = new FeedForwardNetwork();
   network.getInputLayer().createInputs(6);
   network.createHiddenLayer().createPerceptrons(4);
   network.createHiddenLayer().createPerceptrons(3);
   network.getOutputLayer().createPerceptrons(2);
   HiddenLayers[] hiddenLayer = network.getHiddenLayers();
   Node[] inputNode  = network.getInputLayer().getNodes();
   Node[] layer1Node = hiddenLayer[0].getNodes();
   Node[] layer2Node = hiddenLayer[1].getNodes();
   Node[] outputNode = network.getOutputLayer().getNodes();
// Create links between input nodes and 1st hidden layer
   network.link(inputNode[0], layer1Node[0]);
   network.link(inputNode[0], layer1Node[1]);
   network.link(inputNode[1], layer1Node[0]);
   network.link(inputNode[1], layer1Node[1]);
   network.link(inputNode[1], layer1Node[3]);
   network.link(inputNode[2], layer1Node[1]);
   network.link(inputNode[2], layer1Node[2]);
   network.link(inputNode[3], layer1Node[3]);
   network.link(inputNode[4], layer1Node[3]);
   network.link(inputNode[5], layer1Node[3]);
// Create links between 1st and 2nd hidden layers
   network.link(layer1Node[0], layer2Node[0]);
   network.link(layer1Node[0], layer2Node[1]);
   network.link(layer1Node[0], layer2Node[2]);
   network.link(layer1Node[1], layer2Node[0]);
   network.link(layer1Node[1], layer2Node[1]);
   network.link(layer1Node[1], layer2Node[2]);
   network.link(layer1Node[2], layer2Node[0]);
   network.link(layer1Node[2], layer2Node[2]);
   network.link(layer1Node[3], layer2Node[1]);
   network.link(layer1Node[3], layer2Node[2]);
// Create links between 2nd hidden layer and output layer
   network.link(layer2Node[0], outputNode[0]);
   network.link(layer2Node[1], outputNode[0]);
   network.link(layer2Node[1], outputNode[1]);
   network.link(layer2Node[2], outputNode[0]);
   network.link(layer2Node[2], outputNode[1]);
// Create link between input node[0] and ouput node[0]
   network.link(inputNode[0], outputNode[0]);
// ***************************************************************
```

By default, the `FeedForwardNetwork` constructor creates a feed forward network with an empty input layer, no hidden layers and an empty output layer. Input nodes are created by accessing the empty input layer and creating 6 nodes within it. Two hidden layers are then created within the network using the `FeedForwardNetwork.createHiddenLayer().createPerceptrons()` method. Four perceptrons are created within the first hidden layer and three within the second. Output perceptrons are created by accessing the empty output layer and creating the Perceptrons within it: `FeedForwardNetwork.getOutputLayer().createPerceptrons()`.

Links among the input nodes and perceptrons can be created using one of several approaches. If all inputs are connected to every perceptron in the first hidden layer, and if all perceptrons are connected to every perceptron in the following layer, which is a standard architecture for feed forward networks, then a call to the `FeedForwardNetwork.linkAll()` method can be used to create these links.

However, this example does not use that standard configuration. Some links are missing. In this case, the approach used is to construct individual links using the `FeedForwardNetwork.link()` method. This requires one call for every link.

An alternate approach is to first create all links and then to remove those that are not needed. The following code illustrates this approach:

```
// ***********************************************************
// EXAMPLE CODE FOR REMOVING LINKS AMONG NETWORK NODES
// ***********************************************************
   import com.imsl.datamining.neural.*;

   FeedForwardNetwork network = new FeedForwardNetwork();
   InputNode[] inputNode      = network.getInputLayer().createInputs(6);
   Perceptron[] hiddenLayer1  = network.createHiddenLayer().createPerceptrons(4);
   Perceptron[] hiddenLayer2  = network.createHiddenLayer().createPerceptrons(3);
   Perceptron[] outputLayer   = network.getOutputLayer().createPerceptrons(2);
   network.linkAll(); // Creates standard feed forward configuration
// Remove links between input nodes and 1st hidden layer
   network.remove(network.findLink(inputNode[0],hiddenLayer1[2]));
   network.remove(network.findLink(inputNode[0],hiddenLayer1[3]));
   network.remove(network.findLink(inputNode[1],hiddenLayer1[3]));
   network.remove(network.findLink(inputNode[2],hiddenLayer1[0]));
   network.remove(network.findLink(inputNode[2],hiddenLayer1[3]));
   network.remove(network.findLink(inputNode[3],hiddenLayer1[0]));
   network.remove(network.findLink(inputNode[3],hiddenLayer1[1]));
   network.remove(network.findLink(inputNode[3],hiddenLayer1[2]));
   network.remove(network.findLink(inputNode[4],hiddenLayer1[0]));
   network.remove(network.findLink(inputNode[4],hiddenLayer1[1]));
```

```
   network.remove(network.findLink(inputNode[4],hiddenLayer1[2]));
   network.remove(network.findLink(inputNode[5],hiddenLayer1[0]));
   network.remove(network.findLink(inputNode[5],hiddenLayer1[1]));
   network.remove(network.findLink(inputNode[5],hiddenLayer1[2]));
// Remove links between 1st and 2nd hidden layers
   network.remove(network.findLink(hiddenLayer1[2],hiddenLayer2[1]));
   network.remove(network.findLink(hiddenLayer1[3],hiddenLayer2[0]));
// Remove links between 2nd hidden layer and the output layer
   network.remove(network.findLink(hiddenLayer2[0],outputLayer[1]));
// Add link from input node[0] to output node[0]
   network.link(inputNode[0], outputNode[0]);
// ***********************************************************
```

In the above fragment, all links are created using the `FeedForwardNetwork.linkAll()` method. This creates a total of 6*4+4*3+3*2=42 links, not including the link between the first input node and the first output node. Links that skip layers are not created by the `linkAll()` method.

Links are then selectively removed starting with the first input node and proceeding to links between the last hidden layer and the output layers. In this case, there are 6*4=24 possible links between the input nodes and first hidden layer. Fourteen of them had to be removed. Between the first hidden layer and second, there are 4*3=12 possible links. Two of them were removed. Between the second hidden layer and output layer there are 3*2=6 possible links, and only one needed to be removed. Finally the skip-layer link between the first input node and first output node is added.

After creating and removing links among layers, the activation function used with each perceptron can be selected. By default, every perceptron in the hidden layers use the logistic activation function and every perceptron in the output layers uses the linear activation function. The following fragment shows how to change the activation function in the hidden layer perceptrons from logistic to hyperbolic-tangent and the output layer from linear to logistic. It also creates a connection directly from the first input node to the output node.

```
// ***********************************************************
// EXAMPLE CODE FOR SETTING NON-DEFAULT ACTIVATION FUNCTIONS
// ***********************************************************
   import com.imsl.datamining.neural.*;

   FeedForwardNetwork network = new FeedForwardNetwork();
   InputNode[] inputNode      = network.getInputLayer().createInputs(6);
   Perceptron[] hiddenLayer1  = network.createHiddenLayer().createPerceptrons(4);
   Perceptron[] hiddenLayer2  = network.createHiddenLayer().createPerceptrons(3);
   Perceptron[] outputLayer   = network.getOutputLayer().createPerceptrons(2);
```

```
   for (int k = 0;  k  hiddenLayer1.length;  k++) {
       hiddenLayer1[k].setActivation(Activation.TANH);
   }
   for (int k = 0;  k  hiddenLayer2.length;  k++) {
       hiddenLayer2[k].setActivation(Activation.TANH);
   }
   for (int k = 0;  k  outputLayer.length;  k++) {
       output[k].setActivation(Activation.LOGISTIC);
   }
.
.
.
// ***************************************************************
```

## Training

Trainers are used to find the network weights that produce network outputs matching a set of training targets. The training targets together with their associated network inputs are referred to as training patterns. Training patterns can be historical data relating network inputs to its outputs, or they can be developed from expert opinion or theoretical analysis. In the end, each training pattern relates specific network inputs to its real or desired target outputs.

In JMSL, all trainers implement the `com.imsl.datamining.neural.Trainer` interface. The number of training input attributes must equal the number of input nodes, and the number of training outputs, sometimes called training targets, must equal the number of output perceptrons created for the network.

### Single Stage Trainers

`QuasiNewtonTrainer` and `LeastSquaresTrainer` are single stage trainers. They use all available training patterns and a specific optimization method to find optimum network weights. The best set of weights is a set that minimizes the error between the network output and its training targets. The following code fragment illustrates how to use the quasi-Newton method for single stage network training.

```
// ***************************************************************
// EXAMPLE CODE FOR ONE-STAGE TRAINER
// ***************************************************************
   double xData[][] = ...
   double yData[][] = ...
   QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
```

```
   trainer.setGradientTolerance(1.0e-7);
   trainer.train(network, xData, yData);
.
.
.
// *************************************************************
```

In this example, xData and yData are two-dimensional arrays containing the input attributes and output targets respectively. The number of rows in these arrays is equal to the number of training patterns. The number of columns in xData is equal to the number of input attributes, after applying any necessary preprocessing. The number of columns in yData is equal to the number of network outputs. The `setGradientTolerance()` method is one of several optional settings for tailoring the convergence criteria used with the training optimizer.

`LeastSquaresTrainer` is another single stage trainer. There are two principal differences between this trainer and the quasi-Newton trainer. First their optimization algorithms are different. The least squares trainer uses the Levenberg-Marquardt algorithm to optimize the network. As the name implies, the quasi-Newton trainer uses a modified Newton algorithm for optimization. In some applications, depending upon the data and the network architecture, one method may train the network faster than the other.

Another key difference between these single stage trainers is that the least squares trainer only uses one error function, the sum of squared errors. The quasi-Newton trainer, by default, uses the same error function. However, it also has an interface that accepts a user-supplied error function.

### Multistage Trainers

When there are a large number of training patterns, single stage trainers will often take too long to complete network training. For these applications, a multistage trainer could be used to reduce training time. Multistage trainers provide considerably more flexibility in designing an optimum training scheme. All of these trainers break network training into two stages. Stage II is optional. That is, a multistage trainer can be requested to only conduct Stage I training, or it can be requested to conduct both Stage I and II training.

The main difference between Stage I and II training is that Stage I training is conducted multiple times using randomly selected subsets of all available training patterns. Each training session is referred to as an epoch. Although each epoch uses a different set of randomly selected training patterns, the number of patterns is the same for every epoch. Typically, because they are using different data, the solutions vary among epochs.

Stage II training is conducted following the Stage I training using the best set of weights obtained during Stage I. This ensures that the weights developed during Stage II training will always be as good as or better than those determined during Stage I training. The entire set of original training patterns is used during Stage II training, and only one training session is completed.

There is no requirement to use the same trainer for both stages, although there is nothing wrong with that approach. The least squares trainer might be used for Stage I training and the quasi-Newton trainer might be used for Stage II training. In addition, the optimization settings for each trainer can be different. In JMSL, the multistage trainer is implemented using the `EpochTrainer` class.

The following code fragment illustrates the use of the epoch multistage trainer:

```
// ***********************************************************
// EXAMPLE CODE FOR MULTISTAGE EPOCH TRAINER
// ***********************************************************
   double xData[][] = ...
   double yData[][] = ...
   QuasiNewtonTrainer stageITrainer   = new QuasiNewtonTrainer();
   LeastSquaresTrainer stageIITrainer = new LeastSquaresTrainer();
   EpochTrainer trainer = new EpochTrainer(stageITrainer, stageIITrainer);
   trainer.setNumberOfEpochs(20);
   trainer.setEpochSize(3000);
.
.
.
// ***********************************************************
```

In this example, a quasi-Newton trainer is selected for the Stage I trainer, and the least squares trainers is used for Stage II. Stage I will consists of 20 training epochs. The training of each epoch uses 3,000 randomly selected training patterns with the quasi-Newton trainer. The epoch with the smallest training error supplies the starting values for the Stage II trainer.

## Data Preprocessing

Data preprocessing, or filtering, is the term used to describe the process of scaling or transforming input attributes into numerical values suitable for network training. In general it is important to scale all input attributes to a common range, either [0, 1] or [-1, 1]. The algorithm used for obtaining values for the network weights assumes that the inputs are scaled to one of these ranges. If some network inputs have values that cover a much broader range, then the initial weights can be far from optimum causing network training to fail or take an excessively long time.

Network input data are classified into three general categories: continuous, ordinal and nominal. JMSL provides methods for preprocessing all three data types. Continuous data are scaled using the `ScaleFilter` class. In addition, lagged versions of continuous time series data can be created using the `TimeSeriesFilter` or `TimeSeriesClassFilter` class.

Categorical data, such as color or preference ratings, are either ordinal and nominal data. JMSL provides methods `UnsupervisedOrdinalFilter` and `UnsupervisedNominalFilter` to preprocess ordinal and nominal data respectively. `UnsupervisedOrdinalFilter` transforms ordinal data into values between 0 and 1, which allows them to be treated as continuous data.

Nominal data, on the other hand, can be transformed using several methods. `UnsupervisedNominalFilter` converts a single nominal variable with m classes into m columns containing the values 0 and 1. This is referred to as binary encoding of nominal classification information.

The following code fragment illustrates the use of some of these preprocessing methods:

```
// ***************************************************************
// EXAMPLE CODE FOR PREPROCESSING NOMINAL AND CONTINUOUS DATA
// ***************************************************************
   double[][] yData = {....};
   int[] nominalVariable={.....};
   int nClasses = 3;

// Create a nominal filter for binary encoding of a nominal variable
// that has 3 categorical values
   UnsupervisedNominalFilter nominalFilter = new UnsupervisedNominalFilter(nClasses);
   int[][] binaryColumns = nominalFilter(nominalVariable);

// Create a scale filter for scaling continuous data in a range of [0,1]
   ScaleFilter scaleFilter = new ScaleFilter(ScaleFilter.BOUNDED_SCALING);
// Apply the scale filter to two continuous variables, x1 and x2
   scaleFilter.setBounds(-200,1000,0,1); // Original values [-200, 1000]
   scaleFilter.encode(x1);
   scaleFilter.setBounds(0,5000,0,1);    // Original values [0, 5000]
   scaleFilter.encode(x2);

// Load the encoded columns into xData
   int n = nominalVariable.length;
   double[][] xData = new double[n][3+3];
   for(int i=0; i  n; i++){
      xData[i][0] = x1[i];
      xData[i][1] = x2[i];
      for(int j=0; j  nClasses; j++) xData[i][j+2] = binaryColumns[i][j];
   }
```

```
.
.
.
// ***********************************************************
```

In the above example, one nominal variable consisting of values representing 3 different classes, or categories, is encoded into 3 binary columns using `UnsupervisedNominalFilter` class. Two continuous variables are scaled using the `ScaleFilter` class, and these five columns are then loaded into xData in preparation for network training.

### Serialization

Neural network training can require a substantial amount of time, so it is often desirable to save a trained network for later use in forecasting. Java serialization can be used to save the results of network training.

When an object is serialized, its state is saved. However, the code implementing the class (the class file) is not saved with the serialized file. Hence when the object is deserialized, the code that created the serialized object should be in the classpath. Otherwise deserialization will fail.

For an object to be serialized, it must implement the `java.io.Serializable` interface. The following code fragment serializes key network and training information into four files. One contains the network weights, another contains the training statistics, and two additional files contain the training patterns. This is done using a `write(Object,String)` method that takes a file name and writes the serialized object to that file.

```
// ***********************************************************
// EXAMPLE CODE FOR SAVING TRAINED NETWORK USING SERIALIZATION
// ***********************************************************
.
.
.
// ***********************************************************
// SAVE A TRAINED NETWORK BY SAVING THE SERIALIZED NETWORK OBJECTS
// ***********************************************************
// Saving network weights and structural information
   write(network, "MyNetwork.ser");
// Saving training information available from computeStatistics()
   write(trainer, "MyNetworkTrainer.ser");
// Saving xData training targests
   write(xData, "MyNetworkxData.ser");
// Saving yData training targets
```

```
   write(yData, "MyNetworkyData.ser);
// **************************************************************
// WRITE SERIALIZED OBJECT TO A FILE
// **************************************************************
static public void write(Object obj, String filename)
   throws IOException {
   FileOutputStream fos = new FileOutputStream(filename);
   ObjectOutputStream oos = new ObjectOutputStream(fos);
   oos.writeObject(obj);
   oos.close();
   fos.close();
}
// **************************************************************
```

Notice that not only is the network object serialized and saved, the trainer and training patterns, xData and yData, are also saved. This was only done to allow someone to calculate the additional network statistics. If these are not needed, then these training patterns need not be saved. However, for forecasting, it is essential to remember the specifc order and nature of the network inputs used during training. This information is not saved in the network serialized file.

When an object is deserialized, the object is reconstructed using the saved serialization file. The following code deserializes the previously saved network information.

```
// **************************************************************
// EXAMPLE CODE FOR READING TRAINED NETWORK FROM SERIALIZED FILES
// **************************************************************
.
.
.
// **************************************************************
// READ THE TRAINED NETWORK FROM THE SERIALIZED NETWORK OBJECT
   Network network = (Network)read("MyNetwork.ser");
// READ THE SERIALIZED XDATA[][] AND YDATA[][] ARRAYS OF TRAINING
// PATTERNS.
   xData = (double[][])read("MyNetworkxData.ser");
   yData = (double[][])read("MyNetworkyData.ser");
// READ THE SERIALIZED TRAINER OBJECT
   Trainer trainer = (Trainer)read("MyNetworkTrainer.ser");
// **************************************************************
// DISPLAY TRAINING STATISTICS
// **************************************************************
   double stats[] = network.computeStatistics(xData, yData);
```

```
.
.
.

// ****************************************************************
// READ SERIALIZED NETWORK FROM A FILE
// ****************************************************************
static public Object read(String filename)
   throws IOException, ClassNotFoundException {
   FileInputStream fis = new FileInputStream(filename);
   ObjectInputStream ois = new ObjectInputStream(fis);
   Object obj = ois.readObject();
   ois.close();
   fis.close();
   return obj;
}
// ****************************************************************
```

### Logging

The training classes support logging using the standard Java classes. The following code fragment enables logging for an epoch trainer. The log is stored into a file with the name `MyNetworkTraining.log`

```
// ****************************************************************
// EXAMPLE CODE FOR CREATING TRAINING LOG
// ****************************************************************
import java.util.logging.*;
.
.
.
try {
   Handler handler = new FileHandler("MyNetworkTraining.log");
   Logger logger = Logger.getLogger("com.imsl.datamining.neural");
   logger.setLevel(Level.FINEST);
   logger.addHandler(handler);
   handler.setFormatter(EpochTrainer.getFormatter());
}catch (Exception e) {
   e.printStackTrace();
}
.
.
```

```
.
// ***************************************************************
```

The standard Java logging classes are in the package `java.util.logging`. A `FileHandler` is used to write the logging information to the log file. Each of the training classes has a static method that returns a special `Formatter` designed to work with the logging statements in the trainers. All of the trainers use the same `Formatter`.

The name of the logger in each of the trainers is the fully qualified name of the trainer. Because the Java logger is hierarchical, the name `com.imsl.datamining.neural` can be used to log all of the JMSL training classes. More specific names can be used to set trainer specific logging levels. For example, setting the logging level in `com.imsl.datamining.neural.EpochTrainer` to `Level.FINEST`, while setting the level in `com.imsl.datamining.neural.QuasiNewtonTrainer` to `Level.FINE`. The trainers support logging the `Level.FINE`, `Level.FINER` and `Level.FINEST`.

## Example: Neural Network Application

This application illustrates one common approach to time series prediction using a neural network. In this case, the output target for this network is a single time series. In general, the inputs to this network consist of lagged values of the time series together with other concomitant variables, both continuous and categorical. In this application, however, only the first three lags of the time series are used as network inputs.

The objective is to train a neural network for forecasting the series $Y_t$, $t = 0, 1, 2, \ldots$, from the first three lags of $Y_t$, i.e.

$$Y_t = f(Y_{t-1}, Y_{t-2}, Y_{t-3})$$

Since this series consists of data from several company departments, lagging of the series must be done within departments. This creates many missing values. The original data contains 118,519 training patterns. After lagging, 16,507 are identified as missing and are removed, leaving a total of 102,012 usable training patterns. Missing values are denoted using a number not in the training patterns, the value -9,999,999,999.0 .

The structure of the network consists of three input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure depicts this structure:

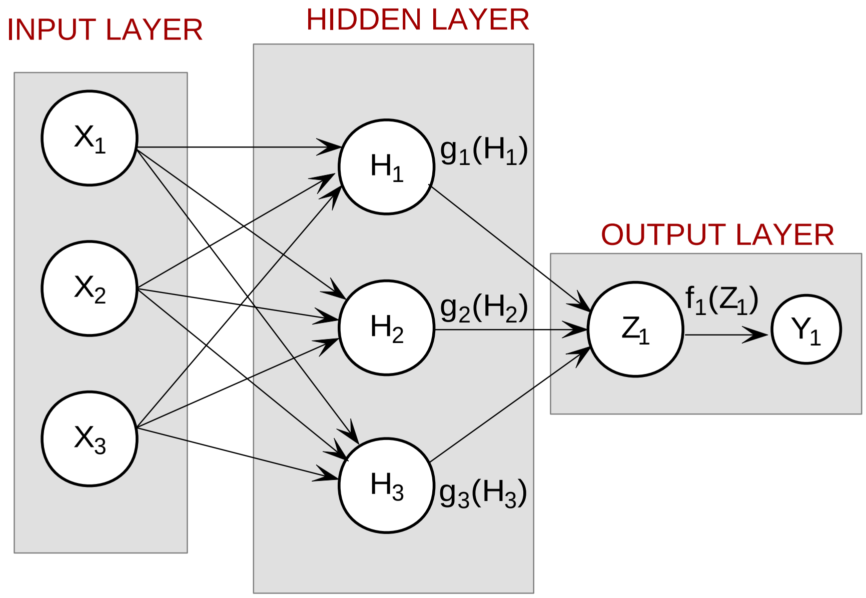



**Figure 9. An example 2-layer Feed Forward Neural Network**

There are a total of 16 weights in this network, including the 4 bias weights. All perceptrons in the hidden layer use logistic activation, and the output perceptron uses linear activation. Because of the large number of training patterns, the `Activation.LOGISTIC_TABLE` activation funtion is used instead of `Activation.LOGISTIC`. `Activation.LOGISTIC_TABLE` uses a table lookup for calculating the logistic activation function, which significantly reduces training time. However, these are not completely interchangable. If a network is trained using `Activation.LOGISTIC_TABLE`, then it is important to use the same activation function for forecasting.

All input nodes are linked to every perceptron in the hidden layer, which are in turn linked to the output perceptron. Then all inputs and the output target are scaled using the `ScaleFilter` class to ensure that all input values and outputs are in the range [0, 1]. This requires forecasts to be unscaled using the `decode()` method of the `ScaleFilter` class.

Training is conducted using the epoch trainer. This trainer allows users to customize training into two stages. Typically this is necessary when training using a large number of training patterns. Stage I training uses randomly selected subsets of training patterns to search for network solutions. Stage II training is optional, and uses the entire set of training patterns. For larger sets of training patterns, training could take many hours, or even days. In that case, Stage II training might be bypassed.

In this example, Stage I training is conducted using the Quasi-Newton trainer applied to 20 epochs, each consisting of 5,000 randomly selected observations. Stage II training also uses the Quasi-Newton trainer. The training results for each Stage I epoch and for the final Stage II solution are stored in a training log file NeuralNetworkEx1.log.

The training patterns are contained in two data files: `continuous.txt` and `output.txt`. The formats of these files are identical. The first line of the file contains the number of columns or variables in that file. The second contains a line of tab-delimited integer values. These are the column indices associated with the incoming data. The remaining lines contain tab-delimited, floating point values, one for each of the incoming variables.

For example, the first four lines of the `continuous.txt` file consists of the following lines:

3
1 2 3
0 0 0
0 0 0

There are 3 continuous input variables which are numbered, or labeled, as 1, 2, and 3.

## Source Code

```
import com.imsl.datamining.neural.*;
import com.imsl.math.*;
import  java.io.*;
import  java.util.*;
import  java.util.logging.*;

//*****************************************************************************
// NeuralNetworkEx1.java                                                      *
// Two Layer Feed-Forward Network Complete Example for Simple Time Series     *
//*****************************************************************************
// Synopsis:   This example illustrates how to use a Feed-Forward Neural      *
//             Network to forecast time series data.  The network target is a *
//             time series and the three inputs are the 1st, 2nd, and 3rd lag *
//             for the target series.                                         *
// Activation: Logistic_Table in Hidden Layer, Linear in Output Layer        *
// Trainer:    Epoch Trainer: Stage I - Quasi-Newton, Stage II - Quasi-Newton *
// Inputs:     Lags 1-3 of the time series                                    *
// Output:     A Time Series sorted chronologically in descending order,      *
//             i.e., the most recent observations occur before the earliest,  *
//             within each department                                         *
//*****************************************************************************

public class NeuralNetworkEx1 implements Serializable {
```

```
    private FeedForwardNetwork network;
    private static String QuasiNewton = "quasi-newton";
    private static String LeastSquares= "least-squares";
// ****************************************************************************
// Network Architecture                                                       *
// ****************************************************************************
    private static int nObs         =118519; // number of training patterns
    private static int nInputs      =  3;    // four inputs
    private static int nCategorical =  0;    // three categorical attributes
    private static int nContinuous  =  3;    // one continuous input attribute
    private static int nOutputs     =  1;    // one continuous output
    private static int nLayers      =  2;    // number of perceptron layers
    private static int nPerceptrons =  3;    // perceptrons in hidden layer
    private static int perceptrons[]={3};    // number of perceptrons in each
                                             // hidden layer
    // PERCEPTRON ACTIVATION
    private static Activation hiddenLayerActivation = Activation.LOGISTIC_TABLE;
    private static Activation outputLayerActivation = Activation.LINEAR;
// ****************************************************************************
// Epoch Training Optimization Settings                                       *
// ****************************************************************************
    private static boolean trace                  = true; //trainer logging    *
    private static int nEpochs                    = 20;   //number of epochs   *
    private static int epochSize                  = 5000; //samples per epoch  *
    // Stage I Trainer - Quasi-Newton Trainer *********************************
    private static int    stage1Iterations        = 5000; //max. iterations    *
    private static double stage1MaxStepsize       = 50.0; //max. stepsize      *
    private static double stage1StepTolerance     = 1e-09;//step tolerance     *
    private static double stage1RelativeTolerance = 1e-11;//rel. tolerance     *
    // Stage II Trainer - Quasi-Newton Trainer ********************************
    private static int    stage2Iterations        = 5000; //max. iterations    *
    private static double stage2MaxStepsize       = 50.0; //max. stepsize      *
    private static double stage2StepTolerance     = 1e-09;//step tolerance     *
    private static double stage2RelativeTolerance = 1e-11;//rel. tolerance     *
// ****************************************************************************
// FILE NAMES AND FILE READER DEFINITIONS                                     *
// ****************************************************************************
    // READERS
    private static BufferedReader attFileInputStream;
    private static BufferedReader contFileInputStream;
    private static BufferedReader outputFileInputStream;
```

