RogueWave SOFTWARE

Accelerating Great Code

JMSL™ NUMERICAL LIBRARY USER GUIDE

Version 7.3

JMSL™ NUMERICAL LIBRARY USER GUIDE

Version 7.3

© 1970-2016 Rogue Wave Software, IMSL and PV-WAVE are registered trademarks of Rogue Wave Software, Inc. in the U.S. and other countries. JMSL, JWAVE, TS-WAVE, PyIMSL are trademarks of Rogue Wave Software, Inc. or its subsidiaries. All other company, product or brand names are the property of their respective owners.

IMPORTANT NOTICE: Information contained in this documentation is subject to change without notice. Use of this document is subject to the terms and conditions of a Rogue Wave Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability. If you do not accept the terms of the license agreement, you may not use this documentation and should promptly return the product for a full refund. This documentation may not be copied or distributed in any form without the express written consent of Rogue Wave.

Company Information

Rogue Wave Software, Inc.

Phone: (303) 473-9118 Toll Free: (800) 487-3217

www.roguewave.com

Support: support@roguewave.com

Sales: Request Demos, Trials, and Quotes from www.roguewave.com

Full contact information: www.roguewave.com/company/contact-us

Contents

1 Introduction **1**

2 Linear Systems **5**

Matrix 9

ComplexMatrix 14

SparseMatrix 19

ComplexSparseMatrix 30

LU 40

SuperLU 45

ComplexLU 60

ComplexSuperLU 65

Cholesky 80

SparseCholesky 85

ComplexSparseCholesky 92

QR 99

SVD 103

GenMinRes 108

ConjugateGradient 128

SingularMatrixException 141

3 Eigensystem Analysis **143**

Eigen 144

SymEigen 148

4 Interpolation and Approximation **153**
Spline 155
CsAkima 159
CsTCB 160
CsInterpolate 166
CsPeriodic 168
CsShape 170
CsSmooth 172
CsSmoothC2 174
BSpline 176
BsInterpolate 180
BsLeastSquares 182
Spline2D 184
Spline2DInterpolate 188
Spline2DLeastSquares 197
RadialBasis 202

5 Quadrature **217**
Quadrature 218
HyperRectangleQuadrature 225

6 Differential Equations **229**
ODE 230
OdeRungeKutta 236
OdeAdamsGear 240
FeynmanKac 251

7 Transforms **317**
FFT 318
ComplexFFT 322

8 Nonlinear Equations **327**
ZeroPolynomial 328

ZerosFunction 333
ZeroSystem 339

9 Optimization **349**
MinUncon 351
MinUnconMultiVar 357
NonlinLeastSquares 369
SparseLP 380
DenseLP 406
QuadraticProgramming 421
MinConGenLin 429
BoundedLeastSquares 441
BoundedVariableLeastSquares 451
NonNegativeLeastSquares 456
MinConNLP 461
NumericalDerivatives 490

10 Special Functions **511**
Sfun 511
Bessel 528
JMath 534
IEEE 542
Hyperbolic 544

11 Miscellaneous **549**
Complex 549
Physical 569
EpsilonAlgorithm 580

12 Printing Functions **583**
PrintMatrix 583
PrintMatrixFormat 588

13 Basic Statistics **595**
Summary . . . 595
Covariances . . . 607
PartialCovariances . . . 617
PooledCovariances . . . 623
NormOneSample . . . 632
NormTwoSample . . . 638
Sort . . . 650
Ranks . . . 659
EmpiricalQuantiles . . . 668
TableOneWay . . . 671
TableTwoWay . . . 676
TableMultiWay . . . 682

14 Regression **691**
RegressorsForGLM . . . 698
LinearRegression . . . 708
NonlinearRegression . . . 720
UserBasisRegression . . . 736
RegressionBasis . . . 741
SelectionRegression . . . 741
StepwiseRegression . . . 756

15 Analysis of Variance **771**
ANOVA . . . 771
ANOVAFactorial . . . 781
ANCOVA . . . 791
MultipleComparisons . . . 803

16 Categorical and Discrete Data Analysis **805**
ContingencyTable . . . 805
CategoricalGenLinModel . . . 818

17 Nonparametric Statistics **845**
SignTest 845
WilcoxonRankSum 849

18 Tests of Goodness of Fit **857**
ChiSquaredTest 857
NormalityTest 864
KolmogorovOneSample 869
KolmogorovTwoSample 872

19 Time Series and Forecasting **877**
AutoCorrelation 880
ARAutoUnivariate 890
ARSeasonalFit 915
ARMA 928
ARMAEstimateMissing 954
ARMAMaxLikelihood 964
ARMAOutlierIdentification 981
AutoARIMA 1000
CrossCorrelation 1024
Difference 1035
GARCH 1040
KalmanFilter 1050
MultiCrossCorrelation 1062
LackOfFit 1076
HoltWintersExponentialSmoothing 1079
TimeSeries 1088
TimeSeriesOperations 1096
VectorAutoregression 1108

20 Multivariate Analysis **1115**
ClusterKMeans 1117

ClusterKNN . . . 1134
Dissimilarities . . . 1139
ClusterHierarchical . . . 1146
FactorAnalysis . . . 1155
DiscriminantAnalysis . . . 1174

21 Survival and Reliability Analysis 1199
KaplanMeierECDF . . . 1199
KaplanMeierEstimates . . . 1203
ProportionalHazards . . . 1211
LifeTables . . . 1231

22 Probability Distribution Functions and Inverses 1237
Cdf . . . 1239
Pdf . . . 1280
InvCdf . . . 1296
CdfFunction . . . 1307
InverseCdf . . . 1308
Distribution . . . 1311
ProbabilityDistribution . . . 1311
NormalDistribution . . . 1313
GammaDistribution . . . 1315
LogNormalDistribution . . . 1317
PoissonDistribution . . . 1319

23 Random Number Generation 1323
Random . . . 1324
FaureSequence . . . 1349
MersenneTwister . . . 1353
MersenneTwister64 . . . 1357
RandomSequence . . . 1361
RandomSamples . . . 1362

24 Input/Output 1373

AbstractFlatFile . . . 1373
FlatFile . . . 1423
Tokenizer . . . 1456
MPSReader . . . 1457

25 Finance 1471

BasisPart . . . 1472
Bond . . . 1473
DayCountBasis . . . 1527
Finance . . . 1530

26 Chart 2D 1561

Chart . . . 1562
AbstractChartNode . . . 1567
ChartNode . . . 1586
Background . . . 1608
ChartTitle . . . 1609
Legend . . . 1609
Annotation . . . 1610
Grid . . . 1612
Axis . . . 1613
AxisXY . . . 1615
Axis1D . . . 1617
AxisLabel . . . 1622
AxisLine . . . 1623
AxisTitle . . . 1623
AxisUnit . . . 1624
MajorTick . . . 1625
MinorTick . . . 1625
Transform . . . 1626
TransformDate . . . 1627

AxisR . . . 1628
AxisRLabel . . . 1630
AxisRLine . . . 1631
AxisRMajorTick . . . 1631
AxisTheta . . . 1632
GridPolar . . . 1633
Data . . . 1634
ChartFunction . . . 1645
ChartSpline . . . 1645
Text . . . 1646
ToolTip . . . 1648
FillPaint . . . 1650
Draw . . . 1653
JFrameChart . . . 1664
JPanelChart . . . 1665
DrawPick . . . 1667
PickEvent . . . 1673
PickListener . . . 1674
JspBean . . . 1675
ChartServlet . . . 1678
DrawMap . . . 1679
BoxPlot . . . 1685
Contour . . . 1695
ErrorBar . . . 1703
HighLowClose . . . 1708
Candlestick . . . 1714
CandlestickItem . . . 1716
SplineData . . . 1717
Bar . . . 1720
BarItem . . . 1726
BarSet . . . 1727

Pie 1728
PieSlice 1732
Dendrogram 1733
Polar 1741
Heatmap 1745
Treemap 1755
Colormap 1764
ChartXML 1766

27 Quality Control and Improvement Charts **1771**

ShewhartControlChart 1771
ControlLimit 1777
XbarR 1779
RChart 1785
XbarS 1789
SChart 1796
XmR 1799
NpChart 1802
PChart 1805
CChart 1809
UChart 1812
EWMA 1815
CuSum 1819
CuSumStatus 1822
ParetoChart 1830

28 Chart 3D **1837**

Chart3D 1837
JFrameChart3D 1840
ChartNode3D 1842
Background 1852
Canvas3DChart 1852

BufferedPaint 1856
ChartLights 1857
AmbientLight 1857
DirectionalLight 1858
PointLight 1860
AxisXYZ 1861
AxisBox 1863
Axis3D 1865
AxisLabel 1868
AxisLine 1869
AxisTitle 1869
MajorTick 1870
Surface 1870
Data 1881
ColorFunction 1893
ColormapLegend 1893

29 Data Mining **1897**

NaiveBayesClassifier 1902
Itemsets 1923
AssociationRule 1925
Apriori 1926
KohonenSOM 1934
KohonenSOMTrainer 1942
PredictiveModel 1945
BootstrapAggregation 1962
CrossValidation 1969
GradientBoosting 1976

30 Support Vector Machines **2003**

SupportVectorMachine 2003
SVClassification 2013

SVOneClass 2025
SVRegression 2029
Kernel 2037
LinearKernel 2039
SigmoidKernel 2040
RadialBasisKernel 2042
PolynomialKernel 2044
DataNode 2045

31 Neural Nets **2047**

Network 2090
FeedForwardNetwork 2099
Layer 2113
InputLayer 2114
HiddenLayer 2115
OutputLayer 2116
Node 2118
InputNode 2118
Perceptron 2119
OutputPerceptron 2120
Activation 2121
Link 2123
Trainer 2124
QuasiNewtonTrainer 2125
LeastSquaresTrainer 2134
EpochTrainer 2138
BinaryClassification 2144
MultiClassification 2187
ScaleFilter 2202
UnsupervisedNominalFilter 2211
Example: UnsupervisedNominalFilter 2214
UnsupervisedOrdinalFilter 2215

Example: UnsupervisedOrdinalFilter . . . 2219
TimeSeriesFilter . . . 2220
TimeSeriesClassFilter . . . 2222

32 Decision Trees **2227**
TreeNode . . . 2229
Tree . . . 2234
DecisionTree . . . 2237
DecisionTreeInfoGain . . . 2265
ALACART . . . 2269
C45 . . . 2275
CHAID . . . 2283
QUEST . . . 2289
RandomTrees . . . 2298

33 Maximum Likelihood Estimation **2311**
MaximumLikelihoodEstimation . . . 2312
ProbabilityDistribution . . . 2321
PDFGradientInterface . . . 2324
PDFHessianInterface . . . 2324
ClosedFormMaximumLikelihoodInterface . . . 2325
MethodOfMomentsInterface . . . 2326
BetaPD . . . 2326
ContinuousUniformPD . . . 2329
ExponentialPD . . . 2333
GammaPD . . . 2336
NormalPD . . . 2339

34 Error Handling **2345**
Messages . . . 2345
Version . . . 2347
IMSLFormatter . . . 2347

Warning 2348
WarningObject 2351
IMSLException 2352
IMSLRuntimeException 2353
IMSLUnexpectedErrorException 2354
LicenseManagerException 2354

35 References **2357**

Index **i**

Chapter 1: Introduction

Distributed Computing and Parallelism

Distributed computing refers to breaking a problem down into parts and the communication of the parts across a network. Parallel computing refers to the simultaneous processing of the parts, often on the same processor or multiple processors which share memory.

The Adobe™ Hadoop® is a Java open-source project for distributed computing. Certain JMSL classes can be used in conjunction with Hadoop for data mining large data sets on a cluster. Example JMSL classes include Linear Regression and the Apriori class used for market basket analysis.

The JMSL Library utilizes Java Threads to implement parallel processing. The JMSL implementation provides methods for specifying the number of threads to be used in selected classes which may benefit most from parallel processing, such as those in the Optimization, Time Series, and Data Mining categories.

JMSL Concurrency (Java Threads)

The JMSL Library utilizes Java `Thread`s to implement parallel processing. The choice whether to leverage this capability and to what extent is left to the user. The advantage of parallel processing is frequently determined by available hardware resources and simultaneous processes. This is such a complex issue that significant research is ongoing in this area of technology.

The JMSL implementation provides accessor and mutator methods for specifying the number of `Threads` to be used in selected classes that may benefit most from parallel processing. These classes include the `Matrix` class as well as several classes from the Optimization, Time Series, and Data Mining categories.

The JMSL parallel API occurs in two contexts. The first is a class mutator. For these cases, the mutator prototype is “`setNumberOfThreads(int numberOfThreads)`“. Classes that implement this mutator method have the capacity to operate in parallel. The second context is a method (usually static) where the ”`numberOfThreads`“ argument is one of multiple arguments. For example, the overloaded parallel version of `Matrix.multiply` is `Matrix.multiply(double[][] a, double[][] b, int numberOfThreads)`.

JMSL’s elementary mechanism for issuing parallel analysis provides flexibility and control while also providing all of the potential performance benefits.

Compressed Sparse Column (CSC) Format

Classes that accept data in coordinate format can also accept data stored in the format described in the Users' Guide for the Harwell-Boeing Sparse Matrix Collection. The scheme is column oriented, with each column held as a sparse vector, represented by a list of the row indices of the entries in an integer array ("`rowInd`" below) and a list of the corresponding values in a separate double array ("`values`" below). Data for each column is stored consecutively and the columns are stored in order. A third array ("`colPtr`" below) indicates the location in array "`values`" in which to place the first nonzero value of each succeeding column of the original sparse matrix. So "`colPtr[i]`" contains the index of the first free location in array "`values`" in which to place the values from the i-th column of the original sparse matrix. In other words, "`values[colPtr[i]]`" holds the first nonzero value of the i-th column of the original sparse matrix. Only entries in the lower triangle and diagonal are stored for symmetric and Hermitian matrices. All arrays are based at zero, which is in contrast to the Harwell-Boeing test suite's one-based arrays.

As in the Harwell-Boeing user guide (link above), the storage scheme is illustrated with the following example: The 5 x 5 matrix

$$\begin{bmatrix} 1 & -3 & 0 & -1 & 0 \\ 0 & 0 & -2 & 0 & 3 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & -4 & 0 \\ 5 & 0 & -5 & 0 & 6 \end{bmatrix}$$

would be stored in the arrays `colPtr` (location of first entry), `rowInd` (row indices), and `values` (nonzero entries) as follows:

Subscripts	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
colPtr	0	3	5	7	9	11					
rowInd	0	2	4	0	3	1	4	0	3	1	4
values	1	2	5	-3	4	-2	-5	-1	-4	3	6

The following program fragment shows the relationship between CSC storage format and coordinate representation:

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

public class CSCFormat {

    public static void main(String args[]) {
        int n = 5;
        int[] colPtr = {0, 3, 5, 7, 9, 11};
        int[] rowInd = {0, 2, 4, 0, 3, 1, 4, 0, 3, 1, 4};
        double[] values = {
            1.0, 2.0, 5.0, -3.0, 4.0, -2.0, -5.0, -1.0, -4.0, 3.0, 6.0
        };
        SparseMatrix a = new SparseMatrix(n, n);

        // Convert CSC format to coordinate format.
```

```
        // Save coordinate format in a SparseMatrix object.
        for (int i = 0; i < n; i++) {
            int start = colPtr[i];
            int stop = colPtr[i + 1];

            for (int j = start; j < stop; j++) {
                // a.set(row, column, value);
                a.set(rowInd[j], i, values[j]);
            }
        }
    }
}
```

Chapter 2: Linear Systems

Types

class Matrix 9
class ComplexMatrix 14
class SparseMatrix 19
class ComplexSparseMatrix 30
class LU 40
class SuperLU 45
class ComplexLU 60
class ComplexSuperLU 65
class Cholesky 80
class SparseCholesky 85
class ComplexSparseCholesky 92
class QR 99
class SVD 103
class GenMinRes 108
class ConjugateGradient 128
exception SingularMatrixException 141

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 $A_{m \mathrm{x} n}x = b$, 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 $R(P^T)x = Q^T b$ 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 $U_{m \mathrm{x} q}$ and $V_{n \mathrm{x} q}$ are orthogonal matrices, and $S_{q \mathrm{x} q}$ 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.

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

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.

Sparse Matrix Storage Modes

All JMSL classes that work with sparse matrices (e.g. classes `SparseCholesky`, `SuperLU`, `ComplexSparseCholesky` and `ComplexSuperLU`) require a matrix input format that only stores information about the nonzero entries of these matrices. JMSL supports two such formats: The sparse coordinate storage (SCS) format and the Java Sparse Array (JSA) format. The SCS format stores the value of each matrix entry together with that entry's row and column index. The JSA format stores the sparse matrix in two arrays of arrays. One array contains the references to the nonzero value arrays for each row of the matrix, the other contains the references to the associated column index arrays. As an example, consider the following sparse 6×6 matrix:

$$A = \begin{pmatrix} 10 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10 & -3 & -1 & 0 & 0 \\ 0 & 0 & 15 & 0 & 0 & 0 \\ -2 & 0 & 0 & 10 & -1 & 0 \\ -1 & 0 & 0 & -5 & 1 & -3 \\ -1 & -2 & 0 & 0 & 0 & 6 \end{pmatrix}$$

This matrix has 15 nonzero elements, and the sparse coordinate representation would be

```
row   0   1   1   1   2   3   3   3   4   4  4   4   5   5  5
col   0   1   2   3   2   0   3   4   0   3  4   5   0   1  5
val  10  10  -3  -1  15  -2  10  -1  -1  -5  1  -3  -1  -2  6
```

Since this representation does not rely on the order of the matrix elements, an equivalent form would be

```
row   5   4   3   0   5   1   2   1   4    3    1   4   3  5   4
col   0   0   0   0   1   1   2   2   3    3    3   4   4  5   5
val  -1  -1  -2  10  -2  10  15  -3  -5   10   -1   1  -1  6  -3
```

In JSA format, matrix *A* can be represented by the following two arrays, `index` and `values`:

```
index:
row 0: 0
row 1: 1, 2, 3
row 2: 2
row 3: 0, 3, 4
row 4: 0, 3, 4, 5
row 5: 0, 1, 5

values:
row 0: 10
row 1: 10, -3, -1
row 2: 15
row 3: -2, 10, -1
row 4: -1, -5, 1, -3
row 5: -1, -2, 6
```

In contrast to the SCS format, the row order is fixed here, but the entries within each row do not rely on any order.

In JMSL, sparse matrices are instances of classes `SparseMatrix/ComplexSparseMatrix`. Use of the SCS and JSA input format is reflected by different constructors. The following code fragment shows how a user can construct a `SparseMatrix` for the matrix *A* above in SCS format:

```
// Generate an empty 6 by 6 SparseMatrix object
int nRows = nColumns = 6;
SparseMatrix a = new SparseMatrix(nRows, nColumns);
// Fill object a with the entries of matrix A,
// A given in SCS format
a.set(0, 0, 10.0); a.set(1, 1, 10.0); a.set(1, 2, -3.0);
a.set(1, 3, -1.0); a.set(2, 2, 15.0); a.set(3, 0, -2.0);
a.set(3, 3, 10.0); a.set(3, 4, -1.0); a.set(4, 0, -1.0);
a.set(4, 3, -5.0); a.set(4, 4, 1.0); a.set(4, 5, -3.0);
a.set(5, 0, -1.0); a.set(5, 1, -2.0); a.set(5, 5, 6.0);
```

Similarly, the following code fragment shows how to construct a `SparseMatrix` in JSA format:

```
// Generate matrix A in JSA format
int nRows = nColumns = 6;
int[][] aIndex =   { {0},
                   {1, 2, 3},
                   {2},
                   {0, 3, 4},
                   {0, 3, 4, 5},
                   {0, 1, 5}};

int[][] aValues =  { {10.0},
```

```
                   {10.0, -3.0, -1.0},
                   {15.0},
                   {-2.0, 10.0, -1.0},
                   {-1.0, -5.0, 1.0, -3.0},
                   {-1.0, -2.0, 6.0}};

// Create SparseMatrix object based on A in JSA format
SparseMatrix a = new SparseMatrix(nRows, nColumns, aIndex, aValues);
```

All JMSL algorithms working on sparse matrices use `SparseMatrix` or `ComplexSparseMatrix` as an input argument type. A $n \times n$ matrix is called *symmetric* if $A^T = A$, where A^T denotes the tranpose of A. It is called *Hermitian* if $A^H = A$. Here, $A^H \equiv \bar{A}^T$ denotes the conjugate transpose of A. For these types of matrices, all matrix information is stored in the lower triangular part of the matrix. Therefore, for all JMSL sparse matrix classes working on symmetric or Hermitian matrices it is sufficient to store only the lower triangle of the matrix in the `SparseMatrix` or `ComplexSparseMatrix` input object.

The complex 4×4 matrix

$$H = \begin{pmatrix} 4 & 1-i & 0 & 0 \\ 1+i & 4 & 1-i & 0 \\ 0 & 1+i & 4 & 1-i \\ 0 & 0 & 1+i & 4 \end{pmatrix}$$

is Hermitian positive definite. Typically, matrices of this type are input to methods of class `ComplexSparseCholesky`.

The following code fragment shows how the lower triangular part of H can be constructed as a `ComplexSparseMatrix` in SCS input format:

```
// Generate an empty 4 by 4 ComplexSparseMatrix object
int nRows = nColumns = 4;
ComplexSparseMatrix h = new ComplexSparseMatrix(nRows, nColumns);
// Store lower triangular part of H in object h
h.set(0, 0, new Complex(4.0,0.0));
h.set(1, 0, new Complex(1.0, 1.0)); h.set(1, 1, new Complex(4.0,0.0));
h.set(2, 1, new Complex(1.0,1.0)); h.set(2, 2, new Complex(4.0, 0.0));
h.set(3, 2, new Complex(1.0, 1.0)); h.set(3, 3, new Complex(4.0, 0.0));
```

Matrix class

```
public class com.imsl.math.Matrix
```

Matrix manipulation functions.

Methods

add

```
static public 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

checkMatrix

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

Description

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

Parameter

a – a double matrix

checkSquareMatrix

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

Description

Check that the matrix is square.

Parameter

a – a double matrix

frobeniusNorm

```
static public double frobeniusNorm(double[][] a)
```

Description

Return the Frobenius norm of a matrix.

Parameter

a – a double rectangular array

Returns

a double scalar value equal to the Frobenius norm of the matrix.

infinityNorm

```
static public double infinityNorm(double[][] a)
```

Description

Return the infinity norm of a matrix.

Parameter

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

inverseLowerTriangular

```
static public double[][] inverseLowerTriangular(double[][] a)
```

Description

Returns the inverse of the lower triangular matrix a.

Parameter

a – a double square lower triangular matrix

Returns

a double matrix containing the inverse of a

inverseUpperTriangular

```
static public double[][] inverseUpperTriangular(double[][] a)
```

Description

Returns the inverse of the upper triangular matrix a.

Parameter

a – a double square upper triangular matrix

Returns

a double matrix containing the inverse of a

multiply

```
static public 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 vector representing the product of the arguments, x*a.

multiply

```
static public 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

multiply

```
static public 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

multiply

```
static public double[][] multiply(double[][] a, double[][] b, int
numberOfThreads)
```

Description

Multiply two rectangular arrays, a * b, using multiple java.lang.Threads.

Parameters

a – a double rectangular array

b – a double rectangular array

numberOfThreads – An int which specifies the number of java.lang.Thread instances to use. If numberOfThreads is less than 1, then numberOfThreads = 1 is used.

Returns

the double matrix product of a times b

oneNorm

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

Description

Return the matrix one norm.

Parameter

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

`static public 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

transpose

`static public double[][] transpose(double[][] a)`

Description

Return the transpose of a matrix.

Parameter

`a` – a `double` matrix

Returns

a `double` matrix which is the transpose of the argument

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
```

ComplexMatrix class

`public class com.imsl.math.ComplexMatrix`

Complex matrix manipulation functions.

Methods

add

`static public 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

Exception

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

```
static public void checkMatrix(Complex[][] a)
```

Description

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

Parameter

a – a Complex matrix

Exception

IllegalArgumentException This exception is thrown when the lengths of the rows of the input matrix are not uniform.

checkSquareMatrix

```
static public void checkSquareMatrix(Complex[][] a)
```

Description

Check that the Complex matrix is square.

Parameter

a – a Complex matrix

Exception

IllegalArgumentException This exception is thrown when the matrix is not square..

frobeniusNorm

```
static public double frobeniusNorm(Complex[][] a)
```

Description

Return the Frobenius norm of a Complex matrix.

Parameter

a – a Complex rectangular matrix

Returns

a double value equal to the Frobenius norm of the matrix

Exception

IllegalArgumentException This exception is thrown when the lengths of the rows of the input matrix is not uniform.

infinityNorm

```
static public double infinityNorm(Complex[][] a)
```

Description

Return the infinity norm of a `Complex` matrix.

Parameter

`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.

Exception

`IllegalArgumentException` This exception is thrown when the lengths of the rows of the input matrix is not uniform.

---

multiply

```
static public 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`.

Exception

`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.

---

multiply

```
static public 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`

Exception

`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

```
static public 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`

Exception

`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

```
static public Complex[][] multiply(Complex[][] a, Complex[][] b, int
numberOfThreads)
```

Description

Multiply two `Complex` rectangular arrays, `a` * `b`, using multiple `java.lang.Threads`.

Parameters

`a` – a `Complex` rectangular array

`b` – a `Complex` rectangular array

`numberOfThreads` – An `int` which specifies the number of `java.lang.Thread` instances to use. If `numberOfThreads` is less than 1, then `numberOfThreads` = 1 is used.

Returns

the `Complex` matrix product of `a` times `b`

Exception

`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`.

---

oneNorm

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

Description

Return the `Complex` matrix one norm.

Parameter

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

Exception

`IllegalArgumentException` This exception is thrown when the lengths of the rows of the input matrix is not uniform.

---

subtract

```
static public 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.