```
    // OUTPUT FILES
    // File Name for training log produced when trace = true
    private static String trainingLogFile = "NeuralNetworkEx1.log";
    // File Name for Serialized Network
    private static String networkFileName = "NeuralNetworkEx1.ser";
    // File Name for Serialized Trainer
    private static String trainerFileName = "NeuralNetworkTrainerEx1.ser";
    // File Name for Serialized xData File (training input attributes)
    private static String xDataFileName   = "NeuralNetworkxDataEx1.ser";
    // File Name for Serialized yData File (training output targets)
    private static String yDataFileName   = "NeuralNetworkyDataEx1.ser";
    // INPUT FILES
    // Continuous input attributes file.  File contains Lags 1-3 of series
    private static String contFileName   = "continuous.txt";
    // Continuous network targets file.  File contains the original series
    private static String outputFileName = "output.txt";
// ****************************************************************************
// Data Preprocessing Settings                                                *
// ****************************************************************************
    private static double lowerDataLimit=-105000;       // lower scale limit
    private static double upperDataLimit=25000000;      // upper scale limit
    private static double missingValue = -9999999999.0;  // missing values
                                                        // indicator
// ****************************************************************************
// Time Parameters for Tracking Training Time                                 *
// ****************************************************************************
    private static Calendar startTime;
    private static Calendar endTime;
// ****************************************************************************
// Error Message Encoding for Stage II Trainer - Quasi-Newton Trainer         *
// ****************************************************************************
// Note:  For the Epoch Trainer, the error status returned is the status for   *
// the Stage II trainer, unless Stage II training is not used.                *
// ****************************************************************************
    private static String errorMsg  = "";
    // Error Status Messages for the Quasi-Newton Trainer
    private static String errorMsg0 =
      "--> Network Training";
    private static String errorMsg1 =
      "--> The last global step failed to locate a lower point than the\n"+
      "current error value.  The current solution may be an approximate\n"+
      "solution and no more accuracy is possible, or the step tolerance\n"+
```

```
      "may be too large.";
    private static String errorMsg2 =
      "--> Relative function convergence;  both both the actual and \n"+
      "predicted relative reductions in the error function are less than\n"+
      "or equal to the relative fu nction convergence tolerance.";
    private static String errorMsg3 =
      "--> Scaled step tolerance satisfied;  the current solution may be\n"+
      "an approximate local solution, or the algorithm is making very slow\n"+
      "progress and is not near a solution, or the step tolerance is too big.";
    private static String errorMsg4 =
      "--> Quasi-Newton Trainer threw a \n"+
      "MinUnconMultiVar.FalseConvergenceException.";
    private static String errorMsg5 =
      "--> Quasi-Newton Trainer threw a \n"+
      "MinUnconMultiVar.MaxIterationsException.";
    private static String errorMsg6 =
      "--> Quasi-Newton Trainer threw a \n"+
      "MinUnconMultiVar.UnboundedBelowException.";
// ***************************************************************************
// MAIN                                                                      *
// ***************************************************************************
    public static void main(String[] args) throws Exception {

        double weight[];   // Network weights
        double gradient[]; // Network gradient after training
        double x[];        // Temporary x space for generating forecasts
        double y[];        // Temporary y space for generating forecasts
        double xData[][];  // Training Patterns Input Attributes
        double yData[][];  // Training Targets Output Attributes
        double contAtt[][];// A 2D matrix for the continuous training attributes
        double outs[][];   // A matrix containing the training output tragets
        int i, j, k, m=0;  // Array indicies
        int nWeights = 0;  // Number of network weights
        int nCol     = 0;  // Number of data columns in input file
        int ignore[];      // Array of 0's and 1's (0=missing value)
        int cont_col[], outs_col[], isMissing[]={0};
        String inputLine="", temp;
        String dataElement[];
    // ***********************************************************************
    // Initialize timers                                                     *
    // ***********************************************************************
        startTime = Calendar.getInstance();
```

```
   System.out.println("--> Starting Data Preprocessing at: "+
                          startTime.getTime());

// ***************************************************************************
// Read continuous attribute data                                            *
// ***************************************************************************
   // Initialize ignore[] for identifying missing observations
   ignore    = new int[nObs];
   isMissing = new int[1];
   openInputFiles();

   nCol = readFirstLine(contFileInputStream);

   nContinuous = nCol;
   System.out.println("--> Number of continuous variables:     "+nContinuous);
   // If the number of continuous variables is greater than zero then read
   // the remainder of this file (contFile)
   if(nContinuous > 0){
      // contFile contains continuous attribute data
      contAtt      = new double[nObs][nContinuous];
      cont_col     = readColumnLabels(contFileInputStream, nContinuous);
      for (i=0; i < nObs; i++){
         isMissing[0]  = -1;
         contAtt[i] = readDataLine(contFileInputStream,
                                       nContinuous, isMissing);
         ignore[i]  = isMissing[0];
         if (isMissing[0] >= 0) m++;
      }
   }else{
      nContinuous  = 0;
      contAtt      = new double[1][1];
      contAtt[0][0]= 0;
   }
   closeFile(contFileInputStream);
// ***************************************************************************
// Read continuous output targets                                            *
// ***************************************************************************
   nCol = readFirstLine(outputFileInputStream);
   nOutputs   = nCol;
   System.out.println("--> Number of output variables:         "+nOutputs);
   outs       = new double[nObs][nOutputs];
   // Read numeric labels for continuous input attributes
```

```
   outs_col     = readColumnLabels(outputFileInputStream, nOutputs);

   m = 0;
   for (i=0; i < nObs; i++){
      isMissing[0] = ignore[i];
      outs[i]   = readDataLine(outputFileInputStream, nOutputs, isMissing);
      ignore[i] = isMissing[0];
      if (isMissing[0] >= 0) m++;
   }
   System.out.println("--> Number of Missing Observations:     " + m);
   closeFile(outputFileInputStream);
   // Remove missing observations using the ignore[] array
   m = removeMissingData(nObs, nContinuous, ignore, contAtt);
   m = removeMissingData(nObs, nOutputs, ignore, outs);

   System.out.println("--> Total Number of Training Patterns:  "+ nObs);
   nObs = nObs - m;
   System.out.println("--> Number of Usable Training Patterns: "+ nObs);

// ***************************************************************************
// Setup Method and Bounds for Scale Filter                                  *
// ***************************************************************************
   ScaleFilter scaleFilter = new ScaleFilter(ScaleFilter.BOUNDED_SCALING);
   scaleFilter.setBounds(lowerDataLimit,upperDataLimit,0,1);
// ***************************************************************************
// PREPROCESS TRAINING PATTERNS                                              *
// ***************************************************************************
   System.out.println("--> Starting Preprocessing of Training Patterns");
   xData = new double[nObs][nContinuous];
   yData = new double[nObs][nOutputs];
   for(i=0; i < nObs; i++) {
      for(j=0; j < nContinuous; j++){
         xData[i][j] = contAtt[i][j];
      }
      yData[i][0] = outs[i][0];
   }
   scaleFilter.encode(0, xData);
   scaleFilter.encode(1, xData);
   scaleFilter.encode(2, xData);
   scaleFilter.encode(0, yData) ;
// ***************************************************************************
// CREATE FEEDFORWARD NETWORK                                                *
```

```
// ***************************************************************************
   System.out.println("--> Creating Feed Forward Network Object");
   FeedForwardNetwork network = new FeedForwardNetwork();
   // setup input layer with number of inputs = nInputs = 3
   network.getInputLayer().createInputs(nInputs);
   // create a hidden layer with nPerceptrons=3 perceptrons
   network.createHiddenLayer().createPerceptrons(nPerceptrons);
   // create output layer with nOutputs=1 output perceptron
   network.getOutputLayer().createPerceptrons(nOutputs);
   // link all inputs and perceptrons to all perceptrons in the next layer
   network.linkAll();
   // Get Network Perceptrons for Setting Their Activation Functions
   Perceptron perceptrons[] = network.getPerceptrons();
   // Set all hidden layer perceptrons to logistic_table activation
   for (i=0; i < perceptrons.length-1; i++) {
      perceptrons[i].setActivation(hiddenLayerActivation);
   }
   perceptrons[perceptrons.length-1].setActivation(outputLayerActivation);
   System.out.println("--> Feed Forward Network Created with 2 Layers");
// *****************************************************************************
// TRAIN NETWORK USING EPOCH TRAINER                                           *
// *****************************************************************************
   System.out.println("--> Training Network using Epoch Trainer");
   Trainer trainer = createTrainer(QuasiNewton,QuasiNewton);
   Calendar startTime = Calendar.getInstance();
   // Train Network
   trainer.train(network, xData, yData);

   // Check Training Error Status
   switch(trainer.getErrorStatus()){
      case 0: errorMsg = errorMsg0;
              break;
      case 1: errorMsg = errorMsg1;
              break;
      case 2: errorMsg = errorMsg2;
              break;
      case 3: errorMsg = errorMsg3;
              break;
      case 4: errorMsg = errorMsg4;
              break;
      case 5: errorMsg = errorMsg5;
              break;
```

```
      case 6: errorMsg = errorMsg6;
              break;
      default:errorMsg = "--> Unknown Error Status Returned from Trainer";
   }
   System.out.println(errorMsg);
   Calendar currentTimeNow = Calendar.getInstance();
   System.out.println("--> Network Training Completed at: "+currentTimeNow.getTime());
   double duration = (double)(currentTimeNow.getTimeInMillis() -
                           startTime.getTimeInMillis())/1000.0;
   System.out.println("--> Training Time: "+duration+" seconds");

// ***********************************************************************
// DISPLAY TRAINING STATISTICS                                           *
// ***********************************************************************
   double stats[] = network.computeStatistics(xData, yData);
   // Display Network Errors
   System.out.println("***********************************************");
   System.out.println("--> SSE:                        "+(float)stats[0]);
   System.out.println("--> RMS:                        "+(float)stats[1]);
   System.out.println("--> Laplacian Error:            "+(float)stats[2]);
   System.out.println("--> Scaled Laplacian Error:     "+(float)stats[3]);
   System.out.println("--> Largest Absolute Residual: "+(float)stats[4]);
   System.out.println("***********************************************");
   System.out.println("");
// ***********************************************************************
// OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS                      *
// ***********************************************************************
   System.out.println("--> Getting Network Weights and Gradients");
   // Get weights
   weight   = network.getWeights();
   // Get number of weights = number of gradients
   nWeights = network.getNumberOfWeights();
   // Obtain Gradient Vector
   gradient = trainer.getErrorGradient();
   // Print Network Weights and Gradients
   System.out.println(" ");
   System.out.println("--> Network Weights and Gradients:");
   System.out.println("***********************************************");
   double[][] printMatrix = new double[nWeights][2];
   for(i=0; i < nWeights; i++){
      printMatrix[i][0] = weight[i];
      printMatrix[i][1] = gradient[i];
```

```
        }
        // Print result without row/column labels.
        String[] colLabels = {"Weight", "Gradient"};
        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat mf;
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setColumnLabels(colLabels);
        pm.setTitle("Weights and Gradients");
        pm.print(mf, printMatrix);

        System.out.println("**********************************************");
     // ******************************************************************
     // SAVE THE TRAINED NETWORK BY SAVING THE SERIALIZED NETWORK OBJECT       *
     // ******************************************************************
        System.out.println("\n--> Saving Trained Network into "+
                                networkFileName);
        write(network, networkFileName);
        System.out.println("--> Saving Network Trainer into "+
                                trainerFileName);
        write(trainer, trainerFileName);
        System.out.println("--> Saving xData into "+
                                xDataFileName);
        write(xData, xDataFileName);
        System.out.println("--> Saving yData into "+
                                yDataFileName);
        write(yData, yDataFileName);
     }
  // ********************************************************************
  // OPEN DATA FILES                                                           *
  // ********************************************************************
     static public void openInputFiles(){
        try{
           // Continuous  Input Attributes
           InputStream contInputStream = new FileInputStream(contFileName);
           contFileInputStream         =
                    new BufferedReader(new InputStreamReader(contInputStream));
           // Continuous Output Targets
           InputStream outputInputStream = new FileInputStream(outputFileName);
           outputFileInputStream         =
                    new BufferedReader(new InputStreamReader(outputInputStream));
        }catch(Exception e){
```

```
            System.out.println("-->ERROR: "+e);
            System.exit(0);
         }
     }
// ***************************************************************************
// READ FIRST LINE OF DATA FILE AND RETURN NUMBER OF COLUMNS IN FILE         *
// ***************************************************************************
    static public int readFirstLine(BufferedReader inputFile){
       String inputLine="", temp;
       int nCol=0;
       try{
          temp      = inputFile.readLine();
          inputLine = temp.trim();
          nCol      = Integer.parseInt(inputLine);
       }catch(Exception e){
          System.out.println("--> ERROR READING 1st LINE OF File" + e);
          System.exit(0);
       }
       return nCol;
   }
// ***************************************************************************
// READ COLUMN LABELS (2ND LINE IN FILE)                                     *
// ***************************************************************************
    static public int[] readColumnLabels(BufferedReader inputFile, int nCol){
          int contCol[] = new int[nCol];
          String inputLine="", temp;
          String dataElement[];
          // Read numeric labels for continuous input attributes
          try{
             temp = inputFile.readLine();
             inputLine = temp.trim();
          }catch(Exception e){
            System.out.println("--> ERROR READING 2nd LINE OF FILE: "+ e);
            System.exit(0);
          }
          dataElement = inputLine.split(" ");
          for (int i=0; i < nCol; i++){
             contCol[i] = Integer.parseInt(dataElement[i]);
          }
          return contCol;
    }
// ***************************************************************************
```

```
// READ DATA ROW                                                             *
// ***************************************************************************
    static public double[] readDataLine(BufferedReader inputFile,
                                    int nCol, int[] isMissing){
          double missingValueIndicator = -9999999999.0;
          double dataLine[] = new double[nCol];
          double contCol[]  = new double[nCol];
          String inputLine="", temp;
          String dataElement[];
          try{
             temp = inputFile.readLine();
             inputLine = temp.trim();
          }catch(Exception e){
             System.out.println("-->ERROR READING LINE: " + e);
             System.exit(0);
          }
          dataElement = inputLine.split(" ");
          for (int j=0; j < nCol; j++){
              dataLine[j] = Double.parseDouble(dataElement[j]);
              if (dataLine[j] == missingValueIndicator)isMissing[0] = 1;
          }
          return dataLine;
    }
// ***************************************************************************
// CLOSE FILE                                                                *
// ***************************************************************************
    static public void closeFile(BufferedReader inputFile){
       try{
          inputFile.close();
       }catch(Exception e){
          System.out.println("ERROR: Unable to close file: " + e);
          System.exit(0);
       }
    }
// ***************************************************************************
// REMOVE MISSING DATA                                                       *
// ***************************************************************************
    // Now remove all missing data using the ignore[] array
    // and recalculate the number of usable observations, nObs
    // This method is inefficient, but it works.  It removes one case at a
    // time, starting from the bottom.  As a case (row) is removed, the cases
    // below are pushed up to take it's place.
```

```
    // ***********************************************************************
    static public int removeMissingData(int nObs,int nCol,int ignore[],
                                           double[][] inputArray){
        int m=0;
        for(int i=nObs-1; i >=0; i--){
            if(ignore[i]>=0){
            // the ith row contains a missing value
            // remove the ith row by shifting all rows below the
            // ith row up by one position, e.g. row i+1  -> row i
               m++;
               if (nCol > 0){
                  for(int j=i; j < nObs-m; j++){
                     for (int k=0; k < nCol; k++){
                         inputArray[j][k]=inputArray[j+1][k];
                     }
                  }
               }
            }
        }
        return m;
    }
// ***************************************************************************
// Create Stage I/Stage II Trainer                                           *
// ***************************************************************************
    static public Trainer createTrainer(String s1, String s2) {
        EpochTrainer epoch = null;         // Epoch Trainer (returned by this method)
        QuasiNewtonTrainer stage1Trainer;  // Stage I  Quasi-Newton Trainer
        QuasiNewtonTrainer stage2Trainer;  // Stage II Quasi-Newton Trainer
        LeastSquaresTrainer stage1LS;      // Stage I  Least Squares Trainer
        LeastSquaresTrainer stage2LS;      // Stage II Least Squares Trainer
        Calendar currentTimeNow ;          // Calendar time tracker

        // Create Epoch (Stage I/Stage II) trainer from above trainers.
        System.out.println("    --> Creating Epoch Trainer");
        if (s1.equals(QuasiNewton)){
           // Setup stage I quasi-newton trainer
           stage1Trainer = new QuasiNewtonTrainer();
           //stage1Trainer.setMaximumStepsize(maxStepSize);
           stage1Trainer.setMaximumTrainingIterations(stage1Iterations);
           stage1Trainer.setStepTolerance(stage1StepTolerance);
           if (s2.equals(QuasiNewton)){
             stage2Trainer = new QuasiNewtonTrainer();
```

```
      //stage2Trainer.setMaximumStepsize(maxStepSize);
      stage2Trainer.setMaximumTrainingIterations(stage2Iterations);
      epoch = new EpochTrainer(stage1Trainer, stage2Trainer);
    }else{
      if (s2.equals(LeastSquares)){
        stage2LS = new LeastSquaresTrainer();
        stage2LS.setInitialTrustRegion(1.0e-3);
        //stage2LS.setMaximumStepsize(maxStepSize);
        stage2LS.setMaximumTrainingIterations(stage2Iterations);
        epoch = new EpochTrainer(stage1Trainer, stage2LS);
      }else{
        epoch = new EpochTrainer(stage1Trainer);
      }
    }
  }else{
    // Setup stage I least squares trainer
    stage1LS = new LeastSquaresTrainer();
    stage1LS.setInitialTrustRegion(1.0e-3);
    stage1LS.setMaximumTrainingIterations(stage1Iterations);
    //stage1LS.setMaximumStepsize(maxStepSize);
    if (s2.equals(QuasiNewton)){
      stage2Trainer = new QuasiNewtonTrainer();
      //stage2Trainer.setMaximumStepsize(maxStepSize);
      stage2Trainer.setMaximumTrainingIterations(stage2Iterations);
      epoch = new EpochTrainer(stage1LS, stage2Trainer);
    }else{
      if (s2.equals(LeastSquares)){
        stage2LS = new LeastSquaresTrainer();
        stage2LS.setInitialTrustRegion(1.0e-3);
        //stage2LS.setMaximumStepsize(maxStepSize);
        stage2LS.setMaximumTrainingIterations(stage2Iterations);
        epoch = new EpochTrainer(stage1LS, stage2LS);
      }else{
        epoch = new EpochTrainer(stage1LS);
      }
    }
  }
  epoch.setNumberOfEpochs(nEpochs);
  epoch.setEpochSize(epochSize);
  epoch.setRandom(new com.imsl.stat.Random(1234567));
  epoch.setRandomSamples(new com.imsl.stat.Random(12345),
                         new com.imsl.stat.Random(67891));
```

```
        System.out.println("    --> Trainer:  Stage I - "+s1+" Stage II "+s2);
        System.out.println("    --> Number of Epochs:   " + nEpochs);
        System.out.println("    --> Epoch Size:         " + epochSize);
        // Describe optimization setup for Stage I training
        System.out.println("    --> Creating Stage I  Trainer");
        System.out.println("    --> Stage I Iterations:          " + stage1Iterations);
        System.out.println("    --> Stage I Step Tolerance:      " + stage1StepTolerance);
        System.out.println("    --> Stage I Relative Tolerance:  " + stage1RelativeTolerance);
        System.out.println("    --> Stage I Step Size:           " + "DEFAULT");
        System.out.println("    --> Stage I Trace:               " + trace);
        if(s2.equals(QuasiNewton) || s2.equals(LeastSquares)){
        // Describe optimization setup for Stage II training
           System.out.println("    --> Creating Stage II Trainer");
           System.out.println("    --> Stage II Iterations:         " + stage2Iterations);
           System.out.println("    --> Stage II Step Tolerance:     " + stage2StepTolerance);
           System.out.println("    --> Stage II Relative Tolerance: " + stage2RelativeTolerance);
           System.out.println("    --> Stage II Step Size:          " + "DEFAULT");
           System.out.println("    --> Stage II Trace:              " + trace);
        }
        if (trace) {
            try {
                Handler handler = new FileHandler(trainingLogFile);
                Logger logger = Logger.getLogger("com.imsl.datamining.neural");
                logger.setLevel(Level.FINEST);
                logger.addHandler(handler);
                handler.setFormatter(EpochTrainer.getFormatter());
                System.out.println("    --> Training Log Stored in "+trainingLogFile);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        currentTimeNow = Calendar.getInstance();
        System.out.println("--> Starting Network Training at "+currentTimeNow.getTime());
        // Return Stage I/Stage II trainer
        return epoch;
    }

// ****************************************************************************
// WRITE SERIALIZED OBJECT TO A FILE                                          *
// ****************************************************************************
    static public void write(Object obj, String filename)
       throws IOException {
```

```
        FileOutputStream fos = new FileOutputStream(filename);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(obj);
        oos.close();
        fos.close();
    }
}
// ****************************************************************************
```

## Output

```
--> Starting Data Preprocessing at: Thu Oct 14 17:27:04 CDT 2004
--> Number of continuous variables:     3
--> Number of output variables:         1
--> Number of Missing Observations:     16507
--> Total Number of Training Patterns:  118519
--> Number of Usable Training Patterns: 102012
--> Starting Preprocessing of Training Patterns
--> Creating Feed Forward Network Object
--> Feed Forward Network Created with 2 Layers
--> Training Network using Epoch Trainer
    --> Creating Epoch Trainer
    --> Trainer:  Stage I - quasi-newton Stage II quasi-newton
    --> Number of Epochs:   20
    --> Epoch Size:         5000
    --> Creating Stage I  Trainer
    --> Stage I Iterations:          5000
    --> Stage I Step Tolerance:      1.0E-9
    --> Stage I Relative Tolerance:  1.0E-11
    --> Stage I Step Size:           DEFAULT
    --> Stage I Trace:               true
    --> Creating Stage II Trainer
    --> Stage II Iterations:         5000
    --> Stage II Step Tolerance:     1.0E-9
    --> Stage II Relative Tolerance: 1.0E-11
    --> Stage II Step Size:          DEFAULT
    --> Stage II Trace:              true
    --> Training Log Stored in NeuralNetworkEx1.log
--> Starting Network Training at Thu Oct 14 17:32:33 CDT 2004
--> The last global step failed to locate a lower point than the
current error value.  The current solution may be an approximate
```

```
solution and no more accuracy is possible, or the step tolerance
may be too larger.
--> Network Training Completed at: Thu Oct 14 18:18:08 CDT 2004
--> Training Time: 2735.341 seconds
**********************************************
--> SSE:                       3.88076
--> RMS:                       0.12284768
--> Laplacian Error:           125.36781
--> Scaled Laplacian Error:    0.20686063
--> Largest Absolute Residual: 0.500993
**********************************************

--> Getting Network Weights and Gradients

--> Network Weights and Gradients:
**********************************************
Weights and Gradients
   Weight   Gradient
   1.921   -0
   1.569    0
-199.709    0
   0.065   -0
  -0.003    0
 106.62     0
   1.221   -0
   0.787    0
 119.169    0
-129.8      0
 146.822    0
  -0.076    0
  -6.022   -0
  -5.257    0.001
   2.19     0
  -0.377    0

**********************************************

--> Saving Trained Network into NeuralNetworkEx1.ser
--> Saving Network Trainer into NeuralNetworkTrainerEx1.ser
--> Saving xData into NeuralNetworkxDataEx1.ser
--> Saving yData into NeuralNetworkyDataEx1.ser
```

### Results

The above output indicates that the network successfully completed its training. The final sum of squared errors was 3.88, and the RMS (the scaled version of the sum of squared errors) was 0.12. All of the gradients at this solution are nearly zero, which is expected if network training found a local or global optima. Non-zero gradients usually indicate there was a problem with network training.

Examining the training log for this application, NeuralNetworkEx1.log, illustrates the importance of Stage II training.

### Portions of the Training Log - NeuralNetworkEx1.log