Exception

`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

```
static public Complex[][] transpose(Complex[][] a)
```

Description

Return the transpose of a `Complex` matrix.

Parameter

a – a `Complex` matrix

Returns

the `Complex` matrix transpose of the argument

Exception

`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
```

SparseMatrix class

`public class com.imsl.math.SparseMatrix implements Serializable`

Sparse matrix of type `double`. The class represents a general real sparse matrix. It is intended to be efficiently and easily updated.

A `SparseMatrix` can be constructed from a set of arrays, or it can be abstractly created as an empty array and then incrementally built into final form. It is usually easier to create an empty `SparseMatrix` of set size and then use the `set` method to set the elements of the array. When setting the elements of the sparse array, their positions should be thought of as their positions in the dense array. Elements can be set in any order, but only the elements actually set are stored.

This class includes methods to update the sparse matrix. There are also methods to multiply a sparse

matrix and a vector or to multiply two sparse matrices. To solve a sparse linear system use `SparseCholesky` or `SuperLU`.

Constructors

SparseMatrix

```
public SparseMatrix(SparseMatrix A)
```

Description

Creates a new instance of `SparseMatrix` which is a copy of another `SparseMatrix`.

Parameter

`A` – the `SparseMatrix` object containing the sparse matrix to be copied.

SparseMatrix

```
public SparseMatrix(SparseMatrix.SparseArray jsa)
```

Description

Constructs a sparse matrix from a `SparseArray` object.

Parameter

`jsa` – is a `SparseArray` used to initialize the sparse matrix. The field `numberOfNonZeros` in `jsa` is not used for initialization, so it does not have to be set.

SparseMatrix

```
public SparseMatrix(int nRows, int nColumns)
```

Description

Creates a new instance of `SparseMatrix`. Initially this is the zero matrix.

Parameters

`nRows` – an `int` containing the number of rows in the sparse matrix.

`nColumns` – an `int` containing the number of columns in the sparse matrix.

SparseMatrix

```
public SparseMatrix(int nRows, int nColumns, int[][] index, double[][] values)
```

Description

Constructs a sparse matrix from SparseArray (Java Sparse Array) data.

Parameters

`nRows` – an `int` containing the number of rows in the sparse matrix.

`nColumns` – an `int` containing the number of columns in the sparse matrix.

`index` – an `int` jagged array containing the column indices of all nonzero elements corresponding to the compressed representation of the sparse matrix in `values`. The size of `index` must be identical to the size of `values`. The *i*-th row contains the column indices of all nonzero elements of row *i* of the sparse matrix. The *j*-th element of row *i* is the column index of the value located at the same position in `values`.

`values` – a `double` jagged array containing the compressed representation of a real sparse matrix of size `nRows` by `nColumns`. The number of rows in `values` must be `nRows`. The *i*-th row contains all nonzero elements of row *i* of the full sparse matrix.

Methods

add

```
static public SparseMatrix add(double alpha, double beta, SparseMatrix A,
SparseMatrix B)
```

Description

Performs element-wise addition of two real sparse matrices `A`, `B` of type `SparseMatrix`, $C \leftarrow \alpha A + \beta B$.

Parameters

`alpha` – a `double` scalar.

`beta` – a `double` scalar.

`A` – a `SparseMatrix` matrix.

`B` – a `SparseMatrix` matrix.

Returns

a `SparseMatrix` matrix representing the computed sum.

Exception

`IllegalArgumentException` This exception is thrown when the matrices are not of the same size.

checkSquareMatrix

```
public void checkSquareMatrix()
```

Description

Check that the matrix is square.

Exception

`IllegalArgumentException` is thrown if the matrix is not square.

frobeniusNorm

```
public double frobeniusNorm()
```

Description

Returns the Frobenius norm of the matrix.

Returns

a `double` scalar value equal to the Frobenius norm of the matrix.

get

```
public double get(int iRow, int jColumn)
```

Description

Returns the value of an element in the matrix.

Parameters

`iRow` – an `int` containing the row index of the element.

`jColumn` – an `int` containing the column index of the element.

Returns

a `double` containing the value of the `iRow`-th and `jColumn`-th element. If the element was never set, its value is zero.

getNumberOfColumns

```
public int getNumberOfColumns()
```

Description

Returns the number of columns in the matrix.

Returns

an `int` containing the number of columns in the matrix.

getNumberOfNonZeros

```
public long getNumberOfNonZeros()
```

Description

Returns the number of nonzeros in the matrix.

Returns

a `long` containing the number of nonzeros in the matrix.

getNumberOfRows

```
public int getNumberOfRows()
```

Description

Returns the number of rows in the matrix.

Returns

an `int` containing the number of rows in the matrix.

infinityNorm

```
public double infinityNorm()
```

Description

Returns the infinity norm of the matrix.

Returns

a `double` scalar value equal to the maximum of the row sums of the absolute values of the array elements of the sparse matrix.

multiply

```
public double[] multiply(double[] x)
```

Description

Multiply the matrix by a vector.

Parameter

`x` – a `double` column vector.

Returns

a `double` vector representing the product of this matrix times x.

Exception

`IllegalArgumentException` This exception is thrown if the number of columns in the `SparseMatrix` object is not equal to the number of elements in the input column vector.

multiply

```
static public SparseMatrix multiply(SparseMatrix A, SparseMatrix B)
```

Description

Multiply two sparse matrices A and B, $C \leftarrow AB$.

Parameters

`A` – a `SparseMatrix` sparse matrix.

`B` – a `SparseMatrix` sparse matrix.

Returns

the `SparseMatrix` product AB of A and B.

Exception

`IllegalArgumentException` This exception is thrown when the number of columns of matrix A is not equal to the number of rows of matrix B.

multiply

```
static public double[] multiply(SparseMatrix A, double[] x)
```

Description

Multiply sparse matrix `A` and column array `x`, Ax.

Parameters

`A` – a `SparseMatrix` matrix.

`x` – a `double` column array.

Returns

a `double` vector representing the product of the arguments, Ax.

Exception

`IllegalArgumentException` This exception is thrown when the number of columns in the input matrix is not equal to the number of elements in the input column vector.

---

multiply

```
static public double[] multiply(double[] x, SparseMatrix A)
```

Description

Multiply row array `x` and sparse matrix `A`, $x^T A$.

Parameters

`x` – a `double` row array.

`A` – a `SparseMatrix` matrix.

Returns

a `double` vector representing the product of the arguments, $x^T A$.

Exception

`IllegalArgumentException` This exception is thrown when the number of elements in the input vector is not equal to the number of rows of the matrix.

---

multiplySymmetric

```
static public double[] multiplySymmetric(SparseMatrix A, double[] x)
```

Description

Multiply sparse symmetric matrix `A` and column vector `x`.

Parameters

`A` – a `SparseMatrix` sparse symmetric matrix, where only the lower triangular part of the matrix is to be used.

`x` – a `double` vector.

Returns

a `double` vector representing the product of the arguments, Ax.

Exception

`IllegalArgumentException` This exception is thrown when the input matrix is not square or the number of columns in the input matrix is not equal to the number of elements in the input column vector.

---

oneNorm

```
public double oneNorm()
```

Description

Returns the matrix one norm of the sparse matrix.

Returns

a `double` value equal to the maximum of the column sums of the absolute values of the array elements.

---

plusEquals

```
public double plusEquals(int iRow, int jColumn, double x)
```

Description

Adds a value to an element in the matrix.

Parameters

`iRow` – an `int` containing the row index of the element.

`jColumn` – an `int` containing the column index of the element.

`x` – a `double` containing the value to be added to the `iRow`-th and `jColumn`-th element.

Returns

a `double` containing the updated value of the element, which equals its old value plus x.

---

set

```
public void set(int iRow, int jColumn, double x)
```

Description

Sets the value of an element in the matrix.

Parameters

`iRow` – an `int` containing the row index of the element.

`jColumn` – an `int` containing the column index of the element.

`x` – a `double` containing the value of the `iRow`-th and `jColumn`-th element.

---

toDenseMatrix

```
public double[][] toDenseMatrix()
```

Description

Returns the sparse matrix as a dense matrix.

Returns

a rectangular Java array of type `double` containing this matrix with all of the zeros explicitly present. The number of rows and columns in the returned matrix is the same as in the sparse matrix.

---

toSparseArray

```
public SparseMatrix.SparseArray toSparseArray()
```

Description

Returns the sparse matrix in the `SparseArray` form.

transpose

`public SparseMatrix transpose()`

Description

Returns the transpose of the matrix.

Returns

a `SparseMatrix` object which is the transpose of the constructed `SparseMatrix` object.

Example 1: SparseMatrix

The matrix product of two matrices is computed using a method from the SparseMatrix class. The one norm of the result is also computed. The matrix and its norm are printed.

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

public class SparseMatrixEx1 {

    public static void main(String args[]) {
        SparseMatrix b = new SparseMatrix(6, 6);
        b.set(0, 0, 10.0);
        b.set(1, 1, 10.0);
        b.set(1, 2, -3.0);
        b.set(1, 3, -1.0);
        b.set(2, 2, 15.0);
        b.set(3, 0, -2.0);
        b.set(3, 3, 10.0);
        b.set(3, 4, -1.0);
        b.set(4, 0, -1.0);
        b.set(4, 3, -5.0);
        b.set(4, 4, 1.0);
        b.set(4, 5, -3.0);
        b.set(5, 0, -1.0);
        b.set(5, 1, -2.0);
        b.set(5, 5, 6.0);

        SparseMatrix c = new SparseMatrix(6, 3);
        c.set(0, 0, 5.0);
        c.set(1, 2, -3.0);
        c.set(2, 0, 1.0);
        c.set(2, 2, 7.0);
        c.set(3, 0, 2.0);
        c.set(4, 1, -5.0);
        c.set(4, 2, 2.0);
        c.set(5, 2, 4.0);

        SparseMatrix A = SparseMatrix.multiply(b, c);

        // Get the one norm of matrix A
        System.out.println("The 1-norm of the matrix is " + A.oneNorm());
```

```
        SparseMatrix.SparseArray sa = A.toSparseArray();

        // Print the matrix and its one norm
        System.out.println("row  column  value");
        for (int i = 0; i < sa.numberOfRows; i++) {
            for (int j = 0; j < sa.index[i].length; j++) {
                int jj = sa.index[i][j];
                System.out.println(" " + i + "     " + jj + "     "
                        + sa.values[i][j]);
            }
        }
    }
}
```

Output

```
The 1-norm of the matrix is 198.0
row  column  value
 0     0     50.0
 1     0     -5.0
 1     2     -51.0
 2     0     15.0
 2     2     105.0
 3     0     10.0
 3     1     5.0
 3     2     -2.0
 4     0     -15.0
 4     1     -5.0
 4     2     -10.0
 5     0     -5.0
 5     2     30.0
```

Example 2: SparseMatrix Using the Matrix Market Format

The matrix market exchange format is an ASCII file format that represents sparse matrices in coordinate format. It consists of three sections: The header section is the first line in the file and contains general information about the matrix, e.g. data type and symmetry properties. This line is followed by the comments section which consists of zero or more lines of comments. The remainder of the file is the data section. The first line of the data section contains the row number, column number, and number of nonzeros of the matrix. The following lines contain the location and value of all nonzero entries of the matrix, usually one per line. A file in Matrix Market format is read and converted to a JMSL SparseMatrix in this example. Matrix information and the one norm of the matrix are printed.