```
.
.
.
End EpochTrainer Stage 1
        Best Epoch       15
        Error Status     17
        Best Error       0.03979299031789641
        Best Residual    0.03979299031789641
        SSE              1072.1281419136983
        RMS              33.93882798404427
        Laplacian        429.30253410528974
        Scaled Laplacian 0.7083620086220087
        Max Residual     11.837166167929052
Exiting com.imsl.datamining.neural.EpochTrainer.train Stage 1
Beginning com.imsl.datamining.neural.EpochTrainer.train Stage 2
.
.
.
Exiting com.imsl.datamining.neural.EpochTrainer.train Stage 2
    Summary
    Error Status     1
    Best Error       3.88076005209094
    SSE              3.88076005209094
    RMS              0.12284767343218107
    Laplacian        125.3678136373788
    Scaled Laplacian 0.20686063843020083
    Max Residual     0.5009930332151435
```

The training log indicates that the best Stage I epoch occurred at iteration 15, and that 17 of the 20 Stage I epochs detected a problem with training optimization. Other parts of the log indicate that these problems included: possible local minima, and maximum number of iterations exceeded. Although these problems are warning messages and not true errors, they do indicate that convergence to a global optima is uncertain for 17 of the 20 epochs. Possibly increasing the epoch size might have provided more stable Stage I training.

More disturbing is the fact that for the best epoch=15, the sum of squared errors totaled over all training patterns is 1072.13. Epoch 15 was used as the starting point for the Stage II training which was able to reduce this sum of squared errors to 3.88. This suggests that although the epoch size, epochSize=5000, was too small for effective Stage I training, the Stage II trainer was able to locate a better solution.

However, even the Stage II trainer returned a non-zero error status, errorStatus=1. This was a warning that the Stage II trainer may have found a local optima. Further attempts were made to determine whether a better network could be found, but these alternate solutions only marginally lowered the sum of squared errors.

The trained network was serialized and stored into four files:

the network file - NeuralNetworkEx1.ser,
the trainer file - NeuralNetworkTrainerEx1.ser,
the xData file - NeuralNetworkxDataEx1.ser, and
the yData file - NeuralNetworkyDataEx1.ser.

# *class* **Network**

Neural network base class.

## Declaration

public abstract class com.imsl.datamining.neural.Network
**extends** java.lang.Object
**implements** java.io.Serializable

## Constructor

- *Network*
  `public` **Network(** `)`

  - **Description**
    Default constructor for Network. Since this class is abstract, it cannot be instantiated directly; this constructor is used by constructors in classes derived from `Network`.

## Methods

- *computeStatistics*
  `public double[]` **computeStatistics(** `double[][]` **xData,** `double[][]` **yData** `)`

  - **Description**
    Computes error statistics.
    This is a static method that can be used to compute the statistics regardless of the training class used to train the `network`.
    Computes statistics related to the error. In this table, the observed values are $y_i$. The forecasted values are $\hat{y}_i$. The mean observed value is $\bar{y} = \sum_i y_i/NC$, where $N$ is the number of observations and $C$ is the number of classes per observation.

| Index | Name | Formula |
|---|---|---|
| 0 | SSE | $\frac{1}{2}\sum_i (y_i - \hat{y}_i)^2$ |
| 1 | RMS | $\frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)}$ |
| 2 | Laplacian | $\sum_i \lvert y_i - \hat{y}_i \rvert$ |
| 3 | Scaled Laplacian | $\frac{\sum_i \lvert y_i - \hat{y}_i \rvert}{\sum_i \lvert y_i - \bar{y}_i \rvert}$ |
| 4 | Max residual | $\max_i \lvert y_i - \hat{y}_i \rvert$ |

  - **Parameters**
    * `xData` – A `double` matrix containing the input values.

    * yData – A double array containing the observed values.
  – **Returns** – A double array containing the above described statistics.

---

• *createHiddenLayer*
public abstract HiddenLayer **createHiddenLayer**( )

  – **Description**
  Creates the next HiddenLayer in the Network.
  – **Returns** – The new HiddenLayer.

---

• *forecast*
public abstract double[] **forecast**( double[] x )

  – **Description**
  Returns a forecast for each of the Network's outputs computed from the trained Network.
  – **Parameters**
    * x – A double array of values with the same length and order as the training patterns used to train the Network.
  – **Returns** – A double array containing the forecasts for the output Perceptrons. Its length is equal to the number of output Perceptrons.

---

• *getForecastGradient*
public abstract double[] **getForecastGradient**( double[] x )

  – **Description**
  Returns the first derivatives with respect to the *weight*s evaluated at x.
  – **Parameters**
    * x – A double array which specifies the input values at which the *gradient* is to be evaluated. It must have the same length and order as the training patterns used to train the Network.
  – **Returns** – A double array containing the derivative values. The $i$-th entry in this array contains $dN(xData, weights)/d\, weights[i]$. Its length is equal to the number of *weight*s in the Network.

---

• *getInputLayer*
public abstract InputLayer **getInputLayer**( )

  – **Description**
  Returns the InputLayer object.
  – **Returns** – The Network InputLayer.

---

• *getLinks*
public abstract Link[] **getLinks**( )

– **Description**
Returns an array containing the `Link` objects in the `Network`.
– **Returns** – An array of `Links` associated with this `Network`.

---

- *getNumberOfInputs*
`public abstract int` **getNumberOfInputs**`( )`
  – **Description**
  Returns the number of `Network` inputs.
  – **Returns** – An `int` which contains the number of inputs.

---

- *getNumberOfLinks*
`public abstract int` **getNumberOfLinks**`( )`
  – **Description**
  Returns the number of `Network Links` among the `nodes`.
  – **Returns** – An `int` which contains the number of `Links` in the `Network`.

---

- *getNumberOfOutputs*
`public abstract int` **getNumberOfOutputs**`( )`
  – **Description**
  Returns the number of `Network` output `Perceptrons`.
  – **Returns** – An `int` which contains the number of outputs.

---

- *getNumberOfWeights*
`public abstract int` **getNumberOfWeights**`( )`
  – **Description**
  Returns the number of *weight*s in the `Network`.
  – **Returns** – An `int` which contains the number of *weight*s associated with this `Network`.

---

- *getOutputLayer*
`public abstract OutputLayer` **getOutputLayer**`( )`
  – **Description**
  Returns the `OutputLayer`.
  – **Returns** – The `Network OutputLayer`.

---

- *getPerceptrons*
`public abstract Perceptron[]` **getPerceptrons**`( )`
  – **Description**
  Returns an array containing the `Perceptrons` in the `Network`.

– **Returns** – An array of `Perceptrons` associated with this `Network`.

---

- *getWeights*
  `public abstract double[]` **getWeights**`( )`
  - **Description**
    Returns the *weight*s.
  - **Returns** – A `double` array containing the *weight*s associated with `Network Links`.

---

- *setWeights*
  `public abstract void` **setWeights**`( double[]` **weights** `)`
  - **Description**
    Sets the *weight*s.
  - **Parameters**
    * `weights` – A `double` array which specifies the *weight*s to be associated with `Network Links`.

### Example: Network

This example uses a network previously trained and serialized into four files to obtain information about the network and forecasts. Training was done using the code for the FeedForwardNetwork Example 1.

The network training targets were generated using the relationship:

y = 10*X1 + 20*X2 + 30*X3 + 2.0*X4, where

X1 to X3 are the three binary columns, corresponding to categories 1 to 3 of the nominal attribute, and X4 is the scaled continuous attribute.

The structure of the network consists of four input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure illustrates this structure:

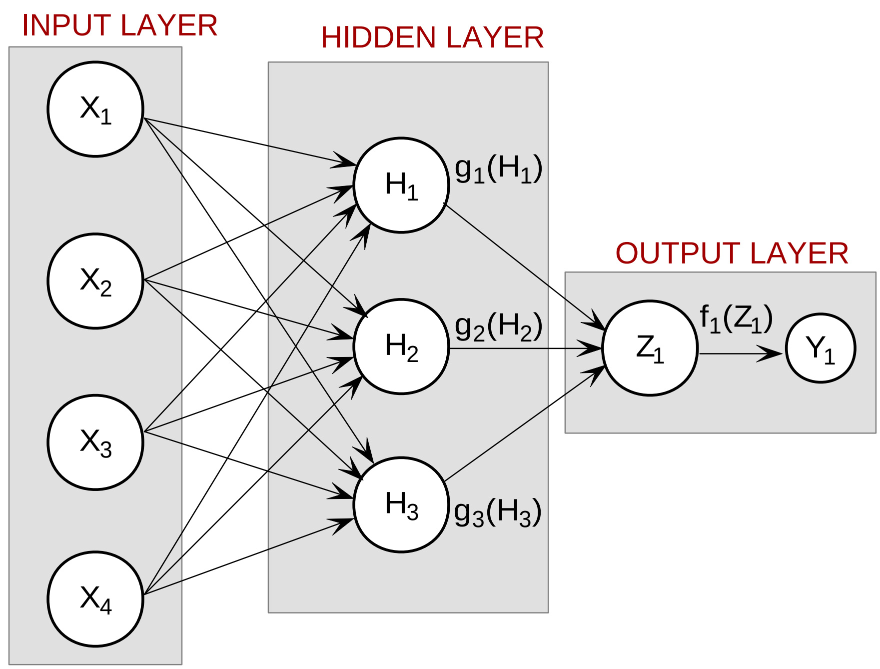



**Figure 10. An example 2-layer Feed Forward Neural Network with 4 Inputs**

All perceptrons were trained using a Linear Activation Function. Forecasts are generated for 9 conditions, corresponding to the following conditions:
Nominal Class 1-3 with the Continuous Input Attribute = 0
Nominal Class 1-3 with the Continuous Input Attribute = 5.0
Nominal Class 1-3 with the Continuous Input Attribute = 10.0

Note that the network training statistics retrieved from the serialized network confirm that this is the same network used in the previous example. Obtaining these statistics requires retrieval of the training patterns which were serialized and stored into separate files. This information is not serialized with the network, nor with the trainer.

```
import com.imsl.datamining.neural.*;
import java.io.*;

//*****************************************************************************
// Two Layer Feed-Forward Network with 4 inputs: 1 categorical with 3 classes
// encoded using binary encoding and 1 continuous input, and 1 output
// target (continuous).  There is a perfect linear relationship between
// the input and output variables:
//
```

```
// MODEL:  Y = 10*X1 + 20*X2 + 30*X3 + 2*X4
//
// Variables X1-X3 are the binary encoded nominal variable and X4 is the
// continuous variable.
//
// This example uses Linear Activation in both the hidden and output layers
// The network uses a 2-layer configuration, one hidden layer and one
// output layer.  The hidden layer consists of 3 perceptrons.  The output
// layer consists of a single output perceptron.
// The input from the continuous variable is scaled to [0,1] before training
// the network.  Training is done using the Quasi-Newton Trainer.
// The network has a total of 19 weights.
// Since the network target is a linear combination of the network inputs, and
// since all perceptrons use linear activation, the network is able to forecast
// the every training target exactly.  The largest residual is 2.78E-08.
//****************************************************************************

public class NetworkEx1 implements Serializable {
 // ***********************************************************************
 // MAIN
 // ***********************************************************************
    public static void main(String[] args) throws Exception {
       double xData[][]; // Input  Attributes for Training Patterns
       double yData[][]; // Output Attributes for Training Patterns
       double weight[];  // network weights
       double gradient[];// network gradient after training
       // Input Attributes for Forecasting
       double x[][] = { {1,0,0,0.0}, {0,1,0,0.0}, {0,0,1,0.0},
                        {1,0,0,5.0}, {0,1,0,5.0}, {0,0,1,5.0},
                        {1,0,0,10.0}, {0,1,0,10.0}, {0,0,1,10.0}
                      };
       double xTemp[], y[];// Temporary areas for storing forecasts
       int i, j;           // loop counters
       // Names of Serialized Files
       String networkFileName = "FeedForwardNetworkEx1.ser"; // the network
       String trainerFileName = "FeedForwardTrainerEx1.ser"; // the trainer
       String xDataFileName   = "FeedForwardxDataEx1.ser";   // xData
       String yDataFileName   = "FeedForwardyDataEx1.ser";   // yData
    // ***********************************************************************
    // READ THE TRAINED NETWORK FROM THE SERIALIZED NETWORK OBJECT
    // ***********************************************************************
       System.out.println("--> Reading Trained Network from " +
```

```
                        networkFileName);
   Network network = (Network)read(networkFileName);
// ***********************************************************************
// READ THE SERIALIZED XDATA[][] AND YDATA[][] ARRAYS OF TRAINING
// PATTERNS.
// ***********************************************************************
   System.out.println("--> Reading xData from " +
                        xDataFileName);
   xData = (double[][])read(xDataFileName);
   System.out.println("--> Reading yData from " +
                        yDataFileName);
   yData = (double[][])read(yDataFileName);
// ***********************************************************************
// READ THE SERIALIZED TRAINER OBJECT
// ***********************************************************************
   System.out.println("--> Reading Network Trainer from " +
                        trainerFileName);
   Trainer trainer = (Trainer)read(trainerFileName);
// ***********************************************************************
// DISPLAY TRAINING STATISTICS
// ***********************************************************************
   double stats[] = network.computeStatistics(xData, yData);
   // Display Network Errors
   System.out.println("***********************************************");
   System.out.println("--> SSE:                       "+(float)stats[0]);
   System.out.println("--> RMS:                       "+(float)stats[1]);
   System.out.println("--> Laplacian Error:           "+(float)stats[2]);
   System.out.println("--> Scaled Laplacian Error:    "+(float)stats[3]);
   System.out.println("--> Largest Absolute Residual: "+(float)stats[4]);
   System.out.println("***********************************************");
   System.out.println("");
   // ***********************************************************************
   // OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS
   // ***********************************************************************
   System.out.println("--> Getting Network Information");
   // Get weights
   weight   = network.getWeights();
   // Get number of weights = number of gradients
   int nWeights = network.getNumberOfWeights();
   // Obtain Gradient Vector
   gradient = trainer.getErrorGradient();
   // Print Network Weights and Gradients
```

```
      System.out.println(" ");
      System.out.println("--> Network Weights and Gradients:");
      for(i=0; i < nWeights; i++){
       System.out.println("w["+i+"]=" + (float)weight[i]+
                          " g["+i+"]="+(float)gradient[i]);
      }
   // ******************************************************************
   // OBTAIN AND DISPLAY FORECASTS FOR THE LAST 10 TRAINING TARGETS
   // ******************************************************************
      // Get number of network inputs
      int nInputs  = network.getNumberOfInputs();
      // Get number of network outputs
      int nOutputs = network.getNumberOfOutputs();
      xTemp = new double[nInputs]; // temporary x space for forecast inputs
      y     = new double[nOutputs];// temporary y space for forecast output
      System.out.println(" ");
      // Obtain example forecasts for input attributes = x[]
      // X1-X3 are binary encoded for one nominal variable with 3 classes
      // X4 is a continuous input attribute ranging from 0-10.  During
      // training, X4 was scaled to [0,1] by dividing by 10.
      for(i=0;i<9;i++){
         for(j=0;j<nInputs;j++) xTemp[j] = x[i][j];
         xTemp[nInputs-1] = xTemp[nInputs-1]/10.0;
         y = network.forecast(xTemp);
         System.out.print("--> X1="+(int)x[i][0]+
                            " X2="+(int)x[i][1]+" X3="+(int)x[i][2]+
                            " | X4="+x[i][3]);
         System.out.println(" | y="+
              (float)(10.0*x[i][0]+20.0*x[i][1]+30.0*x[i][2]+2.0*x[i][3])+
              "| Forecast="+(float)y[0]);
      }
   }
// ***********************************************************************
// READ SERIALIZED NETWORK FROM A FILE
// ***********************************************************************
   static public Object read(String filename)
      throws IOException, ClassNotFoundException {
      FileInputStream fis = new FileInputStream(filename);
      ObjectInputStream ois = new ObjectInputStream(fis);
      Object obj = ois.readObject();
      ois.close();
      fis.close();
```

```
        return obj;
    }
}
```

### Output

```
--> Reading Trained Network from FeedForwardNetworkEx1.ser
--> Reading xData from FeedForwardxDataEx1.ser
--> Reading yData from FeedForwardyDataEx1.ser
--> Reading Network Trainer from FeedForwardTrainerEx1.ser
**********************************************
--> SSE:                      1.0134443E-15
--> RMS:                      2.0074636E-19
--> Laplacian Error:          3.0058038E-7
--> Scaled Laplacian Error:   3.5352343E-10
--> Largest Absolute Residual: 2.784276E-8
**********************************************

--> Getting Network Information

--> Network Weights and Gradients:
w[0]=-1.4917853 g[0]=-2.6110852E-8
w[1]=-1.4917853 g[1]=-2.6110852E-8
w[2]=-1.4917853 g[2]=-2.6110852E-8
w[3]=1.6169184 g[3]=6.182032E-8
w[4]=1.6169184 g[4]=6.182032E-8
w[5]=1.6169184 g[5]=6.182032E-8
w[6]=4.725622 g[6]=-5.273859E-8
w[7]=4.725622 g[7]=-5.273859E-8
w[8]=4.725622 g[8]=-5.273859E-8
w[9]=6.217407 g[9]=-8.7338103E-10
w[10]=6.217407 g[10]=-8.7338103E-10
w[11]=6.217407 g[11]=-8.7338103E-10
w[12]=1.0722584 g[12]=-1.6909877E-7
w[13]=1.0722584 g[13]=-1.6909877E-7
w[14]=1.0722584 g[14]=-1.6909877E-7
w[15]=3.8507552 g[15]=-1.7029118E-8
w[16]=3.8507552 g[16]=-1.7029118E-8
w[17]=3.8507552 g[17]=-1.7029118E-8
w[18]=2.4117248 g[18]=-1.5881545E-8
```

```
--> X1=1 X2=0 X3=0 | X4=0.0 | y=10.0| Forecast=10.0
--> X1=0 X2=1 X3=0 | X4=0.0 | y=20.0| Forecast=20.0
--> X1=0 X2=0 X3=1 | X4=0.0 | y=30.0| Forecast=30.0
--> X1=1 X2=0 X3=0 | X4=5.0 | y=20.0| Forecast=20.0
--> X1=0 X2=1 X3=0 | X4=5.0 | y=30.0| Forecast=30.0
--> X1=0 X2=0 X3=1 | X4=5.0 | y=40.0| Forecast=40.0
--> X1=1 X2=0 X3=0 | X4=10.0 | y=30.0| Forecast=30.0
--> X1=0 X2=1 X3=0 | X4=10.0 | y=40.0| Forecast=40.0
--> X1=0 X2=0 X3=1 | X4=10.0 | y=50.0| Forecast=50.0
```

## *class* **FeedForwardNetwork**

A representation of a feed forward neural network.

A `Network` contains an `InputLayer`, an `OutputLayer` and zero or more `HiddenLayers`. The `null` `InputLayer` and `OutputLayer` are automatically created by the com.imsl.datamining.neural.Network constructor. The `InputNodes` are added using the `getInputLayer().createInputs(nInputs)` method. Output `Perceptrons` are added using the `getOutputLayer().createPerceptrons(nOutputs)`, and `HiddenLayers` can be created using the `createHiddenLayer().createPerceptrons(nPerceptrons)` method.

The `InputLayer` contains `InputNodes`. The `HiddenLayers` and `OutputLayers` contain `Perceptron` nodes. These `Nodes` are created using factory methods in the `Layers`.

The `Network` also contains `Links` between `Nodes`. `Links` are created by methods in this class.

Each `Link` has a *weight* and *gradient* value. Each `Perceptron` node has a *bias* value. When the `Network` is trained, the *weight* and *bias* values are used as initial guesses. After the `Network` is trained the *weight*, *gradient* and *bias* values are set to the values computed by the training.

A feed forward network is a network in which links are only allowed from one layer to a following layer.

### Declaration

public class com.imsl.datamining.neural.FeedForwardNetwork
**extends** com.imsl.datamining.neural.Network (page 1154)

## Constructor

- *FeedForwardNetwork*
  `public` **FeedForwardNetwork**`( )`
  - **Description**
    Creates a new instance of `FeedForwardNetwork`.

## Methods

- *createHiddenLayer*
  `public HiddenLayer` **createHiddenLayer**`( )`
  - **Description**
    Creates a `HiddenLayer`.
  - **Returns** – A `HiddenLayer` object which specifies a neural network hidden layer.

- *findLink*
  `public Link` **findLink**`( Node` **from**`, Node` **to** `)`
  - **Description**
    Returns the `Link` between two `Nodes`.
  - **Parameters**
    * `from` – The origination `Node`.
    * `to` – The destination `Node`.
  - **Returns** – A `Link` between the two `Nodes`, or `null` if no such `Link` exists.

- *findLinks*
  `public Link[]` **findLinks**`( Node` **to** `)`
  - **Description**
    Returns all of the `Links` to a given `Node`.
  - **Parameters**
    * `to` – A `Node` who's `Links` are to be determined.
  - **Returns** – An array of `Links` containing all of the `Links` to the given `Node`.

- *forecast*
  `public double[]` **forecast**`( double[]` **x** `)`
  - **Description**
    Computes a forecast using the `Network`.

- **Parameters**
  - x – A double array of values to which the Nodes in the InputLayer are to be set.
- **Returns** – A double array containing the values of the Nodes in the OutputLayer.

---

- *getForecastGradient*
  public double[] **getForecastGradient**( double[] **xData** )
  - **Description**
    Returns the *gradient* with respect to the *weights*.
  - **Parameters**
    - xData – A double array which specifies the input values at which the gradient is to be evaluated.
  - **Returns** – A double array containing the gradient values. The $i$-th entry in this array contains $dN(xData, weights)/d\, weights[i]$.

---

- *getHiddenLayers*
  public HiddenLayer[] **getHiddenLayers**( )
  - **Description**
    Returns the HiddenLayers in this network.
  - **Returns** – An array of HiddenLayers in this network.

---

- *getInputLayer*
  public InputLayer **getInputLayer**( )
  - **Description**
    Returns the InputLayer.
  - **Returns** – The neural network InputLayer.

---

- *getLinks*
  public Link[] **getLinks**( )
  - **Description**
    Return all of the Links in this Network.
  - **Returns** – An array of Links containing all of the Links in this Network.

---

- *getNumberOfInputs*
  public int **getNumberOfInputs**( )
  - **Description**
    Returns the number of inputs to the Network.
  - **Returns** – An int containing the number of inputs to the Network.

- *getNumberOfLinks*
  `public int` **getNumberOfLinks**`( )`
  - **Description**
    Returns the number of `Links` in the `Network`.
  - **Returns** – An `int` which contains the number of `Links` in the `Network`.

- *getNumberOfOutputs*
  `public int` **getNumberOfOutputs**`( )`
  - **Description**
    Returns the number of outputs from the `Network`.
  - **Returns** – An `int` containing the number of outputs from the `Network`.

- *getNumberOfWeights*
  `public int` **getNumberOfWeights**`( )`
  - **Description**
    Returns the number of *weight*s in the `Network`.
  - **Returns** – An `int` which contains the number of *weight*s in the `Network`.

- *getOutputLayer*
  `public OutputLayer` **getOutputLayer**`( )`
  - **Description**
    Returns the `OutputLayer`.
  - **Returns** – The neural network `OutputLayer`.

- *getPerceptrons*
  `public Perceptron[]` **getPerceptrons**`( )`
  - **Description**
    Returns the `Perceptrons` in this `Network`.
  - **Returns** – An array of `Perceptrons` in this network.

- *getWeights*
  `public double[]` **getWeights**`( )`
  - **Description**
    Returns the *weight*s for the `Links` in this network.
  - **Returns** – An array of `doubles` containing the *weight*s. The array contains the *weight*s for each `Link` followed by the `Perceptron` *bias* values. The `Link` *weight*s are the order in which the `Links` were created. The *weight* values are first, followed by the *bias* values in the `HiddenLayers` and then the *bias* values in the `OutputLayer`, and then by the order in which the `Perceptrons` were created.

---

- *link*
  `public Link` **link**`( Node` **from**`, Node` **to** `)`

  - **Description**
    Establishes a `Link` between two `Nodes`. Any existing `Link` between these `Nodes` is removed.
  - **Parameters**
    * `from` – The origination `Node`.
    * `to` – The destination `Node`.
  - **Returns** – A `Link` between the two `Nodes`.

---

- *link*
  `public Link` **link**`( Node` **from**`, Node` **to**`, double` **weight** `)`

  - **Description**
    Establishes a `Link` between two `Nodes` with a specified `weight`.
  - **Parameters**
    * `from` – The origination `Node`.
    * `to` – The destination `Node`.
    * `weight` – A `double` which specifies the *weight* to be given the `Link`.
  - **Returns** – A `Link` between the two `Nodes`.

---

- *linkAll*
  `public void` **linkAll**`( )`

  - **Description**
    For each `Layer` in the `Network`, link each `Node` in the `Layer` to each `Node` in the next `Layer`.

---

- *linkAll*
  `public void` **linkAll**`( Layer` **from**`, Layer` **to** `)`

  - **Description**
    Link all of the `Nodes` in one `Layer` to all of the `Nodes` in another `Layer`.
  - **Parameters**
    * `from` – The origination `Layer`.
    * `to` – The destination `Layer`.

---

- *remove*
  `public void` **remove**`( Link` **link** `)`

  - **Description**
    Removes a `Link` from the network.

– **Parameters**
  * `link` – The `Link` deleted from the network.

---

• *setWeights*
`public void setWeights( double[] weights )`

– **Description**
  Sets the *weight*s for the `Links` in this `Network`.

– **Parameters**
  * `weights` – A `double` array containing the *weight*s in the same order as getWeights.

---

• *validateLink*
`protected void validateLink( Node from, Node to ) throws java.lang.IllegalArgumentException`

– **Description**
  Checks that a `Link` between two `Nodes` is valid.
  In a feed forward network a link must be from a node in one layer to a node in a later layer. Intermediate layers can be skipped, but a link cannot go backward.

– **Parameters**
  * `from` – The origination `Node`.
  * `to` – The destination `Node`.

– **Throws**
  * `java.lang.IllegalArgumentException` – is thrown if the `Link` is not valid

### Example: FeedForwardNetwork

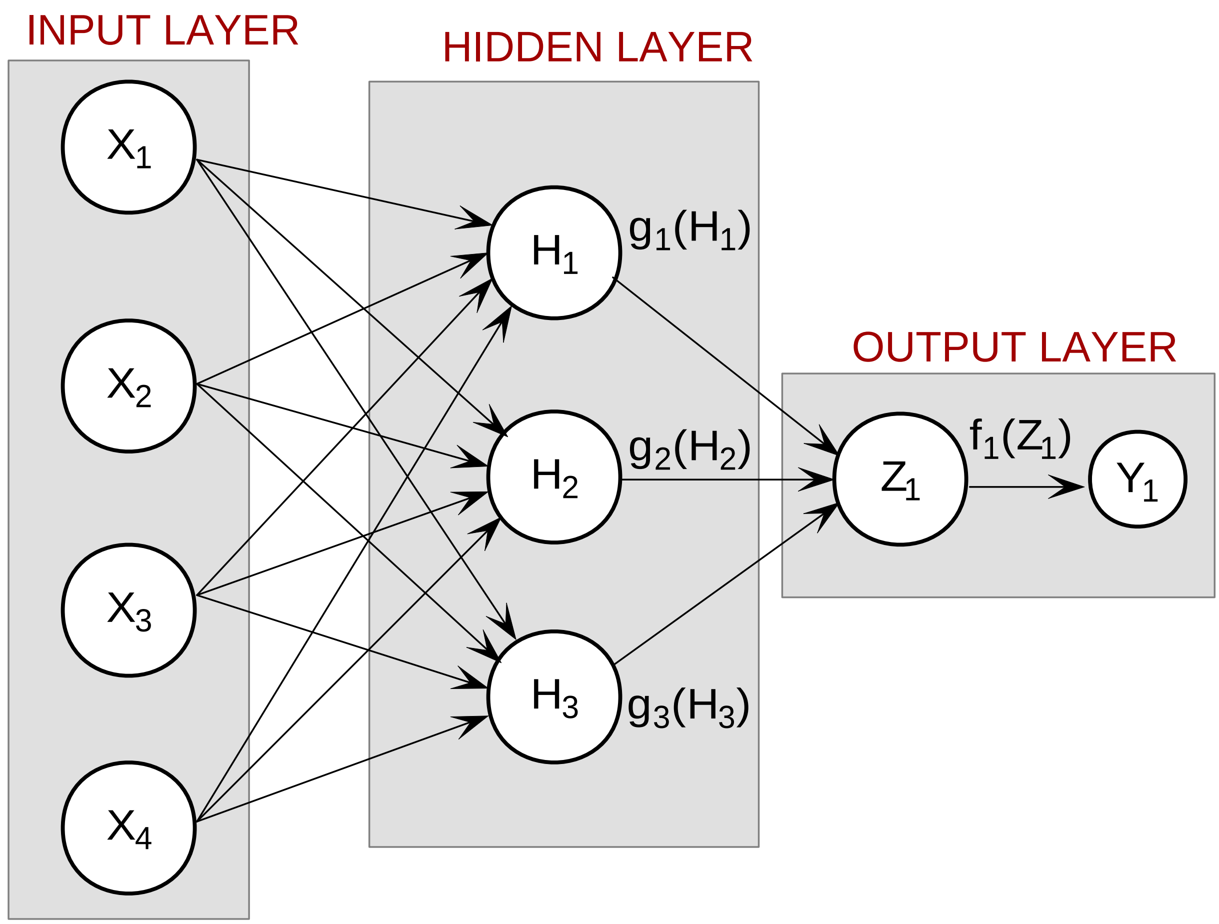
This example trains a 2-layer network using 100 training patterns from one nominal and one continuous input attribute. The nominal attribute has three classifications which are encoded using binary encoding. This results in three binary network input columns. The continuous input attribute is scaled to fall in the interval [0,1].

The network training targets were generated using the relationship:

y = 10*X1 + 20*X2 + 30*X3 + 2.0*X4, where

X1-X3 are the three binary columns, corresponding to categories 1-3 of the nominal attribute, and X4 is the scaled continuous attribute.

The structure of the network consists of four input nodes and two layers, with three perceptrons in the hidden layer and one in the output layer. The following figure illustrates this structure:




There are a total of 19 weights in this network. The activations functions are all linear. Since the target output is a linear function of the input attributes, linear activation functions guarantee that the network forecasts will exactly match their targets. Of course, this same result could have been obtained using linear multiple regression. Training is conducted using the quasi-newton trainer.

```
import com.imsl.datamining.neural.*;
import java.io.*;
import java.util.logging.*;

//*****************************************************************************
// Two Layer Feed-Forward Network with 4 inputs: 1 nominal with 3 categories,
// encoded using binary encoding, 1 continuous input attribute, and 1 output
// target (continuous).
// There is a perfect linear relationship between the input and output
```

```
// variables:
//
// MODEL:  Y = 10*X1+20*X2+30*X3+2*X4
//
// Variables X1-X3 are the binary encoded nominal variable and X4 is the
// continuous variable.
//*****************************************************************************

public class FeedForwardNetworkEx1 implements Serializable {

// Network Settings
    private FeedForwardNetwork network;
    private static int nObs         =100; // number of training patterns
    private static int nInputs      =  4; // four inputs
    private static int nCategorical =  3; // three categorical attributes
    private static int nContinuous  =  1; // one continuous input attribute
    private static int nOutputs     =  1; // one continuous output
    private static int nLayers      =  2; // number of perceptron layers
    private static int nPerceptrons =  3; // perceptrons in hidden layer
    private static boolean trace    = true; // Turns on/off training log
    private static Activation hiddenLayerActivation = Activation.LINEAR;
    private static Activation outputLayerActivation = Activation.LINEAR;
    private static String errorMsg  = "";
    // Error Status Messages for the Least Squares Trainer
    private static String errorMsg0 =
      "--> Least Squares Training Completed Successfully";
    private static String errorMsg1 =
      "--> Scaled step tolerance was satisfied.  The current solution \n"+
      "may be an approximate local solution, or the algorithm is making\n"+
      "slow progress and is not near a solution, or the Step Tolerance\n"+
      "is too big";
    private static String errorMsg2 =
      "--> Scaled actual and predicted reductions in the function are\n"+
      "less than or equal to the relative function convergence\n"+
      "tolerance RelativeTolerance";
    private static String errorMsg3 =
      "--> Iterates appear to be converging to a noncritical point.\n"+
      "Incorrect gradient information, a discontinuous function,\n"+
      "or stopping tolerances being too tight may be the cause.";
    private static String errorMsg4 =
      "--> Five consecutive steps with the maximum stepsize have\n"+
      "been taken.  Either the function is unbounded below, or has\n"+
```

```
      "a finite asymptote in some direction, or the maximum stepsize\n"+
      "is too small.";
    private static String errorMsg5 =
      "--> Too many iterations required";

// categoricalAtt[]: A 2D matrix of values for the categorical training
//                   attribute. In this example, the single categorical
//                   attribute has 3 categories that are encoded using
//                   binary encoding for input into the network.
//                   {1,0,0} = category 1, {0,1,0} = category 2, and
//                   {0,0,1} = category 3.
    private static double categoricalAtt[][] =
      {
      {1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},
      {1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},
      {1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},
      {1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0},

      {0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},
      {0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},
      {0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},{0,1,0},
      {0,1,0},{0,1,0},{0,1,0},

      {0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},
      {0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},
      {0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},
      {0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1},{0,0,1}
      };
//
// contAtt[]:  A matrix of values for the continuous training attribute
//
    private static double contAtt[] = {
      4.007054658,7.10028447,4.740350984,5.714553211,6.205437459,
      2.598930065,8.65089967,5.705787357,2.513348184,2.723795955,
      4.1829356,1.93280416,0.332941608,6.745567628,5.593588463,
      7.273544478,3.162117939,4.205381208,0.16414745,2.883418275,
      0.629342241,1.082223406,8.180324708,8.004894314,7.856215418,
      7.797143157,8.350033996,3.778254431,6.964837082,6.13938006,
      0.48610387,5.686627923,8.146173848,5.879852653,4.587492779,
      0.714028533,7.56324211,8.406012623,4.225261454,6.369220241,
      4.432772218,9.52166984,7.935791508,4.557155333,7.976015058,
      4.913538616,1.473658514,2.592338905,1.386872932,7.046051685,
```

```
      1.432128376,1.153580985,5.6561491,3.31163251,4.648324851,
      5.042514515,0.657054195,7.958308093,7.557870384,7.901990083,
      5.2363088,6.95582150,8.362167045,4.875903563,1.729229471,
      4.380370223,8.527875685,2.489198107,3.711472959,4.17692681,
      5.844828801,4.825754155,5.642267843,5.339937786,4.440813223,
      1.615143829,7.542969339,8.100542684,0.98625265,4.744819569,
      8.926039258,8.813441887,7.749383991,6.551841576,8.637046998,
      4.560281415,1.386055087,0.778869034,3.883379045,2.364501589,
      9.648737525,1.21754765,3.908879368,4.253313879,9.31189696,
      3.811953836,5.78471629,3.414486452,9.345413015,1.024053777
      };
//
// outs[]:  A 2D matrix containing the training outputs for this network
// In this case there is an exact linear relationship between these
// outputs and the inputs:  outs = 10*X1+20*X2+30*X3+2*X4, where
// X1-X3 are the categorical variables and X4=contAtt
//
    private static double outs[] = {
      18.01410932,24.20056894,19.48070197,21.42910642,22.41087492,
      15.19786013,27.30179934,21.41157471,15.02669637,15.44759191,
      18.3658712,13.86560832,10.66588322,23.49113526,21.18717693,
      24.54708896,16.32423588,18.41076242,10.3282949,15.76683655,
      11.25868448,12.16444681,26.36064942,26.00978863,25.71243084,
      25.59428631,26.70006799,17.55650886,23.92967416,22.27876012,
      10.97220774,21.37325585,26.2923477,21.75970531,19.17498556,
      21.42805707,35.12648422,36.81202525,28.45052291,32.73844048,
      28.86554444,39.04333968,35.87158302,29.11431067,35.95203012,
      29.82707723,22.94731703,25.18467781,22.77374586,34.09210337,
      22.86425675,22.30716197,31.3122982,26.62326502,29.2966497,
      30.08502903,21.31410839,35.91661619,35.11574077,35.80398017,
      30.4726176,33.91164302,36.72433409,29.75180713,23.45845894,
      38.76074045,47.05575137,34.97839621,37.42294592,38.35385362,
      41.6896576,39.65150831,41.28453569,40.67987557,38.88162645,
      33.23028766,45.08593868,46.20108537,31.9725053,39.48963914,
      47.85207852,47.62688377,45.49876798,43.10368315,47.274094,
      39.1205628,32.77211017,31.55773807,37.76675809,34.72900318,
      49.29747505,32.4350953,37.81775874,38.50662776,48.62379392,
      37.62390767,41.56943258,36.8289729,48.69082603,32.04810755
      };
 // *****************************************************************
 // MAIN
 // *****************************************************************
```

```
public static void main(String[] args) throws Exception {

   double weight[];  // network weights
   double gradient[];// network gradient after training
   double x[];       // temporary x space for generating forecasts
   double y[];       // temporary y space for generating forecasts
   double xData[][]; // Input  Attributes for Trainer
   double yData[][]; // Output Attributes for Trainer
   int i, j;         // array indicies
   int nWeights = 0; // Number of weights obtained from network
   String networkFileName = "FeedForwardNetworkEx1.ser";
   String trainerFileName = "FeedForwardTrainerEx1.ser";
   String xDataFileName   = "FeedForwardxDataEx1.ser";
   String yDataFileName   = "FeedForwardyDataEx1.ser";
   String trainLogName    = "FeedForwardTraining.log";
// ******************************************************************
// PREPROCESS TRAINING PATTERNS
// ******************************************************************
   System.out.println("--> Starting Preprocessing of Training Patterns");
   xData = new double[nObs][nInputs];
   yData = new double[nObs][nOutputs];
   for(i=0; i < nObs; i++) {
      for(j=0; j < nCategorical; j++){
         xData[i][j] = categoricalAtt[i][j];
      }
      xData[i][nCategorical] = contAtt[i]/10.0; // Scale continuous input
      yData[i][0] = outs[i];  // outputs are unscaled
   }
// ******************************************************************
// CREATE FEEDFORWARD NETWORK
// ******************************************************************
   System.out.println("--> Creating Feed Forward Network Object");
   FeedForwardNetwork network = new FeedForwardNetwork();
   // setup input layer with number of inputs = nInputs = 4
   network.getInputLayer().createInputs(nInputs);
   // create a hidden layer with nPerceptrons=3 perceptrons
   network.createHiddenLayer().createPerceptrons(nPerceptrons);
   // create output layer with nOutputs=1 output perceptron
   network.getOutputLayer().createPerceptrons(nOutputs);
   // link all inputs and perceptrons to all perceptrons in the next layer
   network.linkAll();
   // Get Network Perceptrons for Setting Their Activation Functions
```

```
   Perceptron perceptrons[] = network.getPerceptrons();
   // Set all perceptrons to linear activation
   for (i=0; i < perceptrons.length-1; i++) {
      perceptrons[i].setActivation(hiddenLayerActivation);
   }
   perceptrons[perceptrons.length-1].setActivation(outputLayerActivation);
   System.out.println("--> Feed Forward Network Created with 2 Layers");
// ************************************************************************
// TRAIN NETWORK USING QUASI-NEWTON TRAINER
// ************************************************************************
   System.out.println("--> Training Network using Quasi-Newton Trainer");
   // Create Trainer
   QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
   // Set Training Parameters
   trainer.setMaximumTrainingIterations(1000);
   // If tracing is requested setup training logger
   if (trace) {
        try {
            Handler handler = new FileHandler(trainLogName);
            Logger logger = Logger.getLogger("com.imsl.datamining.neural");
            logger.setLevel(Level.FINEST);
            logger.addHandler(handler);
            handler.setFormatter(QuasiNewtonTrainer.getFormatter());
            System.out.println("--> Training Log Created in "+
                                   trainLogName);
        } catch (Exception e) {
            System.out.println("--> Cannot Create Training Log.");
        }
    }
   // Train Network
   trainer.train(network, xData, yData);
   // Check Training Error Status
   switch(trainer.getErrorStatus()){
      case 0: errorMsg = errorMsg0;
              break;
      case 1: errorMsg = errorMsg1;
              break;
      case 2: errorMsg = errorMsg2;
              break;
      case 3: errorMsg = errorMsg3;
              break;
      case 4: errorMsg = errorMsg4;
```

```
           break;
      case 5: errorMsg = errorMsg5;
           break;
      default:errorMsg = errorMsg0;
   }
   System.out.println(errorMsg);
// **********************************************************************
// DISPLAY TRAINING STATISTICS
// **********************************************************************
   double stats[] = network.computeStatistics(xData, yData);
   // Display Network Errors
   System.out.println("***********************************************");
   System.out.println("--> SSE:                       "+(float)stats[0]);
   System.out.println("--> RMS:                       "+(float)stats[1]);
   System.out.println("--> Laplacian Error:           "+(float)stats[2]);
   System.out.println("--> Scaled Laplacian Error:    "+(float)stats[3]);
   System.out.println("--> Largest Absolute Residual: "+(float)stats[4]);
   System.out.println("***********************************************");
   System.out.println("");
   // **********************************************************************
   // OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS
   // **********************************************************************
   System.out.println("--> Getting Network Weights and Gradients");
   // Get weights
   weight   = network.getWeights();
   // Get number of weights = number of gradients
   nWeights = network.getNumberOfWeights();
   // Obtain Gradient Vector
   gradient = trainer.getErrorGradient();
   // Print Network Weights and Gradients
   System.out.println(" ");
   System.out.println("--> Network Weights and Gradients:");
   System.out.println("***********************************************");
   for(i=0; i < nWeights; i++){
      System.out.println("w["+i+"]=" + (float)weight[i]+
                        " g["+i+"]="+(float)gradient[i]);
   }
   System.out.println("***********************************************");
// **********************************************************************
// SAVE THE TRAINED NETWORK BY SAVING THE SERIALIZED NETWORK OBJECT
// **********************************************************************
   System.out.println("\n--> Saving Trained Network into "+
```

```
                            networkFileName);
        write(network, networkFileName);
        System.out.println("--> Saving xData into "+
                            xDataFileName);
        write(xData, xDataFileName);
        System.out.println("--> Saving yData into "+
                            yDataFileName);
        write(yData, yDataFileName);
        System.out.println("--> Saving Network Trainer into "+
                            trainerFileName);
        write(trainer, trainerFileName);
    }
// **************************************************************************
// WRITE SERIALIZED NETWORK TO A FILE
// **************************************************************************
    static public void write(Object obj, String filename)
        throws IOException {
        FileOutputStream fos = new FileOutputStream(filename);
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(obj);
        oos.close();
        fos.close();
    }
}
```

## Output

```
--> Starting Preprocessing of Training Patterns
--> Creating Feed Forward Network Object
--> Feed Forward Network Created with 2 Layers
--> Training Network using Quasi-Newton Trainer
--> Training Log Created in FeedForwardTraining.log
--> Least Squares Training Completed Successfully
**********************************************
--> SSE:                       1.0134443E-15
--> RMS:                       2.0074636E-19
--> Laplacian Error:           3.0058038E-7
--> Scaled Laplacian Error:    3.5352343E-10
--> Largest Absolute Residual: 2.784276E-8
**********************************************
```

```
--> Getting Network Weights and Gradients

--> Network Weights and Gradients:
*********************************************
w[0]=-1.4917853 g[0]=-2.6110852E-8
w[1]=-1.4917853 g[1]=-2.6110852E-8
w[2]=-1.4917853 g[2]=-2.6110852E-8
w[3]=1.6169184 g[3]=6.182032E-8
w[4]=1.6169184 g[4]=6.182032E-8
w[5]=1.6169184 g[5]=6.182032E-8
w[6]=4.725622 g[6]=-5.273859E-8
w[7]=4.725622 g[7]=-5.273859E-8
w[8]=4.725622 g[8]=-5.273859E-8
w[9]=6.217407 g[9]=-8.7338103E-10
w[10]=6.217407 g[10]=-8.7338103E-10
w[11]=6.217407 g[11]=-8.7338103E-10
w[12]=1.0722584 g[12]=-1.6909877E-7
w[13]=1.0722584 g[13]=-1.6909877E-7
w[14]=1.0722584 g[14]=-1.6909877E-7
w[15]=3.8507552 g[15]=-1.7029118E-8
w[16]=3.8507552 g[16]=-1.7029118E-8
w[17]=3.8507552 g[17]=-1.7029118E-8
w[18]=2.4117248 g[18]=-1.5881545E-8
*********************************************