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

public class SparseMatrixEx2 {

    public static class MTXReader {

        private String typecode;
        private SparseMatrix matrix;
```

```
    public void read(String filename) throws java.io.IOException {
        InputStream s = SparseMatrixEx2.class.getResourceAsStream(filename);
        BufferedReader br = new BufferedReader(new InputStreamReader(s));

        // read type code initial line
        String line = br.readLine();
        typecode = line;

        // read comment lines if any
        boolean comment = true;
        while (comment) {
            line = br.readLine();
            comment = line.startsWith("%");
        }

        // line now contains the size information which needs to be parsed
        String[] str = line.split("( )+");
        int nRows = Integer.valueOf(str[0].trim());
        int nColumns = Integer.valueOf(str[1].trim());

        // now we're into the data section
        matrix = new SparseMatrix(nRows, nColumns);
        while (true) {
            line = br.readLine();
            if (line == null) {
                break;
            }
            str = line.split("( )+");
            int i = Integer.valueOf(str[0].trim());
            int j = Integer.valueOf(str[1].trim());
            double x = Double.valueOf(str[2].trim());
            matrix.set(i - 1, j - 1, x);
        }
        br.close();
        s.close();
    }

    public String getTypeCode() {
        return this.typecode;
    }
}

public static void main(String args[]) throws Exception {
    MTXReader mr = new MTXReader();
    mr.read("bcsstk01.mtx");
    SparseMatrix A = mr.matrix;

    // Print the matrix type
    System.out.println("The matrix type is " + mr.getTypeCode());

    // Print the matrix information and its one norm
    System.out.println("The number of rows is " + A.getNumberOfRows());
    int ncols = A.getNumberOfColumns();
    System.out.println("The number of columns is " + ncols);
    long nnz = A.getNumberOfNonZeros();
```

```
        System.out.println("The number of nonzero elements is " + nnz);
        System.out.println();
        System.out.println("The 1 norm of the matrix is " + A.oneNorm());
    }
}
```

Output

```
The matrix type is %%MatrixMarket matrix coordinate real symmetric
The number of rows is 48
The number of columns is 48
The number of nonzero elements is 224

The 1 norm of the matrix is 3.00944444444744E9
```

SparseMatrix.SparseArray class

`static public class com.imsl.math.SparseMatrix.SparseArray`

The `SparseArray` class uses public fields to hold the data for a sparse matrix in the Java Sparse Array format. This format came about as a means for storing sparse matrices in Java. In this format, a sparse matrix is represented by two arrays of arrays. One array contains the references to the nonzero value arrays for each row of the sparse matrix. The other contains the references to the associated column index arrays.

As an example, consider the following real sparse matrix:

$$A = \begin{pmatrix} 10.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 10.0 & -3.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 15.0 & 0.0 & 0.0 & 0.0 \\ -2.0 & 0.0 & 0.0 & 10.0 & -1.0 & 0.0 \\ -1.0 & 0.0 & 0.0 & -5.0 & 1.0 & -3.0 \\ -1.0 & -2.0 & 0.0 & 0.0 & 0.0 & 6.0 \end{pmatrix}$$

In `SparseArray`, this matrix can be represented by the two jagged arrays, `values` and `index`, where `values` refers to the nonzero entries in A and `index` to the column indices:

```
double values[][] = { {10.0}, {10.0, -3.0, -1.0}, {15.0}, {-2.0, 10.0, -1.0},
{-1.0, -5.0, 1.0, -3.0}, {-1.0, -2.0, 6.0} }
int index[][] = { {0}, {1, 2, 3}, {2}, {0, 3, 4}, {0, 3, 4, 5}, {0, 1, 5} }
```

Fields

index

```
public int[][] index
```

Jagged array containing column indices. The length of this array equals `numberOfRows`. The length of each row equals the number of nonzeros in that row of the sparse matrix.

numberOfColumns

```
public int numberOfColumns
```

Number of columns in the matrix.

numberOfNonZeros

```
public long numberOfNonZeros
```

Number of nonzeros in the matrix.

numberOfRows

```
public int numberOfRows
```

Number of rows in the matrix.

values

```
public double[][] values
```

Jagged array containing sparse array values. This array must have the same shape as `index`.

Constructor

SparseMatrix.SparseArray

```
public SparseMatrix.SparseArray()
```

ComplexSparseMatrix class

```
public class com.imsl.math.ComplexSparseMatrix implements Serializable
```

Sparse matrix of type `Complex`. The class represents a general complex sparse matrix. It is intended to be efficiently and easily updated.

A `ComplexSparseMatrix` can be constructed from a set of arrays, or it can be abstractly created as an empty array and then incrementally built into final form. It is usually easier to create an empty `ComplexSparseMatrix` of set size and then use the `set` method to set the elements of the array. When

setting the elements of the sparse array, their positions should be thought of as their positions in the dense array. Elements can be set in any order, but only the elements set are stored.

This class includes methods to update the sparse matrix. There are also methods to multiply a sparse matrix and a vector or to multiply two sparse matrices. To solve a sparse linear system use `ComplexSparseCholesky` or `ComplexSuperLU`.

Constructors

ComplexSparseMatrix

`public ComplexSparseMatrix(ComplexSparseMatrix A)`

Description

Creates a new instance of `ComplexSparseMatrix` which is a copy of another `ComplexSparseMatrix`.

Parameter

`A` – is the `ComplexSparseMatrix` to be copied.

ComplexSparseMatrix

`public ComplexSparseMatrix(ComplexSparseMatrix.SparseArray sparseArray)`

Description

Constructs a complex sparse matrix from a `SparseArray` object.

Parameter

`sparseArray` – is a `SparseArray` used to initialize the sparse matrix. The field `numberOfNonZeros` in `SparseArray` is not used for initialization, so it does not have to be set.

ComplexSparseMatrix

`public ComplexSparseMatrix(int nRows, int nColumns)`

Description

Creates a new instance of `ComplexSparseMatrix`. Initially this is the zero matrix.

Parameters

`nRows` – an `int` containing the number of rows in the sparse matrix.

`nColumns` – an `int` containing the number of columns in the sparse matrix.

ComplexSparseMatrix

`public ComplexSparseMatrix(int nRows, int nColumns, int[][] index, Complex[][] values)`

Description

Constructs a sparse matrix from SparseArray (Java Sparse Array) data.

Parameters

`nRows` – an `int` containing the number of rows in the sparse matrix.

`nColumns` – an `int` containing the number of columns in the sparse matrix.

`index` – an `int` jagged array containing the column indices of all nonzero elements corresponding to the compressed representation of the sparse matrix in `values`. The size of `index` must be identical to the size of `values`. The *i*-th row contains the column indices of all nonzero elements of row *i* of the sparse matrix. The *j*-th element of row *i* is the column index of the value located at the same position in `values`.

`values` – a `Complex` jagged array containing the compressed representation of a complex sparse matrix of size `nRows` by `nColumns`. The number of rows in `values` must be `nRows`. The *i*-th row contains all nonzero elements of row *i* of the full sparse matrix.

Methods

add

```
static public ComplexSparseMatrix add(Complex alpha, Complex beta,
ComplexSparseMatrix A, ComplexSparseMatrix B)
```

Description

Performs element-wise addition of two complex sparse matrices `A`, `B` of type `ComplexSparseMatrix`, $C \leftarrow \alpha A + \beta B$.

Parameters

`alpha` – a `Complex` scalar.

`beta` – a `Complex` scalar.

`A` – a `ComplexSparseMatrix` matrix.

`B` – a `ComplexSparseMatrix` matrix.

Returns

a `ComplexSparseMatrix` matrix representing the computed sum.

Exception

`IllegalArgumentException` This exception is thrown if the matrices are not of the same size.

checkSquareMatrix

```
public void checkSquareMatrix()
```

Description

Check that the matrix is square.

Exception

`IllegalArgumentException` is thrown if the matrix is not square.

conjugateTranspose

```
public ComplexSparseMatrix conjugateTranspose()
```

Description

Returns the conjugate transpose of the matrix.

Returns

a `ComplexSparseMatrix` object which is the conjugate transpose of the constructed `ComplexSparseMatrix` object.

frobeniusNorm

```
public double frobeniusNorm()
```

Description

Returns the Frobenius norm of the matrix.

Returns

a `double` scalar value equal to the Frobenius norm of the matrix.

get

```
public Complex get(int iRow, int jColumn)
```

Description

Returns the value of an element in the matrix.

Parameters

`iRow` – an `int` containing the row index of the element.

`jColumn` – an `int` containing the column index of the element.

Returns

a `Complex` containing the value of the `iRow`-th and `jColumn`-th element. If the element was never set, its value is zero.

getNumberOfColumns

```
public int getNumberOfColumns()
```

Description

Returns the number of columns in the matrix.

Returns

an `int` containing the number of columns in the matrix.

getNumberOfNonZeros

```
public long getNumberOfNonZeros()
```

Description

Returns the number of nonzeros in the matrix.

Returns

a `long` containing the number of nonzeros in the matrix.

getNumberOfRows

```
public int getNumberOfRows()
```

Description

Returns the number of rows in the matrix.

Returns

an `int` containing the number of rows in the matrix.

infinityNorm

```
public double infinityNorm()
```

Description

Returns the infinity norm of the matrix.

Returns

a `double` scalar value equal to the maximum of the row sums of the absolute values of the array elements of the sparse matrix.

multiply

```
public Complex[] multiply(Complex[] x)
```

Description

Multiply the matrix by a vector.

Parameter

`x` – a `Complex` column array.

Returns

a `Complex` vector representing the product of this matrix times x.

Exception

`IllegalArgumentException` This exception is thrown if the number of columns in the `ComplexSparseMatrix` object is not equal to the number of elements in the input column vector.

multiply

```
static public Complex[] multiply(Complex[] x, ComplexSparseMatrix A)
```

Description

Multiply row array `x` and sparse matrix `A`, $x^T A$.

Parameters

x – a Complex row array.

A – a ComplexSparseMatrix matrix.

Returns

a Complex vector representing the product of the arguments, $x^T A$.

Exception

IllegalArgumentException This exception is thrown when the number of elements in the input vector is not equal to the number of rows of the matrix.

---

multiply

```
static public Complex[] multiply(ComplexSparseMatrix A, Complex[] x)
```

Description

Multiply sparse matrix A and column array x, Ax.

Parameters

A – a ComplexSparseMatrix matrix.

x – a Complex column array.

Returns

a Complex vector representing the product of the arguments, Ax.

Exception

IllegalArgumentException This exception is thrown when the number of columns in the input matrix is not equal to the number of elements in the input column vector.

---

multiply

```
static public ComplexSparseMatrix multiply(ComplexSparseMatrix A,
ComplexSparseMatrix B)
```

Description

Multiply two sparse complex matrices A and B, $C \leftarrow AB$.

Parameters

A – a ComplexSparseMatrix sparse matrix.

B – a ComplexSparseMatrix sparse matrix.

Returns

the ComplexSparseMatrix product AB of A and B.

Exception

IllegalArgumentException This exception is thrown when the column number of matrix A is not equal to the row number of matrix B.

---

multiplyHermitian

```
static public Complex[] multiplyHermitian(ComplexSparseMatrix A, Complex[] x)
```

Description

Multiply sparse Hermitian matrix `A` and column vector `x`.

Parameters

`A` – a `ComplexSparseMatrix` sparse Hermitian matrix, where only the lower triangular part of the matrix is to be used.

`x` – A `Complex` vector.

Returns

a `Complex` vector representing the product of the arguments, Ax.

Exception

`IllegalArgumentException` This exception is thrown when the input matrix is not square or the number of columns in the input matrix is not equal to the number of elements in the input column vector.

oneNorm

```
public double oneNorm()
```

Description

Returns the matrix one norm of the sparse matrix.

Returns

a `double` value equal to the maximum of the column sums of the absolute values of the array elements.

plusEquals

```
public Complex plusEquals(int iRow, int jColumn, Complex x)
```

Description

Adds a value to an element in the matrix.

Parameters

`iRow` – an `int` containing the row index of the element.

`jColumn` – an `int` containing the column index of the element.

`x` – a `Complex` containing the value to be added.

Returns

a `Complex` containing the updated value of the element, which equals its old value plus x.

set

```
public void set(int iRow, int jColumn, Complex x)
```

Description

Sets the value of an element in the matrix.

Parameters

`iRow` – an `int` containing the row index of the element.

`jColumn` – an `int` containing the column index of the element.

`x` – a `Complex` containing the value of the `iRow`-th and `jColumn`-th element.

toDenseMatrix

```
public Complex[][] toDenseMatrix()
```

Description

Returns the sparse matrix as a dense matrix.

Returns

a rectangular Java array of type `Complex` containing this matrix with all of the zeros explicitly present. The number of rows and columns in the returned matrix is the same as in the sparse matrix.

toSparseArray

```
public ComplexSparseMatrix.SparseArray toSparseArray()
```

Description

Returns the sparse matrix in the `SparseArray` form.

Example: ComplexSparseMatrix

The matrix product of two complex sparse matrices is computed using a method from the `ComplexSparseMatrix` class. The one-norm of the result is also computed. The matrix and its norm are printed.

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

public class ComplexSparseMatrixEx1 {

    public static void main(String args[]) {
        ComplexSparseMatrix b = new ComplexSparseMatrix(6, 6);
        b.set(0, 0, new Complex(10.0, -3.0));
        b.set(1, 1, new Complex(10.0, 0.0));
        b.set(1, 2, new Complex(-3.0, 2.0));
        b.set(1, 3, new Complex(-1.0, -1.0));
        b.set(2, 2, new Complex(15.0, 5.0));
        b.set(3, 0, new Complex(-2.0, 0.0));
        b.set(3, 3, new Complex(10.0, 1.0));
        b.set(3, 4, new Complex(-1.0, -2.0));
        b.set(4, 0, new Complex(-1.0, 0.0));
        b.set(4, 3, new Complex(-5.0, 7.0));
        b.set(4, 4, new Complex(1.0, -3.0));
        b.set(4, 5, new Complex(-3.0, 0.0));
        b.set(5, 0, new Complex(-1.0, 4.0));
        b.set(5, 1, new Complex(-2.0, 1.0));
        b.set(5, 5, new Complex(6.0, -5.0));

        ComplexSparseMatrix c = new ComplexSparseMatrix(6, 3);
```

```
        c.set(0, 0, new Complex(5.0, 0.0));
        c.set(1, 2, new Complex(-3.0, -4.0));
        c.set(2, 0, new Complex(1.0, 2.0));
        c.set(2, 2, new Complex(7.0, 1.0));
        c.set(3, 0, new Complex(2.0, -7.0));
        c.set(4, 1, new Complex(-5.0, 2.0));
        c.set(4, 2, new Complex(2.0, 1.0));
        c.set(5, 2, new Complex(4.0, 0.0));

        // A = b * c
        ComplexSparseMatrix A = ComplexSparseMatrix.multiply(b, c);

        // Get the one norm of matrix A
        System.out.println("The 1-norm of the matrix is " + A.oneNorm());

        ComplexSparseMatrix.SparseArray sa = A.toSparseArray();

        System.out.println("row  column  value");
        for (int i = 0; i < sa.numberOfRows; i++) {
            for (int j = 0; j < sa.index[i].length; j++) {
                System.out.println(" " + i + "     " + sa.index[i][j]
                        + "    ("
                        + Complex.real(sa.values[i][j]) + ","
                        + Complex.imag(sa.values[i][j]) + ")");
            }
        }
    }
}
```

Output

```
The 1-norm of the matrix is 253.9370051143438
row  column  value
 0     0     (50.0,-15.0)
 1     0     (-16.0,1.0)
 1     2     (-53.0,-29.0)
 2     0     (5.0,35.0)
 2     2     (100.0,50.0)
 3     0     (17.0,-68.0)
 3     1     (9.0,8.0)
 3     2     (0.0,-5.0)
 4     0     (34.0,49.0)
 4     1     (1.0,17.0)
 4     2     (-7.0,-5.0)
 5     0     (-5.0,20.0)
 5     2     (34.0,-15.0)
```

ComplexSparseMatrix.SparseArray class

static public class com.imsl.math.ComplexSparseMatrix.SparseArray implements Serializable

The SparseArray class uses public fields to hold the data for a sparse matrix in the Java Sparse Array format. This format came about as a means for storing sparse matrices in Java. In this format, a sparse matrix is represented by two arrays of arrays. One array contains the references to the nonzero value arrays for each row of the sparse matrix. The other contains the references to the associated column index arrays.

As an example, consider the following complex sparse matrix:

$$A = \begin{pmatrix} 10.0-3.0i & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 10.0 & -3.0+2.0i & -1.0-1.0i & 0.0 & 0.0 \\ 0.0 & 0.0 & 15.0+5.0i & 0.0 & 0.0 & 0.0 \\ -2.0 & 0.0 & 0.0 & 10.0+1.0i & -1.0-2.0i & 0.0 \\ -1.0 & 0.0 & 0.0 & -5.0+7.0i & 1.0-3.0i & -3.0 \\ -1.0+4.0i & -2.0+1.0i & 0.0 & 0.0 & 0.0 & 6.0-5.0i \end{pmatrix}$$

In SparseArray, this matrix can be represented by the two jagged arrays, values and index, where values refers to the nonzero entries in A and index to the column indices:

```
      Complex values[][] = {
{new Complex(10.0, -3.0)},
{new Complex(10.0, 0.0), new Complex(-3.0, 2.0), new Complex(-1.0,-1.0)},
{new Complex(15.0, 5.0)},
{new Complex(-2.0, 0.0), new Complex(10.0, 1.0), new Complex(-1.0, -2.0)},
{new Complex(-1.0, 0.0), new Complex(-5.0, 7.0), new Complex(1.0, -3.0),
           new Complex(-3.0, 0.0)},
          {new Complex(-1.0, 4.0), new Complex(-2.0, 1.0), new Complex(6.0, -5.0)}
}

int index[][] = {
          {0},
          {1, 2, 3},
          {2},
          {0, 3, 4},
          {0, 3, 4, 5},
          {0, 1, 5}
      }
```

Fields

index

public int[][] index

Jagged array containing column indices. The length of this array equals `numberOfRows`. The length of each row equals the number of nonzeros in that row of the sparse matrix.

numberOfColumns

```
public int numberOfColumns
```

Number of columns in the matrix.

numberOfNonZeros

```
public long numberOfNonZeros
```

Number of nonzeros in the matrix.

numberOfRows

```
public int numberOfRows
```

Number of rows in the matrix.

values

```
public Complex[][] values
```

Jagged array containing sparse array values. This array must have the same shape as `index`.

Constructor

ComplexSparseMatrix.SparseArray

```
public ComplexSparseMatrix.SparseArray()
```

LU class

```
public class com.imsl.math.LU implements Serializable, Cloneable
```

LU factorization of a matrix of type `double`.

`LU` performs an *LU* factorization of a real general coefficient matrix. The `condition` method estimates the reciprocal of the L_1 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).

Note that A is not retained for use by other methods of this class, only the factorization of A is retained. Thus, A is a required parameter to the `condition` method.

An estimated condition number greater than $1/\varepsilon$ (where ε 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. If there is conern about the input matrix being ill-conditioned, the user of this class should check the condition number of the input matrix using the `condition` method before using one of the other class methods.

`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.

Fields

factor

`protected double[][] factor`

This is an n by n matrix containing the LU factorization of the matrix A.

ipvt

`protected int[] ipvt`

Vector of length n containing the pivot sequence for the factorization.

Constructor

LU

`public LU(double[][] a) throws SingularMatrixException`

Description

Creates the LU factorization of a square matrix of type `double`.

Parameter

`a` – the `double` square matrix to be factored

Exceptions

`IllegalArgumentException` is thrown when the row lengths of input matrix are not equal (for example, the matrix edges are “jagged”.)

`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 L_1 condition number of a matrix.

Parameter

a – the double square matrix for which the reciprocal of the L_1 condition number is desired

Returns

a double value representing an estimate of the reciprocal of the L_1 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

getL

```
public double[][] getL()
```

Description

Returns the lower triangular portion of the LU factorization of A.

Scaled partial pivoting is used to achieve the LU factorization. The resulting factorization is such that $AP = LU$, where A is the input matrix a, P is the permutation matrix returned by getPermutationMatrix, L is the lower triangular matrix returned by getL, and U is the unit upper triangular matrix returned by getU.

Returns

a double matrix containing L, the lower triangular portion of the LU factorization of A.

getPermutationMatrix

```
public double[][] getPermutationMatrix()
```

Description

Returns the permutation matrix which results from the LU factorization of A.

Scaled partial pivoting is used to achieve the LU factorization. The resulting factorization is such that $AP = LU$, where A is the input matrix a, P is the permutation matrix returned by getPermutationMatrix, L is the lower triangular matrix returned by getL, and U is the unit upper triangular matrix returned by getU.

Returns

a `double` matrix containing the permuted identity matrix as a result of the LU factorization of A.

---

getU

```
public double[][] getU()
```

Description

Returns the unit upper triangular portion of the LU factorization of A.

Scaled partial pivoting is used to achieve the LU factorization. The resulting factorization is such that $AP = LU$, where A is the input matrix `a`, P is the permutation matrix returned by `getPermutationMatrix`, L is the lower triangular matrix returned by `getL`, and U is the unit upper triangular matrix returned by `getU`.

Returns

a `double` matrix containing U, the unit upper triangular portion of the LU factorization of A.

---

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.

Parameter

`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

```
static public double[] solve(double[][] a, double[] b) throws
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

Exceptions

`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.

`SingularMatrixException` is thrown when the matrix is singular.

solveTranspose

`public double[] solveTranspose(double[] b)`

Description

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

Parameter

`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. The reciprocal of the condition number of the Matrix is then computed and checked against machine precision to determine whether or not to issue a Warning about the results. A linear system is then solved using the factorization. The inverse and determinant 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);

        // Check the reciprocal of the condition number of
        // A against machine precision
        double condition = lu.condition(a);
        if (condition <= 2.220446049250313e-16) {
            System.out.println("WARNING. The matrix is too ill-conditioned.");
            System.out.println("An estimate of the reciprocal of its L1 "
                    + "condition number is " + condition + ".");
```

```
            System.out.println("Results based on this factorization "
                    + "may not be accurate.");
        }

        // 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);

        // Print the condition number of 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
```

SuperLU class

`public class com.imsl.math.SuperLU implements Serializable`

Computes the LU factorization of a general sparse matrix of type `SparseMatrix` by a column method and solves the real sparse linear system of equations $Ax = b$.

Consider the sparse linear system of equations

$$Ax = b.$$

Here, A is a general square, nonsingular, n by n sparse matrix, and x and b are vectors of length n. All entries in A, x and b are of type `double`.

Gaussian elimination, applied to the system above, can be shortly described as follows:
1. Compute a triangular factorization $P_r D_r A D_c P_c = LU$. Here, D_r and D_c are positive definite diagonal matrices to equilibrate the system and P_r and P_c are permutation matrices to ensure numerical stability and preserve sparsity. L is a unit lower triangular matrix and U is an upper triangular matrix.
2. Solve $Ax = b$ by evaluating

$$x = A^{-1}b = D_c(P_c(U^{-1}(L^{-1}(P_r(D_r b))))) .$$

This is done efficiently by multiplying from right to left in the last expression: Scale the rows of b by D_r. Multiplying $P_r(D_r b)$ means permuting the rows of $D_r b$. Multiplying $L^{-1}(P_r D_r b)$ means solving the triangular system of equations with matrix L by substitution. Similarly, multiplying $U^{-1}(L^{-1}(P_r D_r b))$ means solving the triangular system with U.

Class `SuperLU` handles step 1 above in the `solve` method if it has not been computed prior to step 2. More precisely, before $Ax = b$ is solved the following steps are performed:

1. Equilibrate matrix A, i.e. compute diagonal matrices D_r and D_c so that $\hat{A} = D_r A D_c$ is "better conditioned" than A, i.e. $\hat{A}^{-1}$ is less sensitive to perturbations in $\hat{A}$ than A^{-1} is to perturbations in A.
2. Order the columns of $\hat{A}$ to increase the sparsity of the computed L and U factors, i.e. replace $\hat{A}$ by $\hat{A}P_c$ where P_c is a column permutation matrix.
3. Compute the LU factorization of $\hat{A}P_c$. For numerical stability, the rows of $\hat{A}P_c$ are eventually permuted through the factorization process by scaled partial pivoting, leading to the decomposition $\tilde{A} := P_r \hat{A} P_c = LU$. The LU factorization is done by a left looking supernode-panel algorithm with 2-D blocking. See Demmel, Eisenstat, Gilbert et al. (1999) for further information on this technique.
4. Compute the reciprocal pivot growth factor

$$\max_{1 \le j \le n} \frac{\|\tilde{A}_j\|_\infty}{\|U_j\|_\infty}$$

   where $\tilde{A}_j$ and U_j denote the j-th column of matrices $\tilde{A}$ and U, respectively.
5. Estimate the reciprocal of the condition number of matrix $\tilde{A}$.

Method `solve` uses this information to perform the following steps:

1. Solve the system $Ax = b$ using the computed triangular factors.
2. Iteratively refine the solution, again using the computed triangular factors. This is equivalent to Newton's method.

3. Compute forward and backward error bounds for the solution vector x.

Some of the steps mentioned above are optional. Their settings can be controlled by the set methods of class `SuperLU`.

Class `SuperLU` is based on the SuperLU code written by Demmel, Gilbert, Li et al. For more detailed explanations of the factorization and solve steps, see the SuperLU Users' Guide (1999).

Copyright (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy)

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

(3) Neither the name of Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Fields

COLUMN_APPROXIMATE_MINIMUM_DEGREE

```
static final public int COLUMN_APPROXIMATE_MINIMUM_DEGREE
```

For column ordering, use column approximate minimum degree ordering.

COLUMN_SCALING

```
static final public int COLUMN_SCALING
```

Indicates that input matrix A was column scaled before factorization. This is a return value for

getEquilibrationMethod.

FILL_FACTOR

```
static final public int FILL_FACTOR
```

A performance tuning parameter which can be adjusted via method setPerformanceTuningParameters.

MAXIMUM_SUPERNODE_SIZE

```
static final public int MAXIMUM_SUPERNODE_SIZE
```

A performance tuning parameter which can be adjusted via method setPerformanceTuningParameters.

MINIMUM_COLUMN_DIMENSION

```
static final public int MINIMUM_COLUMN_DIMENSION
```

A performance tuning parameter which can be adjusted via method setPerformanceTuningParameters.

MINIMUM_DEGREE_AT_A

```
static final public int MINIMUM_DEGREE_AT_A
```

For column ordering, use minimum degree ordering on the structure of $A^T A$.

MINIMUM_DEGREE_AT_PLUS_A

```
static final public int MINIMUM_DEGREE_AT_PLUS_A
```

For column ordering, use minimum degree ordering on the structure of $A^T + A$.

MINIMUM_ROW_DIMENSION

```
static final public int MINIMUM_ROW_DIMENSION
```

A performance tuning parameter which can be adjusted via method setPerformanceTuningParameters.

NATURAL_ORDERING

```
static final public int NATURAL_ORDERING
```

For column ordering, use the natural ordering.

NO_SCALING

```
static final public int NO_SCALING
```

Indicates that input matrix A was not equilibrated before factorization. This is a return value for getEquilibrationMethod.