--> Saving Trained Network into FeedForwardNetworkEx1.ser
--> Saving xData into FeedForwardxDataEx1.ser
--> Saving yData into FeedForwardyDataEx1.ser
--> Saving Network Trainer into FeedForwardTrainerEx1.ser
```

# *class* **Layer**

The base class for `Layer`s in a neural network.

## Declaration

public abstract class com.imsl.datamining.neural.Layer
**extends** java.lang.Object
**implements** java.io.Serializable

### Constructor

---

- *Layer*
  `protected` **Layer**( `FeedForwardNetwork` **network** )
  - **Description**
    Constructs a `Layer`.
  - **Parameters**
    * `network` – The `FeedForwardNetwork` to which this `Layer` is to be associated.

### Methods

---

- *addNode*
  `protected void` **addNode**( `Node` **node** )
  - **Description**
    Associates a `Perceptron` with this `Layer`.
  - **Parameters**
    * `node` – A `Node` to associate with this `Layer`.

---

- *getIndex*
  `public int` **getIndex**( )
  - **Description**
    Returns the *index* of this `Layer`.
  - **Returns** – An `int` which contains the value of property *index*.

---

- *getNodes*
  `public Node[]` **getNodes**( )
  - **Description**
    Return a list of the `Perceptrons` in this `Layer`.
  - **Returns** – An array containing the `Nodes` associated with this `Layer`.

## *class* InputLayer

Input layer in a neural network. An `InputLayer` is automatically created by `Network`.

## Declaration

public class com.imsl.datamining.neural.InputLayer
**extends** com.imsl.datamining.neural.Layer (page 1178)

## Methods

---

- *createInput*
  `public InputNode` **createInput**`( )`
  - **Description**
    Creates an `InputNode` in the `InputLayer` of the neural network.

---

- *createInputs*
  `public InputNode[]` **createInputs**`( int` **n** `)`
  - **Description**
    Creates a number of `InputNodes` in this `Layer` of the neural network.
  - **Parameters**
    * `n` – An `int` which specifies the number of `InputNodes` to be created in this `layer`.
  - **Returns** – An array containing the created `InputNodes`.

---

- *getNodes*
  `public Node[]` **getNodes**`( )`
  - **Description**
    Return the `Perceptrons` in the `InputLayer`.
  - **Returns** – An `InputNode` array containing the `Nodes` in the `InputLayer`.

# *class* HiddenLayer

Hidden layer in a neural network. This is created by a factory method in `Network`.

## Declaration

public class com.imsl.datamining.neural.HiddenLayer
**extends** com.imsl.datamining.neural.Layer (page 1178)

## Methods

---

- *createPerceptron*
  `public Perceptron` **createPerceptron**`( )`
  - **Description**
    Creates a `Perceptron` in this `Layer` of the neural network. The created `Perceptron` uses the logistic activation function and has an initial *bias* value of zero.

---

- *createPerceptron*
  `public Perceptron` **createPerceptron**`( Activation` **activation**`, double` **bias** `)`
  - **Description**
    Creates a `Perceptron` in this `Layer` with a specified activation function and *bias*.
  - **Parameters**
    * `activation` – The `Activation` object which specifies the activation function to be used.
    * `bias` – A `double` which specifies the initial value for the *bias*.

---

- *createPerceptrons*
  `public Perceptron[]` **createPerceptrons**`( int` **n** `)`
  - **Description**
    Creates a number of `Perceptrons` in this `Layer` of the neural network. The created `Perceptrons` use the logistic activation function and have an initial *bias* value of zero.
  - **Parameters**
    * `n` – An `int` which specifies the number of `Perceptrons` to be created.
  - **Returns** – An array containing the created `Perceptrons`.

---

- *createPerceptrons*
  `public Perceptron[]` **createPerceptrons**`( int` **n**`, Activation` **activation**`, double` **bias** `)`
  - **Description**
    Creates a number of `Perceptrons` in this `Layer` with the specified *bias*.
  - **Parameters**
    * `n` – An `int` which specifies the number of Perceptrons to be created.
    * `activation` – The `Activation` object which specifies the action function to be used.

    * `bias` – A `double` containing the initial value to be applied as the *bias* values for the `Perceptrons`.
  - **Returns** – An array containing the created `Perceptrons`.

## *class* OutputLayer

Output layer in a neural network. An empty `OutputLayer` is automatically created by `FeedForwardNetwork`.

### Declaration

public class com.imsl.datamining.neural.OutputLayer
**extends** com.imsl.datamining.neural.Layer (page 1178)

### Methods

---

- *createPerceptron*
  `public Perceptron` **createPerceptron**`( )`
  - **Description**
    Creates a `Perceptron` in this `Layer` of the neural network. By default, the created `Perceptron` uses the linear activation function and has an initial *bias* value of zero.

---

- *createPerceptron*
  `public Perceptron` **createPerceptron**`( Activation` **activation**`, double` **bias** `)`
  - **Description**
    Creates a `Perceptron` in this `Layer` with a specified `Activation` and `bias`.
  - **Parameters**
    * `activation` – The `Activation` object which specifies the action function to be used.
    * `bias` – A `double` which specifies the initial value for the *bias* for this `Perceptron`.

---

- *createPerceptrons*
  `public Perceptron[]` **createPerceptrons**`( int` **n** `)`

– **Description**
Creates a number of `Perceptrons` in this `Layer` of the neural network. By default, they will use linear activation and a zero initial *bias*.

– **Parameters**
  * `n` – An `int` which specifies the number of `Perceptrons` to be created in this `layer`.

– **Returns** – An array containing the created `Perceptrons`.

---

• *createPerceptrons*
`public Perceptron[]` **createPerceptrons**`( int` **n**`, Activation` **activation**`, double` **bias** `)`

– **Description**
Creates a number of `Perceptrons` in this `Layer` with specified `activation` and `bias`.

– **Parameters**
  * `n` – An `int` which specifies the number of `Perceptrons` to be created.
  * `activation` – The `Activation` object which indicates the action function to be used.
  * `bias` – A `double` which specifies the initial *bias* for the `Perceptrons`.

– **Returns** – An array containing the created `Perceptrons`.

---

• *getNodes*
`public Node[]` **getNodes**`( )`

– **Description**
Return the `Perceptrons` in the `OutputLayer`.
This method overides the method in com.imsl.datamining.neural.Layer to return the `Perceptrons` in an `OutputPerceptron` array.

– **Returns** – An `OutputPerceptron[]` array containing the `Nodes` in the `OutputLayer`.

## *class* Node

A `Node` in a neural network.

`Node` is an abstract class that serves as the base class for the concrete classes `InputNode` and `Perceptron`.

## Declaration

public abstract class com.imsl.datamining.neural.Node
**extends** java.lang.Object
**implements** java.io.Serializable

## Method

---

- *getLayer*
  `public Layer` **getLayer**`( )`
  - **Description**
    Returns the `Layer` in which this `Node` exists.
  - **Returns** – The `Layer` associated with this `Node`.

# *class* InputNode

A `Node` in the `InputLayer`.

`InputNode`s are not created directly. Instead factory methods in `InputLayer` are used to create `InputNode`s within the `InputLayer`. For example, createInput() creates a single `InputNode`.

## Declaration

public class com.imsl.datamining.neural.InputNode
**extends** com.imsl.datamining.neural.Node (page 1183)

## Methods

---

- *getValue*
  `public double` **getValue**`( )`
  - **Description**
    Returns the value of this `node`.
  - **Returns** – A `double` which contains the value of this `InputNode`.

- *setValue*
  `public void` **setValue**`( double` **value** `)`
  - **Description**
    Sets the value of this `Node`.
  - **Parameters**
    * `value` – A `double` which specifies the new value of this `InputNode`.

## *class* Perceptron

A Perceptron node in a neural network. Perceptrons are created by factory methods in a network layer.

Each perceptron has an activation function ($g$) and a bias ($\mu$). The value of a perceptron is given by $g(\sum_i w_i X_i + \mu)$, where $X_i$ are the values of nodes input to this perceptron with weights $w_i$.

Network training will use existing bias values for the starting values for the trainer. Upon completion of network training, the bias values are set to the values computed by the trainer.

### Declaration

public class com.imsl.datamining.neural.Perceptron
**extends** com.imsl.datamining.neural.Node (page 1183)

### Methods

---

- *getActivation*
  `public Activation` **getActivation**`( )`
  - **Description**
    Returns the activation function.
  - **Returns** – An `Activation` object indicating the activation function.

---

- *getBias*
  `public double` **getBias**`( )`
  - **Description**
    Returns the bias for this perceptron.

– **Returns** – A `double` representing the bias for this perceptron.

---

- *setActivation*
  `public void` **setActivation**`( Activation` **activation** `)`

  – **Description**
  Sets the activation function.

  – **Parameters**

  * `activation` – An `Activation` object which represents the activation $g$ to be used by this perceptron.

---

- *setBias*
  `public void` **setBias**`( double` **bias** `)`

  – **Description**
  Sets the bias for this perceptron.

  – **Parameters**

  * `bias` – A `double` scalar value to which the bias is to be set. The bias has a default value of 0.

## *class* OutputPerceptron

A Perceptron in the output layer. `OutputPerceptrons` are created by factory methods in `Outputlayer`.

`OutputPerceptrons` are not created directly. Instead factory methods in `OutputLayer` are used to create `OutputPerceptrons` within the `OutputLayer`. For example, `OutputLayer.createPerceptron()` creates a single `OutputPerceptron`.

### Declaration

public class com.imsl.datamining.neural.OutputPerceptron
**extends** com.imsl.datamining.neural.Perceptron (page 1185)

### Method

---

- *getValue*
  `public double` **getValue**`( )`

– **Description**
  Returns the value of the output perceptron determined using the current network state and inputs.
– **Returns** – A `double` value of the output perceptron determined using the current network state and inputs.

# *interface* Activation

Interface implemented by perceptron activation functions.

Standard activation functions are defined as static members of this interface. New activation functions can be defined by implementing a method, `g(double x)`, returning the value and a method, `derivative(double x, double y)`, returning the derivative of `g` evaluated at $x$ where $y = g(x)$.

## Declaration

public interface com.imsl.datamining.neural.Activation
**implements** java.io.Serializable

## Fields

---

- `long` **serialVersionUID**
- `Activation` **LINEAR**
  – The identity activation function, $g(x) = x$.
- `Activation` **LOGISTIC**
  – The logistic activation function, $g(x) = \frac{1}{1+e^{-x}}$.
- `Activation` **LOGISTIC_TABLE**
  – The logistic activation function computed using a table. This is an approximation to the logistic function that is faster to compute.
  This version of the logistic function differs from the exact version by at most 4.0e-9.
  Networks trained using this activation should not use `Activation.LOGISTIC` for forecasting. Forecasting should be done using the specific function supplied during training.

- `Activation` **TANH**
  - The hyperbolic tangent activation function, $g(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.
- `Activation` **SQUASH**
  - The squash activation function, $g(x) = \frac{x}{1+|x|}$

## Methods

---

- *derivative*
  `double` **`derivative`**`( double x, double y )`
  - **Description**
    Returns the value of the derivative of the activation function.
  - **Parameters**
    * `x` – A `double` which specifies the point at which the activation function is to be evaluated.
    * `y` – A `double` which specifies $y = g(x)$, the value of the activation function at $x$. This parameter is not mathematically required, but can sometimes be used to more quickly compute the derivative.
  - **Returns** – A `double` containing the value of the derivative of the activation function at `x`.

---

- *g*
  `double` **`g`**`( double x )`
  - **Description**
    Returns the value of the activation function.
  - **Parameters**
    * `x` – A `double` is the point at which the activation function is to be evaluated.
  - **Returns** – A `double` containing the value of the activation function at `x`.

# *class* Link

A link in a neural network.

`Link` objects are not created directly. Instead, they are created by factory methods in `FeedForwardNetwork`.

The most useful method is linkAll which creates `Link` objects connecting every `Node` in each `Layer` to every `Node` in the next `Layer` .

The method link(Node,Node) creates a `Link` from a `Node` to any `Node` in a later `Layer`.

The method findLink(Node,Node) returns the `Link` connecting two `Nodes` in the `Network`.

The method remove(Link) removes a `Link` from the `Network`.

Each `Link` object contains a *weight*. Weights are used in computing `Perceptron` values.

### Declaration

public class com.imsl.datamining.neural.Link
**extends** java.lang.Object
**implements** java.io.Serializable

### Methods

---

- *getFrom*
  `public Node` **getFrom**`( )`
  - **Description**
    Returns the origination `Node` for this `Link`.
  - **Returns** – A `Node` which is the origination `Node` for this `Link`.

---

- *getTo*
  `public Node` **getTo**`( )`
  - **Description**
    Returns the destination `Node` for this `Link`.
  - **Returns** – A `Node` which is the destination `Node` for this `Link`.

---

- *getWeight*
  `public double` **getWeight**`( )`
  - **Description**
    Returns the *weight* for this `Link`.
  - **Returns** – A `double` which contains the *weight* attributed to this `Node`.

---

- *setWeight*
  `public void` **setWeight**`( double` **weight** `)`

– **Description**
Sets the *weight* for this `Link`.
– **Parameters**
  * `weight` – A `double` which specifies the weight to attribute to this `Link`.

# *interface* Trainer

Interface implemented by classes used to train a network. The method `train` is used to adjust the weights in a network to best fit a set of observed data. After a network is trained, the other methods in this interface can be used to check the quality of the fit.

## Declaration

public interface com.imsl.datamining.neural.Trainer
**implements** java.io.Serializable

## Methods

---

- *getErrorGradient*
  `double[]` **getErrorGradient**`( )`
  – **Description**
  Returns the value of the gradient of the error function with respect to the weights.
  – **Returns** – A `double` array, the length of the number of weights, containing the value of the gradient of the error function with respect to the weights at the computed optimal point. Before training, `null` is returned.

- *getErrorStatus*
  `int` **getErrorStatus**`( )`
  – **Description**
  Returns the error status.
  – **Returns** – An `int` specifying the error. If there was no error, zero is returned. A non-zero return indicates a potential problem with the trainer.

- *getErrorValue*
  `double` **getErrorValue**`( )`

- **Description**
  Returns the value of the error function minimized by the trainer.
- **Returns** – A `double` indicating the final value of the error function from the last training. Before training, `NaN` is returned.

---

- *train*
  `void` **train**`( Network` **network**`, double[][]` **xData**`, double[][]` **yData** `)`
  - **Description**
    Trains the neural network using supplied training patterns.
  - **Parameters**
    * `network` – A `Network` object, which is the `Network` to be trained.
    * `xData` – A `double` matrix containing the input training patterns. The number of columns in `xData` must equal the number of nodes in the input layer. Each row of `xData` contains a training pattern.
    * `yData` – A `double` matrix containing the output training patterns. The number of columns in `yData` must equal the number of perceptrons in the output layer. Each row of `yData` contains a training pattern.

## *class* QuasiNewtonTrainer

Trains a network using the quasi-Newton method, `MinUnconMultiVar`.

The Java Logging API can be used to trace the performance training. The name of this logger is `com.imsl.datamining.QuasiNewtonTrainer` Accumulated levels of detail correspond to Java's `FINE`, `FINER`, and `FINEST` logging levels with `FINE` yielding the smallest amount of information and `FINEST` yielding the most. The levels of output yield the following:

| **Level** | **Output** |
|---|---|
| `FINE` | A message on entering and exiting method `train`, and any exceptions from and the exit status of `MinUnconMultiVar` |
| `FINER` | All of the messages in `FINE`, the input settings, and a summary report with the statistics from `Network.computeStatistics()`, the number of function evaluations and the elapsed time. |
| `FINEST` | All of the messages in `FINER`, and a table of the computed weights and their gradient values. |

### Declaration

public class com.imsl.datamining.neural.QuasiNewtonTrainer
**extends** java.lang.Object
**implements** Trainer, java.io.Serializable

### Inner Class

*interface* **QuasiNewtonTrainer.Error**

Error function to be minimized by trainer. This trainer attempts to solve the problem

$$\min_{w} \sum_{i=0}^{n-1} e(y_i, \hat{y}_i)$$

where $w$ are the weights, $n$ is the number of training patterns, $y_i$ is a training target output and $\hat{y}_i$ is its forecast value.

This interface defines the function $e(y, \hat{y})$ and its derivative with respect to its computed value, $de/d\hat{y}$.

#### Declaration

public static interface com.imsl.datamining.neural.QuasiNewtonTrainer.Error
**implements** java.io.Serializable

#### Methods

---

- *error*
  `double` **error(** `double` **computed,** `double` **expected** `)`
  - **Description**
    Returns the contribution to the error from a single training output target. This is the function $e(y_i, \hat{y}_i)$.
  - **Parameters**
    - `computed` – A `double` representing the computed value.
    - `expected` – A `double` representing the expected value.
  - **Returns** – A `double` representing the contribution to the error from a single training output target.

- *errorGradient*
  `double` **errorGradient**`(` `double` **computed**`,` `double` **expected** `)`

  – **Description**
    Returns the derivative of the error function with respect to the forecast output.
  – **Parameters**
    * `computed` – A `double` representing the computed value.
    * `expected` – A `double` representing the expected value.
  – **Returns** – A `double` representing the derivative of the error function with respect to the forecast output.

## Field

- public static final `QuasiNewtonTrainer.Error` **SUM_OF_SQUARES**
  – Compute the sum of squares error. The sum of squares error term is $e(y, \hat{y}) = (y - \hat{y})^2/2$.
    This is the default `Error` object used by `QuasiNewtonTrainer`.

## Constructor

- *QuasiNewtonTrainer*
  `public` **QuasiNewtonTrainer**`( )`

  – **Description**
    Constructs a `QuasiNewtonTrainer` object.

## Methods

- *getError*
  `public QuasiNewtonTrainer.Error` **getError**`( )`

  – **Description**
    Returns the function used to compute the error to be minimized.
  – **Returns** – The `Error` object containing the function to be minimized.

- *getErrorGradient*
  `public double[]` **getErrorGradient**`( )`
  - **Description**
    Returns the value of the gradient of the error function with respect to the weights.
  - **Returns** – A `double` array whose length is equal to the number of network weights, containing the value of the gradient of the error function with respect to the weights. Before training, null is returned.

---

- *getErrorStatus*
  `public int` **getErrorStatus**`( )`
  - **Description**
    Returns the error status from the trainer.
  - **Returns** – An `int` representing the error status from the trainer. Zero indicates that no errors were encountered during training. Any non-zero value indicates that some error condition arose during training. In many cases the trainer is able to recover from these conditions and produce a well-trained network.

| **Error Status** | **Condition** |
|---|---|
| 0 | No error occurred during training. |
| 1 | The last global step failed to locate a lower point than the current error value. The current solution may be an approximate solution and no more accuracy is possible, or the step tolerance may be too large. |
| 2 | Relative function convergence; both the actual and predicted relative reductions in the error function are less than or equal to the relative function convergence tolerance. |
| 3 | Scaled step tolerance satisfied; the current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or the step tolerance is too big. |
| 4 | Optimizer threw a `MinUnconMultiVar.FalseConvergenceException`. |
| 5 | Optimizer threw a `MinUnconMultiVar.MaxIterationsException`. |
| 6 | Optimizer threw a `MinUnconMultiVar.UnboundedBelowException`. |

- *getErrorValue*
`public double` **getErrorValue**`( )`
  - **Description**
    Returns the final value of the error function.
  - **Returns** – A `double` representing the final value of the error function from the last training. Before training, `NaN` is returned.

- *getFormatter*
`public static java.util.logging.Formatter` **getFormatter**`( )`
  - **Description**
    Returns the logging formatter object. `Logger` support requires JDK1.4. Use with earlier versions returns null.
    The returned `Formatter` is used as input to setFormatter to format the output log.
  - **Returns** – The `Formatter` object, if present, or `null`.

- *getLogger*
`public static java.util.logging.Logger` **getLogger**`( )`
  - **Description**
    Returns the `Logger` object. This is the `Logger` used to trace this class. It is named `com.imsl.datamining.neural.QuasiNewtonTrainer`.
  - **Returns** – The `Logger` object, if present, or `null`.

- *getTrainingIterations*
`public int` **getTrainingIterations**`( )`
  - **Description**
    Returns the number of iterations used during training.
  - **Returns** – An `int` representing the number of iterations used during training.

- *setError*
`public void` **setError**`( QuasiNewtonTrainer.Error` **error** `)`
  - **Description**
    Sets the function used to compute the network error.
  - **Parameters**
    * `error` – The `Error` object containing the function to be used to compute the network error. The default is to compute the sum of squares error, `SUM_OF_SQUARES`.

- *setFalseConvergenceTolerance*
  `public void` **setFalseConvergenceTolerance**`( double` **falseConvergenceTolerance** `)`
  - **Description**
    Set the false convergence tolerance for the `Trainer`.
  - **Parameters**
    * `falseConvergenceTolerance` – A `double` specifying the false convergence tolerance. Default: 2.22044604925031308e-14.

- *setGradientTolerance*
  `public void` **setGradientTolerance**`( double` **gradientTolerance** `)`
  - **Description**
    Set the gradient tolerance.
  - **Parameters**
    * `gradientTolerance` – A `double` specifying the gradient tolerance. Default: cube root of machine precision.

- *setMaximumStepsize*
  `public void` **setMaximumStepsize**`( double` **maximumStepsize** `)`
  - **Description**
    Sets the maximum step size.
  - **Parameters**
    * `maximumStepsize` – A nonnegative `double` value specifying the maximum allowable step size in the optimizer.

- *setMaximumTrainingIterations*
  `public void` **setMaximumTrainingIterations**`( int` **maximumTrainingIterations** `)`
  - **Description**
    Sets the maximum number of iterations to use in a training.
  - **Parameters**
    * `maximumTrainingIterations` – An `int` representing the maximum number of training iterations. Default: 100.

- *setRelativeTolerance*
  `public void` **setRelativeTolerance**`( double` **relativeTolerance** `)`
  - **Description**
    Sets the relative tolerence.

– **Parameters**
  * `relativeTolerance` – A `double` representing the relative error tolerance. It must be in the interval [0,1]. Its default value is 3.66685e-11.

---

- *setStepTolerance*
  `public void` **setStepTolerance**`( double` **stepTolerance** `)`

  – **Description**
  Sets the scaled step tolerance.
  The second stopping criterion for com.imsl.math.MinUnconMultiVar, the optimizer used by this `Trainer`, is that the scaled distance between the last two steps be less than the step tolerance.
  – **Parameters**
    * `stepTolerance` – A `double` which is the step tolerance. Default: 3.66685e-11.

---

- *train*
  `public void` **train**`( Network` **network**`, double[][]` **xData**`, double[][]` **yData** `)`

  – **Description**
  Trains the neural network using supplied training patterns.
  Each row of `xData` and `yData` contains a training pattern. The number of rows in these two arrays must be at least equal to the number of weights in the network.
  – **Parameters**
    * `network` – The `Network` to be trained.
    * `xData` – An input `double` matrix containing training patterns. The number of columns in `xData` must equal the number of nodes in the input layer.
    * `yData` – An output `double` matrix containing output training patterns. The number of columns in `yData` must equal the number of perceptrons in the output layer.

## *class* LeastSquaresTrainer

Trains a `FeedForwardNetwork` using a Levenberg-Marquardt algorithm for minimizing a sum of squares error.

The Java Logging API can be used to trace the performance training. The name of this `Logger` is `com.imsl.datamining.LeatSquaresTrainer`. Accumulated levels of detail correspond to Java's `FINE`, `FINER`, and `FINEST` logging levels with `FINE` yielding the smallest amount of information and `FINEST` yielding the most. The levels of output yield the following:

| Level | Output |
|---|---|
| `FINE` | A message on entering and exiting method `train`, and any exceptions from and the exit status of `NonlinLeastSquares` |
| `FINER` | All of the messages in `FINE`, the input settings, and a summary report with the statistics from `Network.computeStatistics()` and the elapsed time. |
| `FINEST` | All of the messages in `FINER`, and a table of the computed *weight*s and their *gradient* values. |

## Declaration

public class com.imsl.datamining.neural.LeastSquaresTrainer
**extends** java.lang.Object
**implements** Trainer, java.io.Serializable

## Constructor

- *LeastSquaresTrainer*
  `public` **LeastSquaresTrainer**`( )`
  - **Description**
    Creates a `LeastSquaresTrainer`.

## Methods

- *getErrorGradient*
  `public double[]` **getErrorGradient**`( )`
  - **Description**
    Returns the value of the *gradient* of the error function with respect to the *weight*s.

– **Returns** – A `double` array whose length is equal to the number of network *weights*, containing the value of the *gradient* of the error function with respect to the *weights*. Before training, `null` is returned.

---

- *getErrorStatus*
`public int` **getErrorStatus**`( )`

  – **Description**
  Returns the error status from the trainer.

  – **Returns** – An `int` which contains the error status. Zero indicates that no errors were encountered during training. Any non-zero value indicates that some error condition arose during training.
  In many cases the trainer is able to recover from these conditions and produce a well-trained network.

| **Value** | **Meaning** |
|---|---|
| 0 | All convergence tests were met. |
| 1 | Scaled step tolerance was satisfied. The current point may be an approximate local solution, or the algorithm is making very slow progress and is not near a solution, or `StepTolerance` is too big. |
| 2 | Scaled actual and predicted reductions in the function are less than or equal to the relative function convergence tolerance `RelativeTolerance`. |
| 3 | Iterates appear to be converging to a noncritical point. Incorrect gradient information, a discontinuous function, or stopping tolerances being too tight may be the cause. |
| 4 | Five consecutive steps with the maximum stepsize have been taken. Either the function is unbounded below, or has a finite asymptote in some direction, or the maximum stepsize is too small. |
| 5 | Too many iterations required |

---

- *getErrorValue*
`public double` **getErrorValue**`( )`

  – **Description**

Returns the final value of the error function.

- **Returns** – A `double` containing the final value of the error function from the last training. Before training, `NaN` is returned.

---

- *getFormatter*

`public static java.util.logging.Formatter` **getFormatter**`( )`

  - **Description**
    Returns the logging `Formatter` object. `Logger` support requires JDK1.4. Use with earlier versions returns `null`.
    The returned `Formatter` is used as input to setFormatter to format the output log.
  - **Returns** – A `Formatter` object, if present, or `null` .

---

- *getLogger*

`public static java.util.logging.Logger` **getLogger**`( )`

  - **Description**
    Returns the `Logger` object. This is the `Logger` used to trace this class. It is named `com.imsl.datamining.neural.QuasiNewtonTrainer`.
  - **Returns** – The `Logger` object, if present, or `null` .

---

- *setFalseConvergenceTolerance*

`public void` **setFalseConvergenceTolerance**`( double` **falseConvergenceTolerance** `)`

  - **Description**
    Set the false convergence tolerance.
  - **Parameters**
    * `falseConvergenceTolerance` – a `double` specifying the false convergence tolerance. Default: 1.0e-14.

---

- *setGradientTolerance*

`public void` **setGradientTolerance**`( double` **gradientTolerance** `)`

  - **Description**
    Set the *gradient* tolerance.
  - **Parameters**
    * `gradientTolerance` – A `double` specifying the *gradient* tolerance. Default: 2.0e-5.

---

- *setInitialTrustRegion*

`public void` **setInitialTrustRegion**`( double` **initialTrustRegion** `)`

– **Description**
Sets the intial trust region.

– **Parameters**

∗ `initialTrustRegion` – A `double` which specifies the initial trust region radius. Default: unlimited trust region.

---

- *setMaximumStepsize*
`public void` **setMaximumStepsize**`( double` **maximumStepsize** `)`

– **Description**
Sets the maximum step size.

– **Parameters**

∗ `maximumStepsize` – A nonnegative `double` value specifying the maximum allowable stepsize in the optimizer. Default: $10^3||w||_2$, where $w$ are the values of the weights in the network when training starts.

---

- *setMaximumTrainingIterations*
`public void` **setMaximumTrainingIterations**`( int` **maximumSolverIterations** `)`

– **Description**
Sets the maximum number of iterations used by the nonlinear least squares solver.

– **Parameters**

∗ `maximumSolverIterations` – An `int` which specifies the maximum number of iterations to be used by the nonlinear least squares solver. Its default value is 1000.

---

- *setRelativeTolerance*
`public void` **setRelativeTolerance**`( double` **relativeTolerance** `)`

– **Description**
Sets the relative tolerance.

– **Parameters**

∗ `relativeTolerance` – A `double` which specifies the relative error tolerance. It must be in the interval [0,1]. Its default value is 1.0e-20.

---

- *setStepTolerance*
`public void` **setStepTolerance**`( double` **stepTolerance** `)`

– **Description**
Set the step tolerance used to step between *weight*s.

– **Parameters**

- * `stepTolerance` – A `double` which specifies the scaled step tolerance to use when changing the *weights*. Default: 1.0e-5.

---

- *train*
  `public void` **train**`( Network` **network**`, double[][]` **xData**`, double[][]` **yData** `)`
  - **Description**
    Trains the neural network using supplied training patterns.
    Each row of `xData` and `yData` contains a training pattern. These number of rows in two arrays must be equal.
  - **Parameters**
    * `network` – The `Network` to be trained.
    * `xData` – A `double` matrix which contains the input training patterns. The number of columns in `xData` must equal the number of `Nodes` in the `InputLayer`.
    * `yData` – A `double` matrix which contains the output training patterns. The number of columns in `yData` must equal the number of `Perceptrons` in the `OutputLayer`.

# *class* EpochTrainer

Two-stage training using randomly selected training patterns in stage I. The epoch trainer, is a meta-trainer that combines two trainers. The first trainer is used on a series of randomly selected subsets of the training patterns. For each subset, the weights are initialized to their initial values plus a random offset.

Stage 2 then refines the result found in stage 1. The best result from the stage 1 trainings is used as the initial guess with the second trainer operating on the full set of training patterns. Stage 2 is optional, if the second trainer is `null` then the best stage 1 result is returned as the epoch trainer's result.

The Java Logging API can be used to trace the performance training. The name of this logger is `com.imsl.datamining.EpochTrainer`. Accumulated levels of detail correspond to Java's `FINE`, `FINER`, and `FINEST` logging levels with `FINE` yielding the smallest amount of information and `FINEST` yielding the most. The levels of output yield the following:

| Level | Output |
|---|---|
| `FINE` | A message on entering and exiting method `train`, a message entering and exiting both stages 1 and 2, and a summary report (based on computeStatistics) upon completion of training. |
| `FINER` | All of the messages in `FINE`, a message entering and exiting each epoch in stage 1, the input settings, the value of the function being minimized in stage 1 for each epoch, a time stamp at the start of each iteration in stage 1 and at the beginning and end of stage 2, and (if there is a stage 2) a summary at the end of stage 1. |
| `FINEST` | All of the messages in `FINER` and a table of the computed *weight*s and their *gradient* values. |

## Declaration

public class com.imsl.datamining.neural.EpochTrainer
**extends** java.lang.Object
**implements** Trainer, java.io.Serializable

## Constructors

---

- *EpochTrainer*
  `public` **EpochTrainer**`( Trainer` **stage1Trainer** `)`
  - **Description**
    Creates a single stage `EpochTrainer`. Stage 2 training is bypassed.
  - **Parameters**
    * `stage1Trainer` – The `Trainer` used in stage I.

---

- *EpochTrainer*
  `public` **EpochTrainer**`( Trainer` **stage1Trainer**`, Trainer` **stage2Trainer** `)`
  - **Description**
    Creates an two-stage `EpochTrainer`.
  - **Parameters**

* `stage1Trainer` – The stage I `Trainer`.
* `stage2Trainer` – The stage II `Trainer`, or `null` if stage II is to be bypassed.

## Methods

---

- *getEpochSize*
  `public int` **getEpochSize**`( )`
  - **Description**
    Returns the number of sample training patterns in each stage 1 epoch.
  - **Returns** – An `int` which contains the number of sample training patterns in each stage I epoch.

---

- *getErrorGradient*
  `public double[]` **getErrorGradient**`( )`
  - **Description**
    Returns the value of the *gradient* of the error function with respect to the *weight*s.
  - **Returns** – A `double` array whose length is equal to the number of `Network` *weight*s, containing the value of the *gradient* of the error function with respect to the *weight*s. Before training, `null` is returned.

---

- *getErrorStatus*
  `public int` **getErrorStatus**`( )`
  - **Description**
    Returns the training error status.
  - **Returns** – An `int` containing the error status from stage 2. If there is no stage 2 then the number of stage 1 epochs that returned a non-zero error status is returned.

---

- *getErrorValue*
  `public double` **getErrorValue**`( )`
  - **Description**
    Returns the value of the error function.
  - **Returns** – A `double` containing final value of the error function from the last training. Before training, `NaN` is returned.

---

- *getFormatter*
  `public static java.util.logging.Formatter` **getFormatter**`( )`

  - **Description**
    Returns the logging `Formatter` object. `Logger` support requires JDK1.4. Use with earlier versions returns `null` .
    The returned `Formatter` is used as input to setFormatter to format the output log.
  - **Returns** – The `Formatter` object, if present, or `null` otherwise.

---

- *getLogger*
  `public static java.util.logging.Logger` **getLogger**`( )`

  - **Description**
    Returns the `Logger` object. This is the `Logger` used to trace this class. It is named com.imsl.datamining.neural.QuasiNewtonTrainer.
  - **Returns** – The `Logger` object, if present, or `null` otherwise.

---

- *getNumberOfEpochs*
  `public int` **getNumberOfEpochs**`( )`

  - **Description**
    Returns the number of epochs used during stage I training.
  - **Returns** – An `int` which contains the number of epochs used during stage I training.

---

- *getRandom*
  `public com.imsl.stat.Random` **getRandom**`( )`

  - **Description**
    Returns the random number generator used to perturb the stage 1 guesses.
  - **Returns** – The `Random` object used to generate stage 1 perturbations.

---

- *setEpochSize*
  `public void` **setEpochSize**`( int` **epochSize** `)`

  - **Description**
    Sets the number of randomly selected training patterns in stage 1 epoch.
  - **Parameters**
    * `epochSize` – An `int` which specifies the number of sample training patterns in each stage I epoch.

---

- *setNumberOfEpochs*
  `public void` **setNumberOfEpochs**`( int` **numberOfEpochs** `)`

– **Description**
  Sets the number of epochs.

– **Parameters**
  * `numberOfEpochs` – An `int` which specifies the number of epochs to be used during stage I training.

---

• *setRandom*

`public void` **setRandom**`( com.imsl.stat.Random` **random** `)`

– **Description**
  Sets the random number generator used to perturb the initial stage 1 guesses.

– **Parameters**
  * `random` – The `Random` object used to set the random number generator.

---

• *setRandomSamples*

`public void` **setRandomSamples**`( com.imsl.stat.Random` **randomA**`, com.imsl.stat.Random` **randomB** `)`

– **Description**
  Sets the random number generators used to select random training patterns in stage 1. The two random number generators should be independent.

– **Parameters**
  * `randomA` – A `Random` object which is the first random number generator.
  * `randomB` – A `Random` object which is the second random number generator, independent of `randomA`.

---

• *train*

`public void` **train**`( Network` **network**`, double[][]` **xData**`, double[][]` **yData** `)`

– **Description**
  Trains the neural network using supplied training patterns.

– **Parameters**
  * `network` – The `Network` to be trained.
  * `xData` – A `double` matrix specifying the input training patterns. The number of columns in `xData` must equal the number of `Nodes` in the `InputLayer`.
  * `yData` – A `double` containing the output training patterns. The number of columns in `yData` must equal the number of `Perceptrons` in the `OutputLayer`. Each row of `xData` and `yData` contains a training pattern. These number of rows in two arrays must be equal.

## *class* **ScaleFilter**

Scales or unscales continuous data prior to its use in neural network training, testing, or forecasting.

Bounded scaling is used to ensure that the values in the scaled array fall between a lower and upper bound. The scale limits have the following interpretation:

| Argument | Interpretation |
|---|---|
| `realMin` | The lowest value expected in `x`. |
| `realMax` | The largest value expected in `x`. |
| `targetMin` | The lower bound for the values in the scaled data. |
| `targetMax` | The upper bound for the values in the scaled data. |

The scale limits are set using the method `setBounds`.

The specific scaling used is controlled by the argument `scalingMethod` used when constructing the filter object. If `scalingMethod` is `NO_SCALING`, then no scaling is performed on the data.

If the `scalingMethod` is `BOUNDED_SCALING` then the bounded method of scaling and unscaling is applied to x. The scaling operation is conducted using the scale limits set in method `setBounds`, using the following calculation:

$$z = r(x - realMin) + targetMin,$$

where

$$r = \frac{targetMax - targetMin}{realMax - realMin}.$$

If `scalingMethod` is one of `UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV`, `UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD`, `BOUNDED_Z_SCORE_SCALING_MEAN_STDEV`, or `BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD`, then the z-score method of scaling is used. These calculations are based upon the following scaling calculation:

$$z = \frac{(x - a)}{b},$$

where $a$ is a measure of center for $x$, and $b$ is a measure of the spread of $x$.

If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV`, or `BOUNDED_Z_SCORE_SCALING_MEAN_STDEV`, then $a$ and $b$ are the arithmetic average and sample standard deviation of the training data.

If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD` or `BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD`, then $a$ and $b$ are the median and $\tilde{s}$, where $\tilde{s}$ is a robust estimate of the population standard deviation:

$$\tilde{s} = \frac{\text{MAD}}{0.6745}$$

where MAD is the Mean Absolute Deviation

$$\text{MAD} = median\{\mid x - median\{x\} \mid\}$$

The Mean Absolute Deviation is a robust measure of spread calculated by finding the median of the absolute value of differences between each non-missing value for the $i$th variable and the median of those values.

If the method `decode` is called then an unscaling operation is conducted by inverting using:

$$x = \frac{(z - targetMin)}{r} + realMin.$$

## Unbounded z-score Scaling

If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV` or `UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD`, then a scaling operation is conducted using the z-score calculation:

$$z = \frac{(x - center)}{spread},$$

If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV` then *center* is set equal to the arithmetic average $\bar{x}$ of `x`, and *spread* is set equal to the sample standard deviation of `x`. If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD` then *center* is set equal to the median $\tilde{m}$ of `x`, and *spread* is set equal to the Mean Absolute Difference (MAD).

The method `decode` can be used to unfilter data using the the inverse calculation for the above equation:

$$x = spread \cdot z + center.$$

## Bounded z-score Scaling

This method is essentially the same as the z-score calculation described above with additional scaling or unscaling using the scale limits set in method `setBounds`. The scaling operation is conducted using the well known z-score calculation:

$$z = \frac{r \cdot (x - center)}{spread} - r \cdot realMin + targetMin.$$

If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV` then *center* is set equal to the arithmetic average $\bar{x}$ of `x`, and *spread* is set equal to the sample standard deviation of `x`. If `scalingMethod` is `UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD` then *center* is set equal to the median $\tilde{m}$ of `x`, and *spread* is set equal to the Mean Absolute Difference (MAD).The method `decode` can be used to unfilter data using the the inverse calculation for the above equation:

$$x = \frac{spread \cdot (z - targetMin)}{r} + spread \cdot realMin + center$$

## Declaration

public class com.imsl.datamining.neural.ScaleFilter
**extends** java.lang.Object
**implements** java.io.Serializable

## Fields

- public static final `int` **NO_SCALING**
  - Flag to indicate no scaling.
- public static final `int` **BOUNDED_SCALING**
  - Flag to indicate bounded scaling.
- public static final `int` **UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV**
  - Flag to indicate unbounded z-score scaling using the mean and standard deviation.
- public static final `int` **UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD**
  - Flag to indicate unbounded z-score scaling using the median and mean absolute difference.
- public static final `int` **BOUNDED_Z_SCORE_SCALING_MEAN_STDEV**
  - Flag to indicate bounded z-score scaling using the mean and standard deviation.
- public static final `int` **BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD**
  - Flag to indicate bounded z-score scaling using the median and mean absolute difference.

## Constructor

- *ScaleFilter*
  `public` **ScaleFilter**`( int` **scalingMethod** `)`
  - **Description**
    Constructor for `ScaleFilter`.
  - **Parameters**
    * `scalingMethod` – An `int` specifying the scaling method to be applied. `scalingMethod` is specified by: NO_SCALING, BOUNDED_SCALING, UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV, UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD, BOUNDED_Z_SCORE_SCALING_MEAN_STDEV, or BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD.

## Methods

- *decode*
  `public double` **decode**`( double` **z** `)`
  - **Description**
    Unscales a value.
  - **Parameters**
    * **z** – A `double` containing the value to be unscaled.
  - **Returns** – A `double` containing the filtered data.

- *decode*
  `public double[]` **decode**`( double[]` **z** `)`
  - **Description**
    Unscales an array of values.
  - **Parameters**
    * **z** – A `double` array of values to be unscaled.
  - **Returns** – A `double` array containing the filtered data.

- *decode*
  `public void` **decode**`( int` **columnIndex**`, double[][]` **z** `)`
  - **Description**
    Unscales a single column of a two dimensional array of values.

- **Parameters**
  - `columnIndex` – An `int` specifying the index of the column of `z` to unscale. Indexing is zero-based.
  - `z` – A `double` matrix containing the values to be unscaled. Its `columnIndex`-th column is modified in place.

---

- *encode*
  `public double` **encode**`( double x )`
  - **Description**
    Scales a value.
  - **Parameters**
    - `x` – A `double` containing the value to be scaled.
  - **Returns** – A `double` containing the scaled value.

---

- *encode*
  `public double[]` **encode**`( double[] x )`
  - **Description**
    Scales an array of values.
  - **Parameters**
    - `x` – A `double` array containing the data to be scaled.
  - **Returns** – A `double` array containing the scaled data.

---

- *encode*
  `public void` **encode**`( int` **columnIndex**`, double[][] x )`
  - **Description**
    Scales a single column of a two dimensional array of values.
  - **Parameters**
    - `columnIndex` – An `int` specifying the index of the column of `x` to scale. Indexing is zero-based.
    - `x` – A `double` matrix containing the value to be scaled. Its `columnIndex`-th column is modified in place.

---

- *getBounds*
  `public double[]` **getBounds**`( )`
  - **Description**
    Retrieves bounds used during bounded scaling.

- **Returns** – A `double` array of length 4 containing the values

| i | result[i] |
|---|---|
| 0 | `realMin`. Lowest expected value in the data to be filtered. |
| 1 | `realMax`. Largest expected value in the data to be filtered. |
| 2 | `targetMin`. Lowest allowed value in the filtered data. |
| 3 | `targetMax`. Largest allowed value in the filtered data. |

---

- *getCenter*
  `public double` **getCenter**`( )`
  - **Description**
    Retrieves the measure of center to be used during z-score scaling.
  - **Returns** – A `double` containing the measure of center to be used during z-score scaling.

---

- *getSpread*
  `public double` **getSpread**`( )`
  - **Description**
    Retrieves the measure of spread to be used during scaling.
  - **Returns** – a `double` containing the measure of spread to be used during scaling.

---

- *setBounds*
  `public void` **setBounds**`( double` **realMin**`, double` **realMax**`, double` **targetMin**`, double` **targetMax** `)`
  - **Description**
    Sets bounds to be used during bounded scaling and unscaling. This method is normally called prior to calls to `encode` or `decode`. Otherwise the default bounds are `realMin` $= 0$, `realMax` $= 1$, `targetMin` $= 0$, and `targetMax` $= 1$. These bounds are ignored for unbounded scaling.
  - **Parameters**
    * `realMin` – A `double` containing the lowest expected value in the data to be filtered.
    * `realMax` – A `double` containing the largest expected value in the data to be filtered.
    * `targetMin` – A `double` containing the lowest allowed value in the filtered data.
    * `targetMax` – A `double` containing the largest allowed value in the filtered data.

---

- *setCenter*
  `public void` **setCenter**`( double` **center** `)`
  - **Description**
    Set the measure of center to be used during z-score scaling.
  - **Parameters**
    * `center` – A `double` containing the measure of center to be used during scaling. If this method is not called then the measure of center is computed from the data.

---

- *setSpread*
  `public void` **setSpread**`( double` **spread** `)`
  - **Description**
    Set the measure of spread to be used during z-score scaling.
  - **Parameters**
    * `spread` – A `double` containing the measure of spread to be used during z-score scaling. If this method is not called then the measure of spread is computed from the data.

### Example: ScaleFilter

In this example three sets of data, X0, X1, and X2 are scaled using the methods described in the following table:

Variables and Scaling Methods

| Variable | Method | Description |
|---|---|---|
| X0 | 0 | No Scaling |
| X1 | 4 | Bounded Z-score scaling using the mean and standard deviation of X1 |
| X2 | 5 | Bounded Z-score scaling using the median and MAD of X2 |

The bounds, measures of center and spread for **X1** and **X2** are:

Scaling Limits and Measures of Center and Spread

| Variable | Real Limits | Target Limits | Measure of Center | Measure of Spread |
|---|---|---|---|---|
| X1 | (-6, +6) | (-3, +3) | 3.4 (Mean) | 1.7421 (Std. Dev.) |
| X2 | (-3, +3) | (-3, +3) | 2.4 (Median) | 1.3343 (MAD/0.6745) |

The real and target limits are used for bounded scaling. The measures of center and spread are used to calculate z-scores. Using these values for **x1[0]=3.5** yields the following calculations:

For **x1[0]** , the scale factor is calculated using the real and target limits in the above table:

**r** = (3-(-3))/(6-(-6)) = 0.5

The z-score for **x1[0]** is calculated using the measures of center and spread:

**z1[0]** = (3.5 - 3.4)/1.7421 = 0.057402

Since method=4 is used for **x1**, this z-score is bounded (scaled) using the real and target limits:

**z1(bounded)** = **r**(**z1[0]**) - r(realMin) + (targetMin)
= 0.5(0.057402) - 0.5(-6) + (-3) = 0.029

The calculations for **x2[0]** are nearly identical, except that since method=5 for **x2**, the median and MAD replace the mean and standard deviation used to calculate **z1(bounded):**

**r** = (3-(-3))/(3-(-3)) = 1,

**z2[0]** = (3.1 - 2.4)/1.3343 = 0.525, and

**z2(bounded)** = **r**(**z2[0]**) - r(realMin) + (targetMin)
= 1(0.525) - 1(-3) + (-3) = 0.525