PANEL_SIZE

```
static final public int PANEL_SIZE
```

A performance tuning parameter which can be adjusted via method setPerformanceTuningParameters.

RELAXATION_PARAMETER
`static final public int RELAXATION_PARAMETER`

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

ROW_AND_COLUMN_SCALING
`static final public int ROW_AND_COLUMN_SCALING`

Indicates that input matrix A was row and column scaled before factorization. This is a return value for `getEquilibrationMethod`.

ROW_SCALING
`static final public int ROW_SCALING`

Indicates that input matrix A was row scaled before factorization. This is a return value for `getEquilibrationMethod`.

Constructor

SuperLU
`public SuperLU(SparseMatrix A)`

Description
Constructor for `SuperLU`.

Parameter
`A` – a `SparseMatrix` containing the sparse quadratic input matrix.

Methods

getColumnPermutationMethod
`public int getColumnPermutationMethod()`

Description
Returns the method that will be used to permute the columns of the input matrix.

Returns
an `int` scalar specifying how the columns of the input matrix are to be permuted for sparsity preservation.

return value	*method*
0 = NATURAL_ORDERING	natural ordering, that is $P_c = I$, I the identity matrix.
1 = MINIMUM_DEGREE_AT_PLUS_A	minimum degree ordering on the structure of $A^T + A$
2 = MINIMUM_DEGREE_AT_A	minimum degree ordering on the structure of $A^T A$
3 = COLUMN_APPROXIMATE_MINIMUM_DEGREE	column approximate minimum degree ordering

getConditionNumber

`public double getConditionNumber() throws SingularMatrixException`

Description

Returns the estimate of the reciprocal condition number of the matrix A.

Returns

a `double` scalar containing the reciprocal condition number of the matrix A after equilibration and permutation of rows/columns (if done). If the return value is less than machine precision (in particular, if the return value `= 0`), the matrix is singular to working precision.

getDiagonalPivotThreshold

`public double getDiagonalPivotThreshold()`

Description

Returns the threshold used for a diagonal entry to be an acceptable pivot.

Returns

a `double` scalar specifying the threshold used for a diagonal entry to be an acceptable pivot.

getEquilibrate

`public boolean getEquilibrate()`

Description

Returns the equilibration flag.

Returns

a `boolean` specifying whether or not matrix `A` is equilibrated before factorization. if `getEquilibrate` returns `true` the system is equilibrated, if `getEquilibrate` returns `false`, no equilibration is performed.

getEquilibrationMethod

`public int getEquilibrationMethod()`

Description

Returns information on the type of equilibration used before matrix factorization.

Returns

an `int` value specifying the equilibration option used.

return value	*Description*
1 = `NO_SCALING`	No equilibration is performed.
2 = `ROW_SCALING`	Equilibration is performed with row scaling.
3 = `COLUMN_SCALING`	Equilibration is performed with column scaling.
4 = `ROW_AND_COLUMN_SCALING`	Equilibration is performed with row and column scaling.

getForwardErrorBound

`public double getForwardErrorBound()`

Description

Returns the estimated forward error bound for the solution vector.

Returns

a `double` containing the estimated forward error bound for the solution vector. The estimate is as reliable as the estimate for the reciprocal condition number, and is almost always a slight overestimate of the true error. If iterative refinement is not used, the return value `= 1.0`.

getIterativeRefinement

`public boolean getIterativeRefinement()`

Description

Returns a value specifying whether iterative refinement is to be performed or not.

Returns

a `boolean` scalar specifying whether iterative refinement is to be performed, `true`, or no iterative refinement is to be performed, `false`.

getPerformanceTuningParameters

`public int getPerformanceTuningParameters(int parameter)`

Description

Returns a performance tuning parameter value.

Parameter

`parameter` – an `int` scalar that specifies the parameter for which the value is to be returned.

parameter	*return value description*
PANEL_SIZE	The panel size.
RELAXATION_PARAMETER	The relaxation parameter to control supernode amalgamation.
MAXIMUM_SUPERNODE_SIZE	The maximum allowable size for a supernode.
MINIMUM_ROW_DIMENSIONM	The minimum row dimension to be used for 2D blocking.
MINIMUM_COLUMN_DIMENSION	The minimum column dimension to be used for 2D blocking.
FILL_FACTOR	The estimated fill factor for L and U, compared with A.

Returns

an `int` specifying the current value used for the specified tuning parameter.

getPivotGrowth

```
public boolean getPivotGrowth()
```

Description

Returns the reciprocal pivot growth factor flag.

Returns

a `boolean` specifying whether to compute the reciprocal pivot growth factor. Returns true if the reciprocal pivot growth factor is to be computed.

getReciprocalPivotGrowthFactor

```
public double getReciprocalPivotGrowthFactor() throws SingularMatrixException
```

Description

Returns the reciprocal pivot growth factor.

Returns

a `double` scalar representing the reciprocal growth factor

$$\max_{j \in \{1,\ldots,n\}} \frac{||\tilde{A}_j||_\infty}{||U_j||_\infty}.$$

If the returned value is much less than 1, the stability of the LU factorization could be poor.

getRelativeBackwardError

```
public double getRelativeBackwardError()
```

Description

Returns the componentwise relative backward error of the solution vector.

Returns

a `double` containing the componentwise relative backward error of the solution vector `x`. If iterative refinement is not used, the return value = `1.0`.

getSymmetricMode

```
public boolean getSymmetricMode()
```

Description

Returns the symmetric mode flag.

Returns

a `boolean` scalar indicating if symmetric mode is to be used. Returns true if symmetric mode is to be used.

setColumnPermutationMethod

```
public void setColumnPermutationMethod(int colpermute)
```

Description

Specifies how to permute the columns of the input matrix.

Parameter

`colpermute` – an `int` scalar specifying how to permute the columns of the input matrix for sparsity preservation.

colpermute	*method*
`NATURAL_ORDERING`	natural ordering, that is $P_c = I$, I the identity matrix.
`MINIMUM_DEGREE_AT_PLUS_A`	minimum degree ordering on the structure of $A^T + A$
`MINIMUM_DEGREE_AT_A`	minimum degree ordering on the structure of $A^T A$
`COLUMN_APPROXIMATE_MINIMUM_DEGREE`	column approximate minimum degree ordering

If this method is not called, `colpermute` is set to `SuperLU.COLUMN_APPROXIMATE_MINIMUM_DEGREE`.

Exception

`IllegalArgumentException` is thrown when `colpermute` is not one of the above values.

setDiagonalPivotThreshold

```
public void setDiagonalPivotThreshold(double thresh)
```

Description

Specifies the threshold used for a diagonal entry to be an acceptable pivot.

Parameter

`thresh` – a `double` scalar specifying the threshold used for a diagonal entry to be an acceptable pivot.
Default: `thresh=1.0`, i.e. classical partial pivoting.

Exception

`IllegalArgumentException` is thrown when `thresh` is not in the interval $[0.0, 1.0]$.

setEquilibrate

```
public void setEquilibrate(boolean equilibrate)
```

Description

Determines if input matrix *A* should be equilibrated before factorization.

Parameter

`equilibrate` – a `boolean` determining if matrix *A* should be equilibrated before the factorization.

equilibrate	*action*
`false`	do not equilibrate
`true`	equilibrate

If this method is not called, `equilibrate` is set to true.

setIterativeRefinement

```
public void setIterativeRefinement(boolean refine)
```

Description

Specifies whether to perform iterative refinement.

Parameter

`refine` – a `boolean` specifying whether to use iterative refinement, `refine = true`, or no iterative refinement, `refine = false`.
Default: `refine = false`.

setPerformanceTuningParameters

```
public void setPerformanceTuningParameters(int parameter, int tunedValue)
```

Description

Sets performance tuning parameters.

Parameters

`parameter` – an `int` scalar that specifies the parameter to be tuned.

`tunedValue` – an `int` scalar that specifies the value to be used for the specified tuning parameter.

parameter	*Description*	*Default*
PANEL_SIZE	The panel size.	10
RELAXATION_PARAMETER	The relaxation parameter to control supernode amalgamation.	5
MAXIMUM_SUPERNODE_SIZE	The maximum allowable size for a supernode.	100
MINIMUM_ROW_DIMENSION	The minimum row dimension to be used for 2D blocking.	200
MINIMUM_COLUMN_DIMENSION	The minimum column dimension to be used for 2D blocking.	40
FILL_FACTOR	The estimated fill factor for L and U, compared with A.	20

Exception

`IllegalArgumentException` is thrown when a) `parameter` is not in the interval $[1,\ldots,6]$ or b) `tunedValue` is not greater than zero.

setPivotGrowth

```
public void setPivotGrowth(boolean growth)
```

Description

Specifies whether to compute the reciprocal pivot growth factor.

Parameter

`growth` – a `boolean` specifying whether to compute the reciprocal pivot growth factor.

growth	*action*
false	don't compute growth factor
true	compute growth factor

Default: `growth = false`.

setSymmetricMode

```
public void setSymmetricMode(boolean symmetric)
```

Description

Specifies whether to use the symmetric mode.

Parameter

`symmetric` – a `boolean` indicating if symmetric mode is to be used. This mode should be applied if the input matrix A is diagonally dominant or nearly so. The user should then define a small diagonal pivot threshold (e.g. 0.0 or 0.01) by method `setDiagonalPivotThreshold` and choose an $(A^T + A)$-based column permutation algorithm (e.g. column permutation method `SuperLU.MINIMUM_DEGREE_AT_PLUS_A`).

symmetric	*action*
false	symmetric mode is not used
true	symmetric mode is used

Default: `symmetric=false`.

solve

`public double[] solve(double[] b) throws SingularMatrixException`

Description

Computation of the solution vector for the system $Ax = b$.

Parameter

`b` – a `double` vector of length `n`, `n` the order of input matrix `A`, containing the right hand side.

Returns

a `double` vector containing the solution to the system $Ax = b$. Optionally, the solution is improved by iterative refinement, if `setIterativeRefinement` is set to `true`. Method `solve` internally first factorizes matrix A (step 1 of the introduction) if the factorization has not been done before.

solveTranspose

`public double[] solveTranspose(double[] b) throws SingularMatrixException`

Description

Computation of the solution vector for the system $A^T x = b$.

Parameter

`b` – a `double` vector of length `n`, `n` the order of input matrix `A`, containing the right hand side.

Returns

a `double` vector containing the solution to the system $A^T x = b$. Optionally, the solution is improved by iterative refinement, if `setIterativeRefinement` is set to `true` . Method `solveTranspose` internally first factorizes matrix A (step 1 of the introduction) if the factorization has not been done before.

Example: LU Factorization of a Sparse Matrix

The LU Factorization of the sparse 6×6 matrix

$$A = \begin{pmatrix} 10.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 10.0 & -3.0 & -1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 15.0 & 0.0 & 0.0 & 0.0 \\ -2.0 & 0.0 & 0.0 & 10.0 & -1.0 & 0.0 \\ -1.0 & 0.0 & 0.0 & -5.0 & 1.0 & -3.0 \\ -1.0 & -2.0 & 0.0 & 0.0 & 0.0 & 6.0 \end{pmatrix}$$

is computed. The sparse coordinate form for A is given by a series of row, column, value triplets:

row	column	value
0	0	10.0
1	1	10.0
1	2	-3.0
1	3	-1.0
2	2	15.0
3	0	-2.0
3	3	10.0
3	4	-1.0
4	0	-1.0
4	3	-5.0
4	4	1.0
4	5	-3.0
5	0	-1.0
5	1	-2.0
5	5	6.0

Let

$$y^T = (1.0, 2.0, 3.0, 4.0, 5.0, 6.0)$$

so that $b_1 := Ay = (10.0, 7.0, 45.0, 33.0, -34.0, 31.0)^T$, and

$$b_2 := A^T y = (-9.0, 8.0, 39.0, 13.0, 1.0, 21.0)^T$$

.

The LU factorization of A is used to solve the sparse linear systems $Ax = b_1$ and $A^T x = b_2$ with iterative refinement. The reciprocal pivot growth factor and the reciprocal condition number are also computed.