```
import com.imsl.stat.*;
import com.imsl.math.*;
import com.imsl.datamining.neural.*;

public class ScaleFilterEx1 {
    public static void main(String args[]) throws Exception {
        ScaleFilter[] scaleFilter = new ScaleFilter[3];
        scaleFilter[0] = new ScaleFilter(ScaleFilter.NO_SCALING);
        scaleFilter[1] =
            new ScaleFilter(ScaleFilter.BOUNDED_Z_SCORE_SCALING_MEAN_STDEV);
        scaleFilter[1].setBounds(-6.0, 6.0, -3.0, 3.0);
        scaleFilter[2] =
```

```
    new ScaleFilter(ScaleFilter.BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD);
scaleFilter[2].setBounds(-3.0, 3.0, -3.0, 3.0);
int nObs = 5;
double[] y0, y1, y2;
double[] x0 = {1.2, 0.0, -1.4, 1.5, 3.2};
double[] x1 = {3.5, 2.4, 4.4, 5.6, 1.1};
double[] x2 = {3.1, 1.5, -1.5, 2.4, 4.2};

// Perform forward filtering
y0 = scaleFilter[0].encode(x0);
y1 = scaleFilter[1].encode(x1);
y2 = scaleFilter[2].encode(x2);
// Display x0
System.out.print("X0 = {");
for (int i=0; i<4; i++) System.out.print(x0[i]+", ");
System.out.println(x0[4]+"}");
// Display summary statistics for X1
System.out.print("\nX1 = {");
for (int i=0; i<4; i++) System.out.print(x1[i]+", ");
System.out.println(x1[4]+"}");
System.out.println("X1 Mean:       "+scaleFilter[1].getCenter());
System.out.println("X1 Std. Dev.:  "+scaleFilter[1].getSpread());
// Display summary statistics for X2
System.out.print("\nX2 = {");
for (int i=0; i<4; i++) System.out.print(x2[i]+", ");
System.out.println(x2[4]+"}");
System.out.println("X2 Median:     "+scaleFilter[2].getCenter());
System.out.println("X2 MAD/0.6745: "+scaleFilter[2].getSpread());
System.out.println("");
PrintMatrix pm = new PrintMatrix();
pm.setTitle("Filtered X0 Using Method=0 (no scaling)");
pm.print(y0);
pm.setTitle("Filtered X1 Using Bounded Z-score Scaling\n"+
            "with Center=Mean and Spread=Std. Dev.");
pm.print(y1);
pm.setTitle("Filtered X2 Using Bounded Z-score Scaling\n"+
            "with Center=Median and Spread=MAD/0.6745");
pm.print(y2);

// Perform inverse filtering
double[] z0, z1, z2;
z0 = scaleFilter[0].decode(y0);
```

```
        z1 = scaleFilter[1].decode(y1);
        z2 = scaleFilter[2].decode(y2);
        pm.setTitle("Decoded Z0");
        pm.print(z0);
        pm.setTitle("Decoded Z1");
        pm.print(z1);
        pm.setTitle("Decoded Z2");
        pm.print(z2);

    }
}
```

## Output

```
X0 = {1.2, 0.0, -1.4, 1.5, 3.2}

X1 = {3.5, 2.4, 4.4, 5.6, 1.1}
X1 Mean:       3.4
X1 Std. Dev.:  1.7421251390184345

X2 = {3.1, 1.5, -1.5, 2.4, 4.2}
X2 Median:     2.4
X2 MAD/0.6745: 1.3343419966550414

Filtered X0 Using Method=0 (no scaling)
    0
0   1.2
1   0
2  -1.4
3   1.5
4   3.2

Filtered X1 Using Bounded Z-score Scaling
with Center=Mean and Spread=Std. Dev.
     0
0   0.029
1  -0.287
2   0.287
3   0.631
4  -0.66
```

```
Filtered X2 Using Bounded Z-score Scaling
with Center=Median and Spread=MAD/0.6745
      0
0   0.525
1  -0.674
2  -2.923
3   0
4   1.349

Decoded Z0
    0
0   1.2
1   0
2  -1.4
3   1.5
4   3.2

Decoded Z1
    0
0  3.5
1  2.4
2  4.4
3  5.6
4  1.1

Decoded Z2
    0
0   3.1
1   1.5
2  -1.5
3   2.4
4   4.2
```

## *class* **UnsupervisedNominalFilter**

Converts nominal data into a series of binary encoded columns for input to a neural network. It also reverses the aforementioned encoding, accepting binary encoded data and returns an array of integers representing the classes for a nominal variable.

## Binary Encoding

Method `encode` can be used to apply binary encoding. Referring to the result as `z`, binary encoding takes each category in the nominal variable `x[]`, and creates a column in `z` containing all zeros and ones. A value of zero indicates that this category was not present and a value of one indicates that it is present.

For example, if `x[]={2, 1, 3, 4, 2, 4}` then `nClasses=4`, and

$$z = \begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Notice that the number of columns in the result, `z`, is equal to the number of distinct classes in `x`. The number of rows in `z` is equal to the length of `x`.

## Binary Decoding

Unfiltering can be performed using the method `decode`. In this case, `z` is the input, and we refer to $x$ as the output. Binary unfiltering takes binary representation in `z`, and returns the appropriate class in $x$.

For example, if a row in `z` equals {0, 1, 0, 0}, then the return value from `decode` would be 2 for that row. If a row in `z` equals {1, 0, 0, 0}, then the return value from `decode` would be 1 for that row. Notice these are the same values as the first two elements of the original `x[]` because classes are numbered sequentially from 1 to `nClasses`. This ensures that the results of `decode` are associated with the $i$th class in `x[]`.

## Declaration

public class com.imsl.datamining.neural.UnsupervisedNominalFilter
**extends** java.lang.Object
**implements** java.io.Serializable

## Constructor

---

- *UnsupervisedNominalFilter*
  `public` **UnsupervisedNominalFilter**`( int` **nClasses** `)`

– **Description**
  Constructor for `UnsupervisedNominalFilter`.
– **Parameters**
  * `nClasses` – An `int` specifying the number of categories in the nominal variable to be filtered.

## Methods

---

- *decode*
  `public int decode( int[] z )`

  – **Description**
    Decodes a binary encoded array into its nominal category. This is the inverse of the `encode(int)` method.
  – **Parameters**
    * `z` – An `int` array containing the data to be decoded.
  – **Returns** – An `int` containing the number associated with the category encoded in `z`.

---

- *decode*
  `public int[] decode( int[][] z )`

  – **Description**
    Decodes a matrix representing the binary encoded columns of the nominal variable. This is the inverse of the `encode(int[])` method.
  – **Parameters**
    * `z` – An `int` matrix containing the data to be decoded.
  – **Returns** – An `int` array containing the decoded data.

---

- *encode*
  `public int[] encode( int x )`

  – **Description**
    Apply forward encoding to a value.
  – **Parameters**
    * `x` – An `int` containing the value to be encoding. Class number must be in the range 1 to `nClasses`.
  – **Returns** – An `int` array containing the encoded data.

---

- *encode*
  `public int[][] encode( int[] x )`

– **Description**
  Encodes class data prior to its use in neural network training.
– **Parameters**
  * `x` – An `int` array containing the data to be encoded. Class number must be in the range 1 to `nClasses`.
– **Returns** – An `int` matrix containing the encoded data.

---

- *getNumberOfClasses*
  `public int` **getNumberOfClasses**`( )`
  – **Description**
    Retrieves the number of classes in the nominal variable.
  – **Returns** – An `int` containing the number of classes in the nominal variable.

### Example: UnsupervisedNominalFilter

In this example a data set with 7 observations and 3 classes is filtered.

```
import com.imsl.stat.*;
import com.imsl.math.*;
import com.imsl.datamining.neural.*;

public class UnsupervisedNominalFilterEx1 {
    public static void main(String args[]) throws Exception {
        int nClasses = 3;
        UnsupervisedNominalFilter filter = new UnsupervisedNominalFilter(nClasses);
        int nObs = 7;
        int[] x = {3, 3, 1, 2, 2, 1, 2};
        int[] xBack = new int[nObs];
        int[][] z;

        /* Perform Binary Filtering. */
        z = filter.encode(x);
        PrintMatrix pm = new PrintMatrix();
        pm.setTitle("Filtered x");
        pm.print(z);

        /* Perform Binary Un-filtering. */
        for (int i=0;i<nObs;i++) {
            xBack[i] = filter.decode(z[i]);
        }
```

```
        pm.setTitle("Result of inverse filtering");
        pm.print(xBack);
    }
}
```

### Output

```
 Filtered x
   0  1  2
0  0  0  1
1  0  0  1
2  1  0  0
3  0  1  0
4  0  1  0
5  1  0  0
6  0  1  0

Result of inverse filtering
   0
0  3
1  3
2  1
3  2
4  2
5  1
6  2
```

## *class* UnsupervisedOrdinalFilter

Encodes ordinal data into percentages for input to a neural network. It also allows decoding, accepting a percentage and converting it into an ordinal value.

Class `UnsupervisedOrdinalFilter` is designed to either encode or decode ordinal variables. Encoding consists of transforming the ordinal classes into percentages, with each percentage being equal to the percentage of the data at or below this class.

## Ordinal Encoding

In this case, `x` is input to the method `encode` and is filtered by converting each ordinal class value into a cumulative percentage.

For example, if `x[]={2, 1, 3, 4, 2, 4, 1, 1, 3, 3}` then `nClasses=4`, and `encode` returns the ordinal class designation with the cumulative percentages displayed in the following table. Cumulative percentages are equal to the percent of the data in this class or a lower class.

| Ordinal Class | Frequency | Cumulative Percentage |
|---|---|---|
| `1` | 3 | 30% |
| `2` | 2 | 50% |
| `3` | 3 | 80% |
| `4` | 2 | 100% |

Classes in `x` must be numbered from 1 to `nClasses`.

The values returned from encoding or decoding depend upon the setting of `transform`. In this example, if the filter was constructed with `transform = TRANSFORM_NONE`, then the method `encode` will return

$$z[] = \{50, 30, 80, 100, 50, 100, 30, 30, 80, 80\}.$$

If the filter was constructed with `transform = TRANSFORM_SQRT`, then the square root of these values is returned, i.e.,

$$z[i] = \sqrt{\frac{z[i]}{100}}$$

$$z[] = \{0.71, 0.55, 0.89, 1.0, 0.71, 1.0, 0.55, 0.55, 0.89, 0.89\};$$

If the filter was constructed with `transform = TRANSFORM_ASIN_SQRT`, then the arcsin square root of these values is returned using the following calculation:

$$z[i] = \arcsin\left(\sqrt{\frac{z[i]}{100}}\right)$$

## Ordinal Decoding

Ordinal decoding takes a transformed cumulative proportion and converts it into an ordinal class value.

## Declaration

```
public class com.imsl.datamining.neural.UnsupervisedOrdinalFilter
extends java.lang.Object
implements java.io.Serializable
```

## Fields

- public static final `int` **TRANSFORM_NONE**
  - Flag to indicate no transformation of percentages.
- public static final `int` **TRANSFORM_SQRT**
  - Flag to indicate the square root transform will be applied to the percentages.
- public static final `int` **TRANSFORM_ASIN_SQRT**
  - Flag to indicate the arcsine square root transform will be applied to the percentages.

## Constructor

- *UnsupervisedOrdinalFilter*
  `public` **UnsupervisedOrdinalFilter**`( int` **nClasses**`, int` **transform** `)`
  - **Description**
    Constructor for `UnsupervisedOrdinalFilter`.
  - **Parameters**
    * `nClasses` – An `int` specifying the number of classes in the data to be filtered.
    * `transform` – An `int` specifying the transform to be applied to the percentages. Values for `transform` are: TRANSFORM_NONE, TRANSFORM_SQRT, TRANSFORM_ASIN_SQRT,

## Methods

- *decode*
  `public int` **decode**`( double` **y** `)`

- **Description**
  Decodes an encoded ordinal variable.
- **Parameters**
  * `y` – A `double` containing the encoded value to be decoded.
- **Returns** – An `int` containing the ordinal category associated with `y`.

---

- *decode*
  `public int[] decode( double[] y )`
  - **Description**
    Decodes an array of encoded ordinal values.
  - **Parameters**
    * `y` – A `double` array containing the encoded ordinal data to be decoded.
  - **Returns** – An `int` array containing the decoded ordinal classifications.

---

- *encode*
  `public double encode( int x )`
  - **Description**
    Encodes an ordinal category.
  - **Parameters**
    * `x` – An `int` containing the ordinal category. Must be an integer between 1 and `nClasses`.
  - **Returns** – A `double` containing the encoded value, a transformed cumulative percentage.

---

- *encode*
  `public double[] encode( int[] x )`
  - **Description**
    Encodes an array of ordinal categories into an array of transformed percentages.
  - **Parameters**
    * `x` – An `int` array containing the categories for the ordinal variable. Categories must be numbered from 1 to `nClasses`.
  - **Returns** – A `double` array of the transformed percentages.

---

- *getNumberOfClasses*
  `public int getNumberOfClasses( )`
  - **Description**
    Retrieves the number of categories associated with this ordinal variable.
  - **Returns** – An `int` containing the number of categories associated with this ordinal variable.

- *getPercentages*
  `public double[]` **getPercentages**`( )`
  - **Description**
    Retrieves the cumulative percentages used for encoding and decoding. If a transform has been applied to the percentages then the transformed percentages are returned.
  - **Returns** – A `double` array of length `nClasses` containing the cumulative transformed percentages associated with the ordinal categories.

- *getTransform*
  `public int` **getTransform**`( )`
  - **Description**
    Retrieves the transform flag used for encoding and decoding.
  - **Returns** – An `int` containing the transform flag used for encoding and decoding.

- *setPercentages*
  `public void` **setPercentages**`( double[]` **percentages** `)`
  - **Description**
    Set the untransformed cumulative percentages used during encoding and decoding. Setting percentages with this method bypasses calculating cumulative percentages based on the data being encoded. The percentages must be nondecreasing in the interval [0, 100], with the last element equal to 100. If this method is used it must be called prior to any calls to the encoding and decoding methods.
  - **Parameters**
    * `percentages` – A `double` array of length `nClasses` containing the cumulative percentages to use during encoding and decoding.

### Example: UnsupervisedOrdinalFilter

In this example a data set with 10 observations and 4 classes is filtered.

```
import com.imsl.stat.*;
import com.imsl.math.*;
import com.imsl.datamining.neural.*;

public class UnsupervisedOrdinalFilterEx1 {
```

```
    public static void main(String args[]) throws Exception {
        int nClasses = 4;
        UnsupervisedOrdinalFilter filter =
        new UnsupervisedOrdinalFilter(nClasses,
        UnsupervisedOrdinalFilter.TRANSFORM_ASIN_SQRT);
        int[] x = {2,1,3,4,2,4,1,1,3,3};
        int nObs = x.length;
        int[] xBack;
        double[] z;
        /* Ordinal Filtering. */
        z = filter.encode(x);
        // Print result without row/column labels.
        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat mf;
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        pm.setTitle("Filtered data");
        pm.print(mf, z);

        /* Ordinal Un-filtering. */
        pm.setTitle("Un-filtered data");
        xBack = filter.decode(z);

        // Print results of Un-filtering.
        pm.print(mf, xBack);
    }
}
```

### Output

```
Filtered data

0.785
0.58
1.107
1.571
0.785
1.571
0.58
0.58
```

```
1.107
1.107

Un-filtered data

2
1
3
4
2
4
1
1
3
3
```

## *class* **TimeSeriesFilter**

Converts time series data to a lagged format used as input to a neural network.

Class `TimeSeriesFilter` can be used to operate on a data matrix and lags every column to form a new data matrix. Using the method `computeLags`, each column of the input matrix, `x`, is transformed into (`nLags`+1) columns by creating a column for $lags = 0, 1, \ldots nLags$.

The output data array, $z$, can be symbolically represented as:

$$z = |x(0) : x(1) : x(2) : \ldots : x(nLags - 1)|,$$

where `x(i)` is a lagged column of the incoming data matrix, `x`.

Consider, an example in which `x` has five rows and two columns with all variables continuous input attributes. Using $nObs$ and $nVar$ to represent the number of rows and columns in `x`, let

$$x = \begin{bmatrix} 1 & 6 \\ 2 & 7 \\ 3 & 8 \\ 4 & 9 \\ 5 & 10 \end{bmatrix}$$

If `nLags=1`, then the number of columns in `z[][]` is $nVar$*($nLags$+1)=2*2=4, and the number of rows is (*nObs*-*nLags*)=5-1=4:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 \\ 2 & 7 & 3 & 8 \\ 3 & 8 & 4 & 9 \\ 4 & 9 & 5 & 10 \end{bmatrix}$$

If `nLags=2`, then the number of rows in $z$ will be (*nObs*-*nLags*)=(5-2)=3 and the number of columns will be *nVar**(*nLags*+1)=2*3=6:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 8 \\ 2 & 7 & 3 & 8 & 4 & 9 \\ 3 & 8 & 4 & 9 & 5 & 10 \end{bmatrix}$$

## Declaration

public class com.imsl.datamining.neural.TimeSeriesFilter
**extends** java.lang.Object
**implements** java.io.Serializable

## Constructor

- *TimeSeriesFilter*
  `public` **TimeSeriesFilter**`( )`
  - **Description**
    Constructor for `TimeSeriesClassFilter`.

## Method

- *computeLags*
  `public double[][]` **computeLags**`( int` **nLags**`, double[][]` **x** `)`
  - **Description**
    Lags time series data to a format used for input to a neural network.
  - **Parameters**
    * `nLags` – An `int` containing the requested number of lags. `nLags` must be greater than 0.

* `x` – A `double` matrix, *nObs* by *nVar*, containing the time series data to be lagged. It is assumed that `x` is sorted in descending chronological order.

– **Returns** – A `double` matrix with (*nObs*-*nLags*) rows and (*nVar*(*nLags+1*)) columns. The columns 0 through (*nVar*-1) contain the columns of `x`. The next *nVar* columns contain the first lag of the columns in `x`, etc.

### Example: TimeSeriesFilter

In this example a matrix with 5 rows and 2 columns is lagged twice. This produces a two-dimensional matrix with 5 rows, but 2*3=6 columns. The first two columns correspond to lag=0, which just places the original data into these columns. The 3rd and 4th columns contain the first lags of the original 2 columns and the 5th and 6th columns contain the second lags.

```
import com.imsl.stat.*;
import com.imsl.math.*;
import com.imsl.datamining.neural.*;

public class TimeSeriesFilterEx1 {
    public static void main(String args[]) throws Exception {
        TimeSeriesFilter filter = new TimeSeriesFilter();
        int nLag = 2;
        double[][] x = {
            {1, 6},
            {2, 7},
            {3, 8},
            {4, 9},
            {5, 10}
        };
        double[][] z = filter.computeLags(nLag, x);
         // Print result without row/column labels.
        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat mf;
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setNoColumnLabels();
        pm.setTitle("Lagged data");
        pm.print(mf, z);
    }
}
```

## Output

```
     Lagged data

1   6   2   7   3    8
2   7   3   8   4    9
3   8   4   9   5   10
```

# *class* TimeSeriesClassFilter

Converts time series data contained within nominal categories to a lagged format for processing by a neural network. Lagging is done within the nominal categories associated with the time series.

Class `TimeSeriesClassFilter` can be used with a data array, `x[]` to compute a new data array, `z[][]`, containing lagged columns of `x[]`.

When using the method `computeLags`, the output array, `z[][]` of lagged columns, can be symbolically represented as:

$$z = |x(0) : x(1) : x(2) : \ldots : x(nLags - 1)|,$$

where *x(i)* is a lagged column of the incoming data array `x`, and *nLags* is the number of computed lags. The lag associated with *x(i)* is equal to the value in `lag[i]`, and lagging is done within the nominal categories given in `iClass[]`. This requires the time series data in `x[]` be sorted in time order within each category `iClass`.

Consider an example in which the number of observations in `x[]` is 10. There are two lags requested in `lag[]`. If

$$x^T = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\},$$

$$iClass^T = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1\},$$

and

$$lag^T = \{0, 2\}$$

then, all the time series data fall into a single category, i.e. `nClasses` $= 1$, and $z$ would contain 2 columns and 10 rows. The first column reproduces the values in `x[]` because `lags[0]`=0, and the second column is the 2nd lag because `lags[0]`=2.

$$z = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & 6 \\ 5 & 7 \\ 6 & 8 \\ 7 & 9 \\ 8 & 10 \\ 9 & NaN \\ 10 & NaN \end{bmatrix}$$

On the other hand, if the data were organized into two classes with

$$iClass^T = \{1, 1, 1, 1, 1, 2, 2, 2, 2, 2\},$$

then `nClasses` is 2, and $z$ is still a 2 by 10 matrix, but with the following values:

$$z = \left[\begin{array}{cc} 1 & 3 \\ 2 & 4 \\ 3 & 5 \\ 4 & NaN \\ 5 & NaN \\ \hline 6 & 8 \\ 7 & 9 \\ 8 & 10 \\ 9 & NaN \\ 10 & NaN \end{array}\right]$$

The first 5 rows of $z$ are the lagged columns for the first category, and the last five are the lagged columns for the second category.

## Declaration

public class com.imsl.datamining.neural.TimeSeriesClassFilter
**extends** java.lang.Object
**implements** java.io.Serializable

## Constructor

- *TimeSeriesClassFilter*
  `public` **TimeSeriesClassFilter**`( int` **nClasses** `)`

- **Description**
  Constructor for `TimeSeriesClassFilter`.
- **Parameters**
  * `nClasses` – An `int` specifying the number of nominal categories associated with the time series.

### Method

---

- *computeLags*
  `public double[][]` **computeLags**`( int[]` **lags**`, int[]` **iClass**`, double[] x )`

  - **Description**
    Computes lags of an array sorted first by class designations and then descending chronological order.
  - **Parameters**
    * `lags` – An `int` array containing the requested lags. Every lag must be non-negative.
    * `iClass` – An `int` array containing class number associated with each element of `x`, sorted in ascending order. The $i$th element is equal to the class associated with the $i$th element of `x`. `iClass` and `x` must be the same length.
    * `x` – A `double` array containing the time series data to be lagged. This array is assumed to be sorted first by class designations and then descending chronological order, i.e., most recent observations appear first within a class.
  - **Returns** – A `double` matrix containing the lagged data. The $i$-th column of this array is the lagged values of `x` for a lag equal to `lags[i]`. The number of rows is equal to the length of `x`.

### Example: TimeSeriesClassFilter

For illustration purposes, the time series in this example consists of the integers 1, 2, ..., 10, organized into two classes. Of course, it is assumed that these data are sorted in chronologically descending order. That is for each class, the first number is the latest value and the last number in that class is the earliest.

The values 1-4 are in class 1, and the values 5-10 are in class 2. These values represent two separate time series, one for each class. If you were to list them in chronologically ascending order, starting with time = T0, the values would be:

Class 1: T0=4, T1=3, T2=2, T3=1
Class 2: T0=10, T1=9, T2=8, T3=7, T4=6, T5=5

This example requests lag calculations for lags 0, 1, 2, 3. For lag=0, no lagging is performed. For lag=1, the value at time = t replaced with the value at time = t-1, the previous value in that class. If $t-1<0$, then a missing value is placed in that position.

For example, the first lag of a time series at time=t are the values at time=t-1. For the time series values of Class 1 (lag=1), these values are:

Class 1, lag 1: T0=NaN, T1=4, T2=3, T3=2

The second lag for time=t consists of the values at time=t-2:

Class 1, lag 2: T0=NaN, T1=NaN, T2=4, T3=3

Notice that the second lag now has two missing observations. In general, lag=n will have n missing values. In some cases this can result in all missing values for classes with few observations. A class will have all missing values in any of its lag columns that have a lag value larger than or equal to the number of observations in that class.

```
import com.imsl.stat.*;
import com.imsl.math.*;
import com.imsl.datamining.neural.*;

public class TimeSeriesClassFilterEx1 {
    private static int nClasses = 2;
    private static int nObs     =10;
    private static int nLags    = 4;
    public static void main(String args[]) throws Exception {

        double[] x         = {1,2,3,4,5,6,7,8,9,10};
        double[] time      = {3,2,1,0,5,4,3,2,1,0};
        int[] iClass       = {1,1,1,1,2,2,2,2,2,2};
        int[] lag          = {0,1,2,3};
        String[] colLabels = {"Class","Time","Lag=0","Lag=1","Lag=2","Lag=3"};

        // Filter Classified Time Series Data
        TimeSeriesClassFilter filter = new TimeSeriesClassFilter(nClasses);
        double[][] y = filter.computeLags(lag, iClass, x);
        double[][] z = new double[nObs][nLags+2];
        for(int i=0; i < nObs;i++){
           z[i][0] = (double)iClass[i];
           z[i][1] = time[i];
```

```
            for(int j=0; j < nLags; j++){
                z[i][j+2] = y[i][j];
            }
        }

        // Print result without row/column labels.
        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat mf;
        mf = new PrintMatrixFormat();
        mf.setNoRowLabels();
        mf.setColumnLabels(colLabels);
        pm.setTitle("Lagged data");