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

public class SuperLUEx1 {

    public static void main(String args[]) throws Exception {
        int m;
        SuperLU slu;
        double conditionNumber, recip_pivot_growth;
        double[] sol = null;
        double Ferr, Berr;

        double[] b1 = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
        double[] b2 = {-9.0, 8.0, 39.0, 13.0, 1.0, 21.0};

        // Initialize matrix A.
        m = 6;
        SparseMatrix a = new SparseMatrix(m, m);

        a.set(0, 0, 10.0);
        a.set(1, 1, 10.0);
        a.set(1, 2, -3.0);
        a.set(1, 3, -1.0);
        a.set(2, 2, 15.0);
```

```
a.set(3, 0, -2.0);
a.set(3, 3, 10.0);
a.set(3, 4, -1.0);
a.set(4, 0, -1.0);
a.set(4, 3, -5.0);
a.set(4, 4, 1.0);
a.set(4, 5, -3.0);
a.set(5, 0, -1.0);
a.set(5, 1, -2.0);
a.set(5, 5, 6.0);

// Compute the sparse LU factorization of a
slu = new SuperLU(a);

slu.setEquilibrate(false);
slu.setColumnPermutationMethod(SuperLU.NATURAL_ORDERING);
slu.setPivotGrowth(true);

//slu.setPerformanceTuningParameters(1,2);
//slu.setPerformanceTuningParameters(2,1);
// Set option of iterative refinement
slu.setIterativeRefinement(true);

// Solve sparse system A*x = b1
System.out.println();
System.out.println("Solve sparse System Ax=b1");
System.out.println("========================");
System.out.println();

sol = slu.solve(b1);
new PrintMatrix("Solution").print(sol);

Ferr = slu.getForwardErrorBound();
Berr = slu.getRelativeBackwardError();

System.out.println();
System.out.println("Forward error bound: " + Ferr);
System.out.println("Relative backward error: " + Berr);
System.out.println();
System.out.println();

// Solve sparse system A^T*x = b2
System.out.println();
System.out.println("Solve sparse System A^Tx=b2");
System.out.println("==========================");
System.out.println();

sol = slu.solveTranspose(b2);
new PrintMatrix("Solution").print(sol);

Ferr = slu.getForwardErrorBound();
Berr = slu.getRelativeBackwardError();

System.out.println();
System.out.println("Forward error bound: " + Ferr);
System.out.println("Relative backward error: " + Berr);
```

```
        System.out.println();
        System.out.println();

        // Compute reciprocal pivot growth factor and condition number
        recip_pivot_growth = slu.getReciprocalPivotGrowthFactor();
        conditionNumber = slu.getConditionNumber();

        System.out.println("Pivot growth factor and condition number");
        System.out.println("========================================");
        System.out.println();

        System.out.println("Reciprocal pivot growth factor: "
                + recip_pivot_growth);
        System.out.println("Reciprocal condition number: " + conditionNumber);
        System.out.println();
    }
}
```

Output

```
Solve sparse System Ax=b1
=========================

Solution
   0
0  1
1  2
2  3
3  4
4  5
5  6


Forward error bound: 2.7448942589780064E-14
Relative backward error: 0.0



Solve sparse System A^Tx=b2
===========================

Solution
   0
0  1
1  2
2  3
3  4
4  5
5  6


Forward error bound: 2.413791011361905E-15
Relative backward error: 0.0
```

```
Pivot growth factor and condition number
========================================

Reciprocal pivot growth factor: 1.0
Reciprocal condition number: 0.02445109780439122
```

ComplexLU class

`public class com.imsl.math.ComplexLU implements Serializable, Cloneable`

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 reciprocal of the L_1 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).

Note that A is not retained for use by other methods of this class, only the factorization of A is retained. Thus, A is a required parameter to the `condition` method.

An estimated condition number greater than $1/\varepsilon$ (where ε 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. If there is conern about the input matrix being ill-conditioned, the user of this class should check the condition number of the input matrix using the `condition` method before using one of the other class methods.

`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.

Fields

factor

`protected Complex[][] factor`

This is an *n* by *n* `Complex` matrix containing the *LU* factorization of the matrix *A*.

ipvt

`protected int[] ipvt`

Vector of length *n* containing the pivot sequence for the factorization.

Constructor

ComplexLU

`public ComplexLU(Complex[][] a) throws SingularMatrixException`

Description

Creates the *LU* factorization of a square matrix of type `Complex`.

Parameter

`a` – `Complex` square matrix to be factored

Exceptions

`IllegalArgumentException` is thrown when the row lengths of input matrix are not equal (for example, the matrix edges are "jagged".)

`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 L_1 condition number.

Parameter

`a` – a `Complex` matrix

Returns

a `double` scalar value representing the estimate of the reciprocal of the L_1 condition number of the matrix A, where A is the parameter `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

getL

```
public Complex[][] getL()
```

Description

Returns the lower triangular portion of the LU factorization of A.

Scaled partial pivoting is used to achieve the LU factorization. The resulting factorization is such that $AP = LU$, where A is the input matrix `a`, P is the permutation matrix returned by `getPermutationMatrix`, L is the lower triangular matrix returned by `getL`, and U is the unit upper triangular matrix returned by `getU`.

Returns

a `Complex` matrix containing L, the lower triangular portion of the LU factorization of A.

getPermutationMatrix

```
public Complex[][] getPermutationMatrix()
```

Description

Returns the permutation matrix which results from the LU factorization of A.

Scaled partial pivoting is used to achieve the LU factorization. The resulting factorization is such that $AP = LU$, where A is the input matrix `a`, P is the permutation matrix returned by `getPermutationMatrix`, L is the lower triangular matrix returned by `getL`, and U is the unit upper triangular matrix returned by `getU`.

Returns

A `Complex` matrix containing the permuted identity matrix as a result of the LU factorization of A.

getU

```
public Complex[][] getU()
```

Description

Returns the unit upper triangular portion of the LU factorization of A.

Scaled partial pivoting is used to achieve the LU factorization. The resulting factorization is such that $AP = LU$, where A is the input matrix `a`, P is the permutation matrix returned by `getPermutationMatrix`, L is the lower triangular matrix returned by `getL`, and U is the unit upper triangular matrix returned by `getU`.

Returns

a `Complex` matrix containing U, the unit upper triangular portion of the LU factorization of A.

inverse

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

Description

Returns the inverse of the matrix used to construct this instance.

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.

Parameter

`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

```
static public Complex[] solve(Complex[][] a, Complex[] b) throws
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.

Exceptions

`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.

`SingularMatrixException` is thrown when the matrix is singular.

solveTranspose

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

Description

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

Parameter

`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 structure is used to convert a real matrix to a Complex matrix. An LU decomposition of the matrix is performed. The reciprocal of the condition number of the Matrix is then computed and checked against machine precision to determine whether or not to issue a Warning about the results. A linear system is then solved using the factorization. The determinant of the input matrix is also computed.

```
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);

        // Check the reciprocal of the condition number of A
        //   against machine precision
        double condition = clu.condition(a);
        if (condition <= 2.220446049250313e-16) {
            System.out.println("WARNING. The matrix is too ill-conditioned.");
            System.out.println("An estimate of the reciprocal of its L1 "
                    + "condition number is " + condition + ".");
            System.out.println("Results based on this factorization may "
                    + "not be accurate.");
        }
```

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

        // Print the condition number of 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
```

ComplexSuperLU class

`public class com.imsl.math.ComplexSuperLU implements Serializable`

Computes the LU factorization of a general sparse matrix of type `ComplexSparseMatrix` by a column method and solves a sparse linear system of equations $Ax = b$.

Consider the sparse linear system of equations

$$Ax = b.$$

Here, A is a general square, nonsingular, n by n sparse matrix, and x and b are vectors of length n. All entries in A, x and b are of type `Complex`.

Gaussian elimination, applied to the system above, can be shortly described as follows:

1. Compute a triangular factorization $P_r D_r A D_c P_c = LU$. Here, D_r and D_c are positive definite diagonal matrices to equilibrate the system and P_r and P_c are permutation matrices to ensure

numerical stability and preserve sparsity. L is a unit lower triangular matrix and U is an upper triangular matrix.

2. Solve $Ax = b$ by evaluating

$$x = A^{-1}b = D_c(P_c(U^{-1}(L^{-1}(P_r(D_r b))))).$$

This is done efficiently by multiplying from right to left in the last expression: Scale the rows of b by D_r. Multiplying $P_r(D_r b)$ means permuting the rows of $D_r b$. Multiplying $L^{-1}(P_r D_r b)$ means solving the triangular system of equations with matrix L by substitution. Similarly, multiplying $U^{-1}(L^{-1}(P_r D_r b))$ means solving the triangular system with U.

Class `ComplexSuperLU` handles step 1 above in the `solve` method if it is has not been computed prior to step 2. More precisely, before $Ax = b$ is solved the following steps are performed:

1. Equilibrate matrix A, i.e. compute diagonal matrices D_r and D_c so that $\hat{A} = D_r A D_c$ is "better conditioned" than A, i.e. $\hat{A}^{-1}$ is less sensitive to perturbations in $\hat{A}$ than A^{-1} is to perturbations in A.

2. Order the columns of $\hat{A}$ to increase the sparsity of the computed L and U factors, i.e. replace $\hat{A}$ by $\hat{A}P_c$ where P_c is a column permutation matrix.

3. Compute the LU factorization of $\hat{A}P_c$. For numerical stability, the rows of $\hat{A}P_c$ are eventually permuted through the factorization process by scaled partial pivoting, leading to the decomposition $\tilde{A} := P_r\hat{A}P_c = LU$. The LU factorization is done by a left looking supernode-panel algorithm with 2-D blocking. See Demmel, Eisenstat, Gilbert et al. (1999) for further information on this technique.

4. Compute the reciprocal pivot growth factor

$$\max_{1 \leq j \leq n} \frac{\|\tilde{A}_j\|_\infty}{\|U_j\|_\infty},$$

where $\tilde{A}_j$ and U_j denote the j-th column of matrices $\tilde{A}$ and U, respectively.

5. Estimate the reciprocal of the condition number of matrix $\tilde{A}$.

Method `solve` uses this information to perform the following steps:

1. Solve the system $Ax = b$ using the computed triangular factors.

2. Iteratively refine the solution, again using the computed triangular factors. This is equivalent to Newton's method.

3. Compute forward and backward error bounds for the solution vector x.

Some of the steps mentioned above are optional. Their settings can be controlled by the set methods of class `ComplexSuperLU`.

Class ComplexSuperLU is based on the SuperLU code written by Demmel, Gilbert, Li et al. For more detailed explanations of the factorization and solve steps, see the SuperLU Users' Guide (1999).

Copyright (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy)

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

(1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

(2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

(3) Neither the name of Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Fields

COLUMN_APPROXIMATE_MINIMUM_DEGREE

```
static final public int COLUMN_APPROXIMATE_MINIMUM_DEGREE
```

For column ordering, use column approximate minimum degree ordering.

COLUMN_SCALING

```
static final public int COLUMN_SCALING
```

Indicates that input matrix A was column scaled before factorization. This is a return value for getEquilibrationMethod.

FILL_FACTOR

```
static final public int FILL_FACTOR
```

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

MAXIMUM_SUPERNODE_SIZE

`static final public int MAXIMUM_SUPERNODE_SIZE`

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

MINIMUM_COLUMN_DIMENSION

`static final public int MINIMUM_COLUMN_DIMENSION`

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

MINIMUM_DEGREE_AT_A

`static final public int MINIMUM_DEGREE_AT_A`

For column ordering, use minimum degree ordering on the structure of $A^T A$.

MINIMUM_DEGREE_AT_PLUS_A

`static final public int MINIMUM_DEGREE_AT_PLUS_A`

For column ordering, use minimum degree ordering on the structure of $A^T + A$.

MINIMUM_ROW_DIMENSION

`static final public int MINIMUM_ROW_DIMENSION`

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

NATURAL_ORDERING

`static final public int NATURAL_ORDERING`

For column ordering, use the natural ordering.

NO_SCALING

`static final public int NO_SCALING`

Indicates that input matrix A was not equilibrated before factorization. This is a return value for `getEquilibrationMethod`.

PANEL_SIZE

`static final public int PANEL_SIZE`

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

RELAXATION_PARAMETER

`static final public int RELAXATION_PARAMETER`

A performance tuning parameter which can be adjusted via method `setPerformanceTuningParameters`.

ROW_AND_COLUMN_SCALING

`static final public int ROW_AND_COLUMN_SCALING`

Indicates that input matrix *A* was row and column scaled before factorization. This is a return value for `getEquilibrationMethod`.

ROW_SCALING

`static final public int ROW_SCALING`

Indicates that input matrix *A* was row scaled before factorization. This is a return value for `getEquilibrationMethod`.

Constructor

ComplexSuperLU

`public ComplexSuperLU(ComplexSparseMatrix A)`

Description

Constructor for `ComplexSuperLU`.

Parameter

`A` – a `ComplexSparseMatrix` containing the sparse quadratic input matrix.

Methods

getColumnPermutationMethod

`public int getColumnPermutationMethod()`

Description

Returns the method that will be used to permute the columns of the input matrix.

Returns

an `int` scalar specifying how the columns of the input matrix are to be permuted for sparsity preservation.

return value	*method*
0 = NATURAL_ORDERING	natural ordering, that is $P_c = I$, I the identity matrix
1 = MINIMUM_DEGREE_AT_PLUS_A	minimum degree ordering on the structure of $A^T + A$
2 = MINIMUM_DEGREE_AT_A	minimum degree ordering on the structure of $A^T A$
3 = COLUMN_APPROXIMATE_MINIMUM_DEGREE	column approximate minimum degree ordering

getConditionNumber

`public double getConditionNumber() throws SingularMatrixException`

Description

Returns the estimate of the reciprocal condition number of the matrix A.

Returns

a `double` scalar containing the reciprocal condition number of the matrix A after equilibration and permutation of rows/columns (if done). If the reciprocal condition number is less than machine precision, in particular if it is equal to 0, the matrix is singular to working precision.

getDiagonalPivotThreshold

`public double getDiagonalPivotThreshold()`

Description

Returns the threshold used for a diagonal entry to be an acceptable pivot.

Returns

a `double` scalar specifying the threshold used for a diagonal entry to be an acceptable pivot.

getEquilibrate

`public boolean getEquilibrate()`

Description

Returns the equilibration flag.

Returns

a `boolean` specifying whether or not matrix `A` is equilibrated before factorization. If `getEquilibrate` returns `true` the system is equilibrated, if `getEquilibrate` returns `false`, no equilibration is performed.

getEquilibrationMethod

`public int getEquilibrationMethod()`

Description

Returns information on the type of equilibration used before matrix factorization.

Returns

an `int` value specifying the equilibration option used.

return value	*option description*
1 = NO_SCALING	No equilibration is performed.
2 = ROW_SCALING	Equilibration is performed with row scaling.
3 = COLUMN_SCALING	Equilibration is performed with column scaling.
4 = ROW_AND_COLUMN_SCALING	Equilibration is performed with row and column scaling.

getForwardErrorBound

`public double getForwardErrorBound()`

Description

Returns the estimated forward error bound for each solution vector.

Returns

a `double` containing the estimated forward error bound for the solution vector. The estimate is as reliable as the estimate for the reciprocal condition number, and is almost always a slight overestimate of the true error. If iterative refinement is not used, the return value = 1.0.

getIterativeRefinement

`public boolean getIterativeRefinement()`

Description

Returns a value specifying whether iterative refinement is to be performed or not.

Returns

a `boolean` scalar specifying whether iterative refinement is to be performed, `true`, or no iterative refinement is to be performed, `false`.

getPerformanceTuningParameters

`public int getPerformanceTuningParameters(int parameter)`

Description

Returns a performance tuning parameter value.

Parameter

`parameter` – an `int` scalar that specifies the parameter for which the value is to be returned.

parameter	*return value description*
PANEL_SIZE	The panel size.
RELAXATION_PARAMETER	The relaxation parameter to control supernode amalgamation.
MAXIMUM_SUPERNODE_SIZE	The maximum allowable size for a supernode.
MINIMUM_ROW_DIMENSION	The minimum row dimension to be used for 2D blocking.
MINIMUM_COLUMN_DIMENSION	The minimum column dimension to be used for 2D blocking.
FILL_FACTOR	The estimated fill factor for L and U, compared with A.

Returns

an `int` specifying the current value used for the specified tuning parameter.

getPivotGrowth