        pm.print(mf, z);
    }
}
```

## Output

```
              Lagged data
Class  Time  Lag=0  Lag=1  Lag=2  Lag=3
1      3     1      2      3      4
1      2     2      3      4      ?
1      1     3      4      ?      ?
1      0     4      ?      ?      ?
2      5     5      6      7      8
2      4     6      7      8      9
2      3     7      8      9     10
2      2     8      9     10      ?
2      1     9     10      ?      ?
2      0    10      ?      ?      ?
```

# Chapter 26

# Miscellaneous

Classes



## *class* Messages

Retrieve and format message strings.

## Declaration

public class com.imsl.Messages
**extends** java.lang.Object

## Constructor

---

- *Messages*
  `public` **Messages**`( )`

## Methods

---

- *check*
  `public static int` **check**`( int` **arg** `)`

---

- *formatMessage*
  `public static java.lang.String` **formatMessage**`( java.lang.String` **bundleName**`, java.lang.String` **key** `)`
  - **Description**
    A message is formatted, without arguments, using a MessageFormat string retrieved from the named resource bundle using the given key.
  - **Parameters**
    * `bundleName` – is the resource bundle name.
    * `key` – is the key of the MessageFormat string in the resource bundle.

---

- *formatMessage*
  `public static java.lang.String` **formatMessage**`( java.lang.String` **bundleName**`, java.lang.String` **key**`, java.lang.Object[]` **arg** `)`
  - **Description**
    A message is formatted using a MessageFormat string retrieved from the named resource bundle using the given key.
  - **Parameters**
    * `bundleName` – is the resource bundle name.
    * `key` – is the key of the MessageFormat string in the resource bundle.
    * `arg` – is an array of arguments passed to the MessageFormat.format method.

---

- *throwIllegalArgumentException*
  `public static void` **throwIllegalArgumentException**`( java.lang.String` **packageName**`, java.lang.String` **key**`, java.lang.Object[]` **args** `)`
  - **Description**
    Throws an IllegalArgumentException with a formatted String argument.
  - **Parameters**
    * `packageName` – is package from which the error is thrown. The resource bundle "ErrorMessages" in this package contains the error MessageFormat string.
    * `key` – is the key of the MessageFormat string in the resource bundle.
    * `args` – is an array of arguments passed to the MessageFormat.format method.

# *class* Version

Print the version information.

## Declaration

public class com.imsl.Version
**extends** java.lang.Object

## Constructor

- *Version*
  `public` **Version**`( )`

## Method

- *main*
  `public static void` **main**`( java.lang.String[]` **args** `) throws java.text.ParseException`
  - **Description**
    Print the version information about the envirnoment and this library.

## *class* **Warning**

Handle warning messages. This class maintains a single, private, WarningObject that actually displays the warning messages.

### Declaration

public final class com.imsl.Warning
**extends** java.lang.Object

### Constructor

---

- *Warning*
  `public` **Warning**`( )`

### Methods

---

- *getWarning*
  `public static synchronized WarningObject` **getWarning**`( )`
  - **Description**
    Gets the WarningObject.
  - **Returns** – The current warning object.

---

- *print*
  `public static synchronized void` **print**`( java.lang.Object` **source**`, java.lang.String` **bundleName**`, java.lang.String` **key**`, java.lang.Object[]` **arg** `)`
  - **Description**
    Issue a warning message. Warning messages are stored as MessageFormat patterns in a ResourceBundle. This method retrieves the pattern from the bundle, formats the message with the supplied arguments, and prints the message to the warning stream.
  - **Parameters**
    * `source` – is the object that is the source of the warning.
    * `bundleName` – is the prefix of the ResourceBundle name. The actual name is formed by appending ".ErrorMessages".

* `key` – identifies the warning message in the bundle.
* `arg` – are the arguments used to format the message.

---

- *setOut*

  `public static synchronized void` **setOut**`( java.io.PrintStream` **out** `)`

  – **Description**

  Reassigns the output stream. The default warning stream is @see System.err.

  – **Parameters**

  * `out` – is the new warning output stream. It may be null, in which case warnings are not printed.

---

- *setWarning*

  `public static synchronized void` **setWarning**`( WarningObject` **warningObject** `)`

  – **Description**

  Sets a new WarningObject. Replacing the WarningObject allows warning errors to be handled in a more custom fashion.

  – **Parameters**

  * `warningObject` – is the new WarningObject. It may be null, in which case error messages will be ignored.

## *class* WarningObject

Handle warning messages.

### Declaration

public class com.imsl.WarningObject
**extends** java.lang.Object

### Field

---

- protected `java.io.PrintStream` **out**

  – The warning stream. Its default value is System.err.

### Constructor

---

- *WarningObject*
  `public` **WarningObject**`( )`

### Methods

---

- *print*
  `public synchronized void` **print**`( java.lang.Object` **source**`,`
  `java.lang.String` **bundleName**`, java.lang.String` **key**`,`
  `java.lang.Object[]` **arg** `)`
  - **Description**
    Issue a warning message. Warning messages are stored as MessageFormat patterns in a ResourceBundle. This method retrieves the pattern from the bundle, formats the message with the supplied arguments, and prints the message to the warning stream.
  - **Parameters**
    * `source` – is the object that is the source of the warning.
    * `bundleName` – is the prefix of the ResourceBundle name. The actual name is formed by appending ".ErrorMessages".
    * `key` – identifies the warning message in the bundle.
    * `arg` – are the arguments used to format the message.

---

- *setOut*
  `public synchronized void` **setOut**`( java.io.PrintStream` **out** `)`
  - **Description**
    Reassigns the output stream. The default warning stream is err.
  - **Parameters**
    * `out` – is the new warning output stream. It may be null, in which case warnings are not printed.

## *class* IMSLException

Signals that a mathematical exception has occurred.

## Declaration

public abstract class com.imsl.IMSLException
**extends** java.lang.Exception

## Constructors

---

- *IMSLException*
  `public` **IMSLException**`( )`
  - **Description**
    Constructs an IMSLException with no detail message. A detail message is a String that describes this particular exception.

---

- *IMSLException*
  `public` **IMSLException**`( java.lang.String s )`
  - **Description**
    Constructs an IMSLException with the specified detail message. A detail message is a String that describes this particular exception.
  - **Parameters**
    * `s` – the detail message

---

- *IMSLException*
  `public` **IMSLException**`( java.lang.String` **packageName**`, java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an IMSLException with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    * `packageName` – is the name of the package containing the ErrorMessages resource bundle.
    * `key` – is the key of the error message in the resource bundle.
    * `arguments` – is an array containing arguments used within the error message string.

# *class* **IMSLRuntimeException**

Signals that an error has occurred. This is used for programming mistake type of errors. Since IMSLRuntimeException is a subclass of RuntimeException, this exception does not have to be caught.

## Declaration

public abstract class com.imsl.IMSLRuntimeException
**extends** java.lang.RuntimeException

## Constructors

- *IMSLRuntimeException*
  `public` **IMSLRuntimeException**`( )`
  - **Description**
    Constructs an IMSLRuntimeException with no detail message. A detail message is a String that describes this particular exception.

- *IMSLRuntimeException*
  `public` **IMSLRuntimeException**`( java.lang.String s )`
  - **Description**
    Constructs an IMSLRuntimeException with the specified detail message. A detail message is a String that describes this particular exception.
  - **Parameters**
    * `s` – the detail message

- *IMSLRuntimeException*
  `public` **IMSLRuntimeException**`( java.lang.String` **packageName**`, java.lang.String` **key**`, java.lang.Object[]` **arguments** `)`
  - **Description**
    Constructs an IMSLRuntimeException with the specified detail message. The error message string is in a resource bundle, ErrorMessages.
  - **Parameters**
    * `packageName` – is the name of the package containing the ErrorMessages resource bundle.
    * `key` – is the key of the error message in the resource bundle.

∗ `arguments` – is an array containing arguments used within the error message string.

## *class* **LicenseManagerException**

A LicenseManagerException exception is thrown if a license to use the product cannot be obtained. Either a LicenseManagerException exception will be thrown or a `ExceptionInInitializerError` exception will be thrown with `LicenseManagerException` as the cause.

The behavior of the license manager is controlled by the following system properties.

| Property | Value | Meaning |
|---|---|---|
| com.imsl.license.path | License file path | A location in your installation hierarchy which indicates the expected license file location. This is a combination of one or more license file paths and `[port]@host` specifications. Multiple components of the list are separated by a semicolon (;) on Windows or colon (:) on UNIX. Redundant servers are not supported in Java. Default is `license.dat:@localhost` (Windows) or `license.dat:@localhost` (Unix). |
| com.imsl.license.queue | `‘‘true’’` or `‘‘false’’` | If `‘‘true’’`, automatically wait in the queue for a license without asking. Default is to ask the user. |
| com.imsl.license.popup | `‘‘true’’` or `‘‘false’’` | If `‘‘true’’`, use a dialog box to show any license manager errors or to ask the user about waiting for a license. If `‘‘false’’`, errors only result in this exception being thrown. The user is asked on the console about waiting for a license. Default is to use a popup. |

## Declaration

public class com.imsl.LicenseManagerException
**extends** com.imsl.IMSLRuntimeException (page 1242)

## Methods

- *getErrorNumber*
  `public int` **getErrorNumber**`( )`
  - **Description**
    Returns the FlexLM error number for this exception.

---

- *getFeature*
  `public java.lang.String` **getFeature**`( )`
  - **Description**
    Returns the name of the feature that could not be licensed.

---

- *getLicensePath*
  `public java.lang.String` **getLicensePath**`( )`
  - **Description**
    Returns the license file path for this exception.

---

- *getLocalizedMessage*
  `public java.lang.String` **getLocalizedMessage**`( )`
  - **Description**
    Returns the localized error message for this exception.

# Chapter 27

# References


**Abe**

Abe, S. (2001) *Pattern Classification: Neuro-Fuzzy Methods and their Comparison*, Springer-Verlag.

**Abramowitz and Stegun**

Abramowitz, Milton, and Irene A. Stegun (editors) (1964), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, National Bureau of Standards, Washington.

**Ahrens and Dieter**

Ahrens, J.H., and U. Dieter (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing*, **12**, 223-246.

**Akima**

Akima, H. (1970), A new method of interpolation and smooth curve fitting based on local procedures, *Journal of the ACM*, **17**, 589-602.

Akima, H. (1978), A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points, *ACM Transactions on Mathematical Software*, **4**, 148-159.

**Anderberg**

Anderberg, Michael R. (1973), *Cluster Analysis for Applications*, Academic Press, New York.

**Ashcraft**

Ashcraft, C. (1987), *A vector implementation of the multifrontal method for large sparse symmetric positive definite systems*, Technical Report ETA-TR-51, Engineering

Technology Applications Division, Boeing Computer Services, Seattle, Washington.

**Ashcraft et al.**

Ashcraft, C., R. Grimes, J. Lewis, B. Peyton, and H. Simon (1987), Progress in sparse matrix methods for large linear systems on vector supercomputers. *Intern. J. Supercomputer Applic.* , **1(4)**, 10-29.

**Atkinson (1979)**

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141-145.

**Atkinson (1978)**

Atkinson, Ken (1978), *An Introduction to Numerical Analysis*, John Wiley & Sons, New York.

**Barnett**

Barnett, A.R. (1981), An algorithm for regular and irregular Coulomb and Bessel functions of real order to machine accuracy, *Computer Physics Communication*, **21**, 297-314.

**Barrett and Heal**

Barrett, J.C., and M. J.R. Healy (1978), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **27**, 379-380.

**Bays and Durham**

Bays, Carter, and S.D. Durham (1976), Improving a poor random number generator, *ACM Transactions on Mathematical Software*, **2**, 59-64.

**Berry and Linoff**

Berry, M. J. A. and Linoff, G. (1997) *Data Mining Techniques*, John Wiley & Sons, Inc.

**Bishop**

Bishop, C. M. (1995) *Neural Networks for Pattern Recognition*, Oxford University Press.

**Blom**

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

**Boisvert**

Boisvert, Ronald (1984), A fourth order accurate fast direct method of the Helmholtz equation, *Elliptic Problem solvers II*, (edited by G. Birkhoff and A. Schoenstadt), Academic Press, Orlando, Florida, 35-44.

**Bosten and Battiste**

Bosten, Nancy E., and E.L. Battiste (1974), Incomplete beta ratio, *Communications of the ACM*, **17**, 156-157.

**Box and Jenkins**

Box, G. E. P. and Jenkins, G. M. (1970) *Time Series Analysis: Forecasting and Control*, Holden-Day, Inc.

**Breiman et al.**

Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984) *Classification and Regression Trees*, Chapman & Hall.

**Brent**

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

**Bridle**

Bridle, J. S. (1990) *Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition*, in F. Fogelman Soulie and J. Herault (Eds.), *Neuralcomputing: Algorithms, Architectures and Applications*, Springer-Verlag, 227-236.

**Brighamv**

Brigham, E. Oran (1974), *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey.

**Burgoyne**

Burgoyne, F.D. (1963), Approximations to Kelvin functions, *Mathematics of Computation*, **83**, 295-298.

**Calvo**

Calvo, R. A. (2001) *Classifying Financial News with Neural Networks*, Proceedings of the 6th Australasian Document Computing Symposium.

**Carlson**

Carlson, B.C. (1979), Computing elliptic integrals by duplication, *Numerische Mathematik*, **33**, 1-16.

**Carlson and Notis**

Carlson, B.C., and E.M. Notis (1981), Algorithms for incomplete elliptic integrals, *ACM Transactions on Mathematical Software*, **7**, 398-403.

**Carlson and Foley**

Carlson, R.E., and T.A. Foley (1991),The parameter $R^2$ in multiquadric interpolation, *Computer Mathematical Applications*, **21**, 29-42.

**Cheng**

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317-322.

**Clarkson and Jenrich**

Clarkson, Douglas B. and Robert B Jenrich (1991), Computing extended maximum likelihood estimates for linear parameter models, submitted to *Journal of the Royal Statistical Society*, Series B, **53**, 417-426.

**Cohen and Taylor**

Cohen, E. Richard, and Barry N. Taylor (1986), *The 1986 Adjustment of the Fundamental Physical Constants*, Codata Bulletin, Pergamon Press, New York.

**Cooley and Tukey**

Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297-301.

**Cooper**

Cooper, B.E. (1968), Algorithm AS4, An auxiliary function for distribution integrals, *Applied Statistics*, **17**, 190-192.

**Courant and Hilbert**

**Cook and Weisberg**

Cook, R. Dennis and Sanford Weisberg (1982), *Residuals and Influence in Regression*, Chapman and Hall, New York.

Courant, R., and D. Hilbert (1962), Methods of Mathematical Physics, Volume II, John Wiley & Sons, New York, NY.

**Craven and Wahba**

Craven, Peter, and Grace Wahba (1979), Smoothing noisy data with spline functions, *Numerische Mathematik*, **31**, 377-403.

**Crowe et al.**

Crowe, Keith, Yuan-An Fan, Jing Li, Dale Neaderhouser, and Phil Smith (1990), *A direct sparse linear equation solver using linked list storage*, IMSL Technical Report 9006, IMSL, Houston.

**Davis and Rabinowitz**

Davis, Philip F., and Philip Rabinowitz (1984), *Methods of Numerical Integration*, Academic Press, Orlando, Florida.

**de Boor**

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, New York.

**Dennis and Schnabel**

Dennis, J.E., Jr., and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, New Jersey.

**Dongarra et al.**

Dongarra, J.J., J.R. Bunch, C.B. Moler, and G.W. Stewart (1979), *LINPACK User's Guide*, SIAM, Philadelphia.

**Draper and Smith**

Draper, N.R., and H. Smith (1981), *Applied Regression Analysis*, 2nd. ed., John Wiley & Sons, New York.

**DuCroz et al.**

Du Croz, Jeremy, P. Mayes, and G. Radicati (1990), Factorization of band matrices using Level-3 BLAS, *Proceedings of CONPAR 90-VAPP IV*, Lecture Notes in Computer Science, Springer, Berlin, 222.

**Duff et al.**

Duff, I. S., A. M. Erisman, and J. K. Reid (1986), *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

**Duff and Reid**

Duff, I.S., and J.K. Reid (1983), The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, **9**, 302-325.

Duff, I.S., and J.K. Reid (1984), The multifrontal solution of unsymmetric sets of linear equations. *SIAM Journal on Scientific and Statistical Computing*, **5**, 633-641.

**Elman**

Elman, J. L. (1990) *Finding Structure in Time, Cognitive Science*, **14**, 179-211.

**Enright and Pryce**

Enright, W.H., and J.D. Pryce (1987), Two FORTRAN packages for assessing initial value methods, *ACM Transactions on Mathematical Software*, **13**, 1-22.

**Farebrother and Berry**

Farebrother, R.W., and G. Berry (1974), A remark on Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **23**, 477.

**Fisher**

Fisher, R.A. (1936), The use of multiple measurements in taxonomic problems, *Annals of Eugenics*, **7**, 179-188.

**Fishman and Moore**

Fishman, George S. and Louis R. Moore (1982), A statistical evaluation of multiplicative congruential random number generators with modulus 231 - 1, *Journal of the American Statistical Association*, **77**, 129-136.

**Forsythe**

Forsythe, G.E. (1957), Generation and use of orthogonal polynomials for fitting data with a digital computer, *SIAM Journal on Applied Mathematics*, **5**, 74-88.

**Franke**

Franke, R. (1982), Scattered data interpolation: Tests of some methods, *Mathematics of Computation*, **38**, 181-200.

**Furnival and Wilson**

Furnival, G.M. and R.W. Wilson, Jr. (1974), Regressions by leaps and bounds, *Technometrics*, **16**, 499-511.

**Garbow et al.**

Garbow, B.S., J.M. Boyle, K.J. Dongarra, and C.B. Moler (1977), *Matrix Eigensystem Routines - EISPACK Guide Extension*, Springer-Verlag, New York.

Garbow, B.S., G. Giunta, J.N. Lyness, and A. Murli (1988), Software for an implementation of Weeks' method for the inverse Laplace transform problem, *ACM Transactions on Mathematical Software*, **14**, 163-170.

**Gautschi**

Gautschi, Walter (1968), Construction of Gauss-Christoffel quadrature formulas, *Mathematics of Computation*, **22**, 251-270.

**Gear**

Gear, C.W. (1971), Numerical Initial Value Problems in Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, New Jersey.

**Gentleman**

Gentleman, W. Morven (1974), Basic procedures for large, sparse or weighted linear least squares problems, *Applied Statistics*, **23**, 448-454.

**George and Liu**

George, A., and J.W.H. Liu (1981), *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, Englewood Cliffs, New Jersey.

**Gill and Murray**

Gill, Philip E., and Walter Murray (1976), *Minimization subject to bounds on the variables*, NPL Report NAC 92, National Physical Laboratory, England.

**Gill et al.**

Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright (1985), Model building and practical aspects of nonlinear programming, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

**Giudici**

Giudici, P. (2003) *Applied Data Mining: Statistical Methods for Business and Industry*, John Wiley & Sons, Inc.

**Goldfarb and Idnani**

Goldfarb, D., and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs, *Mathematical Programming*, **27**, 1-33.

**Golub**

Golub, G.H. (1973), Some modified matrix eigenvalue problems, *SIAM Review*, **15**, 318-334.

**Golub and Van Loan**

Golub, G.H., and C.F. Van Loan (1989), *Matrix Computations*, Second Edition, The Johns Hopkins University Press, Baltimore, Maryland.

Golub, Gene H., and Charles F. Van Loan (1983), *Matrix Computations*, Johns Hopkins University Press, Baltimore, Maryland.

**Golub and Welsch**

Golub, G.H., and J.H. Welsch (1969), Calculation of Gaussian quadrature rules, *Mathematics of Computation*, **23**, 221-230.

**Gregory and Karney**

Gregory, Robert, and David Karney (1969), *A Collection of Matrices for Testing Computational Algorithms*, Wiley-Interscience, John Wiley & Sons, New York.

**Griffin and Redfish**

Griffin, R., and K A. Redish (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 54.

**Grosse**

Grosse, Eric (1980), Tensor spline approximation, *Linear Algebra and its Applications*, **34**, 29-41.

**Guerra and Tapia**

Guerra, V., and R. A. Tapia (1974), *A local procedure for error detection and data smoothing*, MRC Technical Summary Report 1452, Mathematics Research Center, University of Wisconsin, Madison.

**Hageman and Young**

Hageman, Louis A., and David M. Young (1981), *Applied Iterative Methods*, Academic Press, New York.

**Hanson**

Hanson, Richard J. (1986), Least squares with bounds and linear constraints, *SIAM Journal Sci. Stat. Computing*, 7, #3.

**Hardy**

Hardy, R.L. (1971), Multiquadric equations of topography and other irregular surfaces, *Journal of Geophysical Research*, **76**, 1905-1915.

**Hart et al.**

**Harman**

Harman, Harry H. (1976), *Modern Factor Analysis*, 3d ed. revised, University of Chicago Press, Chicago.

Hart, John F., E.W. Cheney, Charles L. Lawson, Hans J.Maehly, Charles K. Mesztenyi, John R. Rice, Henry G. Thacher, Jr., and Christoph Witzgall (1968), *Computer Approximations*, John Wiley & Sons, New York.

**Healy**

Healy, M.J.R. (1968), Algorithm AS 6: Triangular decomposition of a symmetric matrix, *Applied Statistics*, **17**, 195-197.

**Hebb**

Hebb, D. O. (1949) *The Organization of Behaviour: A Neuropsychological Theory*, John Wiley.

**Herraman**

Herraman, C. (1968), Sums of squares and products matrix, *Applied Statistics*, **17**, 289-292.

**Higham**

Higham, Nicholas J. (1988), FORTRAN Codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation, *ACM Transactions on Mathematical Software*, **14**, 381-396.

**Hill**

Hill, G.W. (1970), Student's $t$-distribution, *Communications of the ACM*, **13**, 617-619.

**Hindmarsh**

Hindmarsh, A.C. (1974), *GEAR:* Ordinary Differential Equation System Solver, Lawrence Livermore National Laboratory Report UCID-30001, Revision 3, Lawrence Livermore National Laboratory, Livermore, Calif.

**Hinkley**

Hinkley, David (1977), On quick choice of power transformation, *Applied Statistics*, **26**, 67-69.

**Hocking**

Hocking, R.R. (1972), Criteria for selection of a subset regression: Which one should be used?, *Technometrics*, **14**, 967-970.

Hocking, R.R. (1973), A discussion of the two-way mixed model, *The American Statistician*, **27**, 148-152.

**Hopfield**

Hopfield, J. J. (1987) *Learning Algorithms and Probability Distributions in Feed-Forward and Feed-Back Networks*, Proceedings of the National Academy of Sciences, **84**, 8429-8433.

**Huber**

Huber, Peter J. (1981), *Robust Statistics*, John Wiley & Sons, New York.

**Hutchinson**

Hutchinson, J. M. (1994) *A Radial Basis Function Approach to Financial Timer Series Analysis*, Ph.D. dissertation, Massachusetts Institute of Technology.

**Hull et al.**

Hull, T.E., W.H. Enright, and K.R. Jackson (1976), *User's guide for DVERK–A subroutine for solving non-stiff ODEs*, Department of Computer Science Technical Report 100, University of Toronto.

**Hwang and Ding**

Hwang, J. T. G. and Ding, A. A. (1997) *Prediction Intervals for Artificial Neural Networks*, Journal of the American Statistical Society, **92**(438) 748-757.

**Irvine et al.**

Irvine, Larry D., Samuel P. Marin, and Philip W. Smith (1986), Constrained interpolation and smoothing, *Constructive Approximation*, **2**, 129-151.

**Jackson et al.**

Jackson, K.R., W.H. Enright, and T.E. Hull (1978), A theoretical criterion for comparing Runge-Kutta formulas, *SIAM Journal of Numerical Analysis*, **15**, 618-641.

**Jacobs et al.**

Jacobs, R. A., Jorday, M. I., Nowlan, S. J., and Hinton, G. E. (1991) Adaptive Mixtures of Local Experts, *Neural Computation*, **3(1)**, 79-87.

**Jenkins**

Jenkins, M.A. (1975), Algorithm 493: Zeros of a real polynomial, *ACM Transactions on Mathematical Software*, **1**, 178-189.

**Jenkins and Traub**

Jenkins, M.A., and J.F. Traub (1970), A three-stage algorithm for real polynomials using quadratic iteration, *SIAM Journal on Numerical Analysis*, 7, 545-566.

Jenkins, M.A., and J.F. Traub (1970), A three-stage variable-shift iteration for polynomial zeros and its relation to generalized Rayleigh iteration, *Numerishe Mathematik*, **14**, 252-263.

Jenkins, M.A., and J.F. Traub (1972), Zeros of a complex polynomial, *Communications of the AC*M, **15**, 97- 99.

**Jhnk**

Jhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufalls-zahlen, *Metrika*, **8**, 5-15.

**Johnson and Kotz**

Johnson, Norman L., and Samuel Kotz (1969), *Discrete Distributions*, Houghton Mifflin Company, Boston.

Johnson, Norman L., and Samuel Kotz (1970a), *Continuous Univariate Distributions-1*, John Wiley & Sons, New York.

Johnson, Norman L., and Samuel Kotz (1970b), *Continuous Univariate Distributions-2*, John Wiley & Sons, New York.

**Jreskog**

Jreskog, M.D. (1977), Factor analysis by least squares and maximum-likelihood methods, *Statistical Methods for Digital Computers*, (edited by Kurt Enslein, Anthony Ralston, and Herbert S. Wilf), John Wiley & Sons, New York, 125-153.

**Kaiser**

Kaiser, H.F. (1963), Image analysis, *Problems in Measuring Change*, (edited by C. Harris), University of Wisconsin Press, Madison, Wis.

**Kaiser and Caffrey**

Kaiser, H.F. and J. Caffrey (1965), Alpha factor analysis, *Psychometrika*, **30**, 1-14.

**Kachitvichyanukul**

Kachitvichyanukul, Voratas (1982), *Computer generation of Poisson, binomial, and hypergeometric random variates*, Ph.D. dissertation, Purdue University, West Lafayette, Indiana.

**Kendall and Stuart**

Kendall, Maurice G., and Alan Stuart (1973), *The Advanced Theory of Statistics*, Volume II, *Inference and Relationship*, Third Edition, Charles Griffin & Company, London, Chapter 30.

**Kennedy and Gentle**

Kennedy, William J., Jr., and James E. Gentle (1980), *Statistical Computing*, Marcel Dekker, New York.

**Kernighan and Richtie**

Kernighan, Brian W., and Richtie, Dennis M. 1988, "The C Programming Language" Second Edition, **241**.

**Kinnucan and Kuki**

Kinnucan, P., and Kuki, H., (1968), *A single precision inverse error function subroutine*, Computation Center, University of Chicago.

**Kirk**

Kirk, Roger, E., (1982), "Experimental Design" Second Edition, *Procedures in Behavioral Sciences*, Brooks/Cole Publishing Company, Monterey, CA.

**Kohonen**

Kohonen, T. (1995) *Self-Organizing Maps*, Springer-Verlag.

**Knuth**

Knuth, Donald E. (1981), The Art of Computer Programming, Volume II: *Seminumerical Algorithms*, 2nd. ed., Addison-Wesley, Reading, Mass.

**Lachenbruch**

Lachenbruch, Peter A. (1975), *Discriminant Analysis*, Hafner Press, London.

**Lawrence et al**

Lawrence, S., Giles, C. L, Tsoi, A. C., Back, A. D. (1997) Face Recognition: A Convolutional Neural Network Approach, *IEEE Transactions on Neural Networks, Special Issue on Neural Networks and Pattern Recognition*, 8(1), 98-113.

**Learmonth and Lewis**

Learmonth, G.P., and P.A.W. Lewis (1973), *Naval Postgraduate School Random Number Generator Package LLRANDOM, NPS55LW73061A*, Naval Postgraduate School, Monterey, California.

**Lehmann**

Lehmann, E.L. (1975), *Nonparametrics: Statistical Methods Based on Ranks*, Holden-Day, San Francisco.

**Levenberg**

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164-168.

**Leavenworth**

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

**Lewis et al.**

Lewis, P.A.W., A.S. Goodman, and J.M. Miller (1969), A pseudorandom number generator for the System/ 360, *IBM Systems Journal*, **8**, 136-146.

**Li**

Li, L. K. (1992) *Approximation Theory and Recurrent Networks*, Proc. Int. Joint Conference On Neural Networks, vol. II, 266-271.

**Liepman**

Liepman, David S. (1964), Mathematical constants, in *Handbook of Mathematical Functions*, Dover Publications, New York.

**Lippmann**

Lippmann, R. P. (1989) *Review of Neural Networks for Speech Recognition,*, Neural Computation, ,b>I, 1-38.

**Liu**

Liu, J.W.H. (1987), *A collection of routines for an implementation of the multifrontal method*, Technical Report CS-87-10, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J.W.H. (1989), The multifrontal method and paging in sparse Cholesky factorization. *ACM Transactions on Mathematical Software*, **15**, 310-325.

Liu, J.W.H. (1990), *The multifrontal method for sparse matrix solution: theory and practice*, Technical Report CS-90-04, Department of Computer Science, York University, North York, Ontario, Canada.

Liu, J.W.H. (1986), On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, **12**, 249-264.

**Loh and Shih**

Loh, W.-Y. and Shih, Y.-S. (1997) Split Selection Methods for Classification Trees, *Statistica Sinica*, **7**, 815-840.

**Lyness and Giunta**

Lyness, J.N. and G. Giunta (1986), A modification of the Weeks Method for numerical inversion of the Laplace transform, *Mathematics of Computation*, **47**, 313-322.

**Madsen and Sincovec**

Madsen, N.K., and R.F. Sincovec (1979), Algorithm 540: PDECOL, General collocation software for partial differential equations, *ACM Transactions on Mathematical Software*,**5**, #3, 326-351.

**Maindonald**

Maindonald, J.H. (1984), *Statistical Computation*, John Wiley & Sons, New York.

**Mandic and Chambers**

Mandic, D. P. and Chambers, J. A. (2001) *Recurrent Neural Networks for Prediction*, John Wiley & Sons, LTD.

**Manning and Schtze**

Manning, C. D. and Schtze, H. (1999) *Foundations of Statistical Natural Language Processing*, MIT Press.

**Marquardt**

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431-441.

**Marsaglia**

Marsaglia, G. (1972), The structure of linear congruential sequences, in *Applications of Number Theory to Numerical Analysis*, (edited by S. K. Zaremba), Academic Press, New York, 249-286.

**Martin and Wilkinson**

Martin, R.S., and J.H. Wilkinson (1971), Reduction of the Symmetric Eigenproblem Ax = lBx and Related Problems to Standard Form, *Volume II, Linear Algebra Handbook*, Springer, New York.

Martin, R.S., and J.H. Wilkinson (1971), The Modified LR Algorithm for Complex Hessenberg Matrices, *Handbook, Volume II, Linear Algebra*, Springer, New York.

**Mayle**

Mayle, Jan, (1993), Fixed Income Securities Formulas for Price, Yield, and Accrued Interest, *SIA Standard Securities Calculation Methods*, Volume I, Third Edition, pages 17-35.

**McCulloch and Pitts**

McCulloch, W. S. and Pitts, W. (1943) A Logical Calculus for Ideas Imminent in Nervous Activity, *Bulletin of Mathematical Biophysics*, **5**, 115-133.

**Michelli**

Micchelli, C.A. (1986), Interpolation of scattered data: Distance matrices and conditionally positive definite functions, *Constructive Approximation*, **2**, 11-22.

**Michelli et al.**

Micchelli, C.A., T.J. Rivlin, and S. Winograd (1976), The optimal recovery of smooth functions, *Numerische Mathematik*, **26**, 279-285.

Micchelli, C.A., Philip W. Smith, John Swetits, and Joseph D. Ward (1985), Constrained $L_p$ approximation, *Constructive Approximation*, **1**, 93-102.

**Microsoft Excel User Education Team**

Microsoft Excel 5 - Worksheet Function Reference, (1994), *Covers Microsoft Excel 5 for Windows*$^{tm}$ and the *Apple Macintosh*$^{tm}$, Microsoft Press. Redmond, VA.

**Moler and Stewart**

Moler, C., and G.W. Stewart (1973), An algorithm for generalized matrix eigenvalue problems, *SIAM Journal on Numerical Analysis*, **10**, 241-256. *Covers Microsoft Excel 5 for Windows*$^{tm}$.

**Mor et al.**

Mor, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74, Argonne, Illinois.

**Mller**

Mller, D.E. (1956), A method for solving algebraic equations using an automatic computer, *Mathematical Tables and Aids to Computation*, **10**, 208-215.

**Murtagh**

Murtagh, Bruce A. (1981), Advanced Linear Programming: Computation and Practice, McGraw-Hill, New York.

**Murty**

Murty, Katta G. (1983), *Linear Programming*, John Wiley and Sons, New York.

**Neter and Wasserman**

Neter, John, and William Wasserman (1974), *Applied Linear Statistical Models*, Richard D. Irwin, Homewood, Illinois.

**Neter et al.**

Neter, John, William Wasserman, and Michael H. Kutner (1983), *Applied Linear Regression Models*, Richard D. Irwin, Homewood, Illinois.

**Østerby and Zlatev**

Østerby, Ole, and Zahari Zlatev (1982), Direct Methods for Sparse Matrices, *Lecture Notes in Computer Science*, **157**, Springer-Verlag, New York.

**Owen**

Owen, D.B. (1962), *Handbook of Statistical Tables*, Addison-Wesley Publishing Company, Reading, Mass.

Owen, D.B. (1965), A special case of the bivariate non-central $t$ distribution, *Biometrika*, **52**, 437-446.

**Pao**

Pao, Y. (1989) *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley Publishing.

**Parlett**

Parlett, B.N. (1980), *The Symmetric Eigenvalue Problem*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

**Petro**

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13**, 624.

**Piessens et al.**

Piessens, R., E. deDoncker-Kapenga, C.W. berhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

**Poli and Jones**

Poli, I. and Jones, R. D. (1994) *A Neural Net Model for Prediction*, Journal of the American Statistical Society, 89(425) 117-121.

**Powell**

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, *Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics*, (edited by G. A. Watson), **630**, Springer-Verlag, Berlin, Germany, 144-157.

Powell, M.J.D. (1985), On the quadratic programming algorithm of Goldfarb and Idnani, *Mathematical Programming Study*, **25**, 46-61.

Powell, M.J.D. (1988), *A tolerant algorithm for linearly constrained optimizations calculations*, DAMTP Report NA17, University of Cambridge, England.

Powell, M.J.D. (1989), *TOLMIN: A Fortran package for linearly constrained optimizations calculations*, DAMTP Report NA2, University of Cambridge, England.

Powell, M.J.D. (1983), *ZQPCVX a FORTRAN subroutine for convex quadratic programming,* DAMTP Report 1983/NA17, University of Cambridge, Cambridge, England.

**Pregibon**

Pregibon, Daryl (1981), Logistic regression diagnostics, *The Annals of Statistics*, **9**, 705-724.

**Quinlan**

Quinlan, J. R. (1993), C4.5 *Programs for Machine Learning*, Morgan Kaufmann.

**Reed and Marks**

Reed, R. D. and Marks, R. J. II (1999) *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, The MIT Press, Cambridge, MA.

**Reinsch**

Reinsch, Christian H. (1967), Smoothing by spline functions, *Numerische Mathematik*, **10**, 177-183.

**Rice**

Rice, J.R. (1983), *Numerical Methods, Software, and Analysis*, McGraw-Hill, New Yor.

**Ripley**

Ripley, B. D. (1994) Neural Networks and Related Methods for Classification, *Journal of the Royal Statistical Society B*, **56(3)**, 409-456.

Ripley, B. D. (1996) *Pattern Recognition and Neural Networks*, Cambridge University Press.

**Rosenblatt**

Rosenblatt, F. (1958) The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain,Psychol. Rev., **65**, 386-408.

**Rumelhart et al**

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. (1986) Learning Representations by Back-Propagating Errors, *Nature*, **323**, 533-536.

Rumelhart, D. E. and McClelland, J. L. eds. (1986) *Parallel Distributed Processing:*

*Explorations in the Microstructure of Cognition*, **1**, 318-362, MIT Press.

**Saad and Schultz**

Saad, Y., and M. H. Schultz (1986), GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems, *SIAM Journal of Scientific and Statistical Computing*, **7**, 856-869.

**Sallas and Lionti**

Sallas, William M., and Abby M. Lionti (1988), Some useful computing formulas for the nonfull rank linear model with linear equality restrictions, IMSL Technical Report 8805, IMSL, Houston.

**Savage**DVERK (1956), Contributions to the theory of rank order statistics–the two-sample case, *Annals of Mathematical Statistics*, **27**, 590-615.

**Schittkowski**

Schittkowski, K. (1987), *More test examples for nonlinear programming codes*, Springer-Verlag, Berlin, **74**.

Schittkowski, K. (1986), NLPQL: A FORTRAN subroutine solving constrained nonlinear programming problems, (edited by Clyde L. Monma), *Annals of Operations Research*, **5**, 485-500.

Schittkowski, K. (1980), Nonlinear programming codes, *Lecture Notes in Economics and Mathematical Systems*, **183**, Springer-Verlag, Berlin, Germany.

Schittkowski, K. (1983), On the convergence of a sequential quadratic programming method with an augmented Lagrangian line search function, *Mathematik Operationsforschung und Statistik, Series Optimization*, **14**, 197-216.

**Schmeiser**

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, in *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154-160.

**Schmeiser and Babu**

Schmeiser, Bruce W., and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917-926.

**Schmeiser and Kachitvichyanukul**

Schmeiser, Bruce, and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81–4, School of Industrial Engineering, Purdue University, West Lafayette, Indiana.

**Schmeiser and Lal**

Schmeiser, Bruce W., and Ram Lal (1980), Squeeze methods for generating gamma variates, *Journal of the American Statistical Association*, **75**, 679-682.

**Seidler and Carmichael**

Seidler, Lee J. and Carmichael, D.R., (editors) (1980), *Accountants' Handbook*, Volume I, Sixth Edition, The Ronald Press Company, New York.

**Shampine**

Shampine, L.F. (1975), Discrete least squares polynomial fits, *Communications of the ACM*, **18**, 179-180.

**Shampine and Gear**

Shampine, L.F. and C.W. Gear (1979), A user's view of solving stiff ordinary differential equations, *SIAM Review*, **21**, 1-17.

**Sincovec and Madsen**

Sincovec, R.F., and N.K. Madsen (1975), Software for nonlinear partial differential equations, *ACM Transactions on Mathematical Software*, **1**, #3, 232-260.

**Singleton**

Singleton, T.C. (1969), Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **12**, 185-187.

**Smith et al.**

Smith, B.T., J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler (1976), *Matrix Eigensystem Routines – EISPACK Guide*, Springer-Verlag, New York.

**Smith**

Smith, M. (1993) *Neural Networks for Statistical Modeling*, New York: Van Nostrand Reinhold.

**Smith**

Smith, P.W. (1990), On knots and nodes for spline interpolation, *Algorithms for Approximation II*, J.C. Mason and M.G. Cox, Eds., Chapman and Hall, New York.

**Stewart**

Stewart, G.W. (1973), Introduction to Matrix Computations, Academic Press, New York.

**Stoer**

Stoer, J. (1985), Principles of sequential quadratic programming methods for solving nonlinear programs, in *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

**Strecok**

Strecok, Anthony J. (1968), On the calculation of the inverse of the error function, *Mathematics of Computation*, **22**, 144-158.

**Stroud and Secrest**

Stroud, A.H., and D.H. Secrest (1963), *Gaussian Quadrature Formulae*, Prentice-Hall, Englewood Cliffs, New Jersey.

**Studenmund**

Studenmund, A. H. (1992) *Using Economics: A Practical Guide*, New York: Harper Collins.

**Swingler**

Swingler, K. (1996) *Applying Neural Networks: A Practical Guide*, Academic Press.

**Temme**

Temme, N.M (1975), On the numerical evaluation of the modified Bessel Function of the third kind, *Journal of Computational Physics*, **19**, 324-337.

**Tesauro**

Tesauro, G. (1990) Neurogammon Wins Computer Olympiad, *Neural Computation*, **1**, 321-323.

**Tezuka**

Tezuka, S. (1995), *Uniform Random Numbers: Theory and Practice.* Academic Publishers, Boston.

**Thompson and Barnett**

Thompson, I.J. and A.R. Barnett (1987), Modified Bessel functions $I_n(z)$ and $K_n(z)$ of real order and complex argument, *Computer Physics Communication*, **47**, 245-257.

**Tukey**

Tukey, John W. (1962), The future of data analysis, *Annals of Mathematical Statistics*, **33**, 1-67.

**Velleman and Hoaglin**

Velleman, Paul F., and David C. Hoaglin (1981), *Applications, Basics, and Computing of Exploratory Data Analysis*, Duxbury Press, Boston.

**Walker**

Walker, H.F. (1988), Implementation of the GMRES method using Householder transformations, *SIAM Journal of Scientific and Statistical Computing*, **9**, 152-163.

**Warner and Misra**

Warner, B. and Misra, M. (1996) Understanding Neural Networks as Statistical Tools, *The American Statistician*, **50(4)** 284-293.

**Watkins**

Watkins, David S., L. Elsner (1991), Convergence of algorithm of decomposition type for the eigenvalue problem, *Linear Algebra Applications*, **143**, pp. 29-47.

**Weeks**

Weeks, W.T. (1966), Numerical inversion of Laplace transforms using Laguerre functions, *J. ACM*, **13**, 419-429.

**Werbos**

Werbos, P. (1974) Beyond Regression: *New Tools for Prediction and Analysis in the Behavioral Science*, PhD thesis, Harvard University, Cambridge, MA.Werbos, P. (1990) Backpropagation Through Time: What It Does and How to do It, Proc.*IEEE*, **78**, 1550-1560.

**Williams and Zipser**

Williams, R. J. and Zipser, D. (1989) A Learning Algorithm for Continuously Running Fully Recurrent Neural Networks, *Neural Computation*, **1**, 270-280.

**Witten and Frank**

Witten, I. H. and Frank, E. (2000) Data Mining: *Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann Publishers.

**Wu**

Wu, S-I (1995) Mirroring Our Thought Processes, *IEEE Potentials*, **14**, 36-41.

# Index