```
public boolean getPivotGrowth()
```

Description

Returns the reciprocal pivot growth factor flag.

Returns

a `boolean` specifying whether to compute the reciprocal pivot growth factor. Returns true if the reciprocal pivot growth factor is to be computed.

getReciprocalPivotGrowthFactor

```
public double getReciprocalPivotGrowthFactor() throws SingularMatrixException
```

Description

Returns the reciprocal pivot growth factor.

Returns

a `double` scalar representing the reciprocal growth factor

$$\max_{1 \le j \le n} \frac{||\tilde{A}_j||_\infty}{||U_j||_\infty}.$$

If the returned value is much less than 1, the stability of the LU factorization could be poor.

getRelativeBackwardError

```
public double getRelativeBackwardError()
```

Description

Returns the componentwise relative backward error of the solution vector.

Returns

a `double` containing the componentwise relative backward error of the solution vector `x`. If `setIterativeRefinement` is not set to `true`, then `getRelativeBackwardError` returns 1.0.

getSymmetricMode

```
public boolean getSymmetricMode()
```

Description

Returns the symmetric mode flag.

Returns

a `boolean` scalar indicating if symmetric mode is to be used. Returns true if symmetric mode is to be used.

setColumnPermutationMethod

```
public void setColumnPermutationMethod(int colpermute)
```

Description

Specifies how to permute the columns of the input matrix.

Parameter

`colpermute` – an `int` scalar specifying how to permute the columns of the input matrix for sparsity preservation.

colpermute	*method*
NATURAL_ORDERING	natural ordering, that is $P_c = I$, I the identity matrix
MINIMUM_DEGREE_AT_PLUS_A	minimum degree ordering on the structure of $A^T + A$
MINIMUM_DEGREE_AT_A	minimum degree ordering on the structure of $A^T A$
COLUMN_APPROXIMATE_MINIMUM_DEGREE	column approximate minimum degree ordering

Default: `colpermute` = `SuperLU.COLUMN_APPROXIMATE_MINIMUM_DEGREE`.

setDiagonalPivotThreshold

```
public void setDiagonalPivotThreshold(double thresh)
```

Description

Specifies the threshold used for a diagonal entry to be an acceptable pivot.

Parameter

`thresh` – a `double` scalar specifying the threshold used for a diagonal entry to be an acceptable pivot.
Default: `thresh=1.0`, i.e. classical partial pivoting.

Exception

`IllegalArgumentException` is thrown if `thresh` is not in the interval $[0.0, 1.0]$.

setEquilibrate

```
public void setEquilibrate(boolean equilibrate)
```

Description

Specifies if input matrix A should be equilibrated before factorization.

Parameter

`equilibrate` – a `boolean` determining if matrix A should be equilibrated before the factorization.

equilibrate	*action*
`false`	do not equilibrate
`true`	equilibrate

Default: `equilibrate` = `true`.

setIterativeRefinement

```
public void setIterativeRefinement(boolean refine)
```

Description

Specifies whether to perform iterative refinement.

Parameter

`refine` – a `boolean` scalar specifying whether to use iterative refinement, `refine = true` or no iterative refinement, `refine = false`.
Default: `refine = false`.

setPerformanceTuningParameters

```
public void setPerformanceTuningParameters(int parameter, int tunedValue)
```

Description

Sets performance tuning parameters.

Parameters

`parameter` – an `int` scalar that specifies the parameter to be tuned.

`tunedValue` – an `int` scalar that specifies the value to be used for the specified tuning parameter.

parameter	*Description*	*Default*
PANEL_SIZE	The panel size.	10
RELAXATION_PARAMETER	The relaxation parameter to control supernode amalgamation.	5
MAXIMUM_SUPERNODE_SIZE	The maximum allowable size for a supernode.	100
MINIMUM_ROW_DIMENSION	The minimum row dimension to be used for 2D blocking.	200
MINIMUM_COLUMN_DIMENSION	The minimum column dimension to be used for 2D blocking.	40
FILL_FACTOR	The estimated fill factor for L and U, compared with A.	20

Exception

`IllegalArgumentException` is thrown when a) `parameter` is not in the interval $[1,\ldots,6]$ or b) `tunedValue` is not greater than zero.

setPivotGrowth

```
public void setPivotGrowth(boolean growth)
```

Description

Specifies whether to compute the reciprocal pivot growth factor.

Parameter

`growth` – a `boolean` specifying whether to compute the reciprocal pivot growth factor.

growth	*action*
`false`	don't compute growth factor
`true`	compute growth factor

Default: `growth = false`.

setSymmetricMode

```
public void setSymmetricMode(boolean symmetric)
```

Description

Specifies whether to use the symmetric mode.

Parameter

`symmetric` – a `boolean` indicating if symmetric mode is to be used. This mode should be applied if the input matrix A is diagonally dominant or nearly so. The user should then define a small diagonal pivot threshold (e.g. 0.0 or 0.01) by method `setDiagonalPivotThreshold` and choose an $(A^T + A)$-based column permutation algorithm (e.g. column permutation method `ComplexSuperLU.MINIMUM_DEGREE_AT_PLUS_A`).

symmetric	*action*
`false`	symmetric mode is not used
`true`	symmetric mode is used

Default: `symmetric=false`.

solve

`public Complex[] solve(Complex[] b) throws SingularMatrixException`

Description

Computation of the solution vector for the system $Ax = b$.

Parameter

`b` – a `Complex` vector of length `n`, `n` the order of input matrix `A`, containing the right hand side.

Returns

a `Complex` vector containing the solution to the system $Ax = b$. Optionally, the solution is improved by iterative refinement, if `setIterativeRefinement` is set to `true`. Method `solve` internally first factorizes matrix A (step 1 of the introduction) if the factorization has not been done before.

solveConjugateTranspose

`public Complex[] solveConjugateTranspose(Complex[] b) throws SingularMatrixException`

Description

Computation of the solution vector for the system $A^H x = b$.

Parameter

`b` – a `Complex` vector of length `n`, `n` the order of input matrix `A`, containing the right hand side.

Returns

a `Complex` vector containing the solution to the system $A^H x = b$. Optionally, the solution is improved by iterative refinement, if `setIterativeRefinement` is set to `true` . Method `solveConjugateTranspose` internally first factorizes matrix A (step 1 of the introduction) if the factorization has not been done before.

solveTranspose

`public Complex[] solveTranspose(Complex[] b) throws SingularMatrixException`

Description

Computation of the solution vector for the system $A^T x = b$.

Parameter

`b` – a `Complex` vector of length `n`, `n` the order of input matrix `A`, containing the right hand side.

Returns

a `Complex` vector containing the solution to the system $A^T x = b$. Optionally, the solution is improved by iterative refinement, if `setIterativeRefinement` is set to `true` . Method `solveTranspose` internally first factorizes matrix A (step 1 of the introduction) if the factorization has not been done before.

Example: LU Factorization of a Complex Sparse Matrix

The LU Factorization of the sparse complex 6×6 matrix

$$A = \begin{pmatrix} 10+7i & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 3+2i & -3+0i & -1+2i & 0.0 & 0.0 \\ 0.0 & 0.0 & 4+2i & 0.0 & 0.0 & 0.0 \\ -2-4i & 0.0 & 0.0 & 1+6i & -1+3i & 0.0 \\ -5+4i & 0.0 & 0.0 & -5+0i & 12+2i & -7+7i \\ -1+12i & -2+8i & 0.0 & 0.0 & 0.0 & 3+7i \end{pmatrix}$$

is computed. The sparse coordinate form for A is given by row, column, value triplets:

row	column	value
0	0	10+7i
1	1	3+2i
1	2	-3+0i
1	3	-1+2i
2	2	4+2i
3	0	-2-4i
3	3	1+6i
3	4	-1+3i
4	0	-5+4i
4	3	-5+0i
4	4	12+2i
4	5	-7+7i
5	0	-1+12i
5	1	-2+8i
5	5	3+7i

Let

$$x^T = (1+i, 2+2i, 3+3i, 4+4i, 5+5i, 6+6i)$$

so that

$$b_1 := Ax = (3+17i, -19+5i, 6+18i, -38+32i, -63+49i, -57+83i)^T$$

and

$$b_2 := A^H x = (54-112i, 46-58i, 12, 5-51i, 78+34i, 60-94i)^T.$$

The LU factorization of A is used to solve the complex sparse linear systems $Ax = b_1$ and $A^H x = b_2$ with iterative refinement. The reciprocal pivot growth factor and the reciprocal condition number are also computed.

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

public class ComplexSuperLUEx1 {

    public static void main(String args[]) throws Exception {
        int m;
```

```
ComplexSuperLU ComplexSparseLU;
double conditionNumber, recip_pivot_growth;
Complex[] sol = null;
double Ferr, Berr;

Complex[] b1 = {
    new Complex(3.0, 17.0), new Complex(-19.0, 5.0),
    new Complex(6.0, 18.0), new Complex(-38.0, 32.0),
    new Complex(-63.0, 49.0), new Complex(-57.0, 83.0)
};

Complex[] b2 = {
    new Complex(54.0, -112.0), new Complex(46.0, -58.0),
    new Complex(12.0, 0.0), new Complex(5.0, -51.0),
    new Complex(78.0, 34.0), new Complex(60.0, -94.0)
};

// Initialize input matrix A.
m = 6;
ComplexSparseMatrix a = new ComplexSparseMatrix(m, m);

a.set(0, 0, new Complex(10.0, 7.0));
a.set(1, 1, new Complex(3.0, 2.0));
a.set(1, 2, new Complex(-3.0, 0.0));
a.set(1, 3, new Complex(-1.0, 2.0));
a.set(2, 2, new Complex(4.0, 2.0));
a.set(3, 0, new Complex(-2.0, -4.0));
a.set(3, 3, new Complex(1.0, 6.0));
a.set(3, 4, new Complex(-1.0, 3.0));
a.set(4, 0, new Complex(-5.0, 4.0));
a.set(4, 3, new Complex(-5.0, 0.0));
a.set(4, 4, new Complex(12.0, 2.0));
a.set(4, 5, new Complex(-7.0, 7.0));
a.set(5, 0, new Complex(-1.0, 12.0));
a.set(5, 1, new Complex(-2.0, 8.0));
a.set(5, 5, new Complex(3.0, 7.0));

// Compute the sparse LU factorization of a
ComplexSparseLU = new ComplexSuperLU(a);

ComplexSparseLU.setEquilibrate(false);
ComplexSparseLU.setColumnPermutationMethod(
        ComplexSuperLU.NATURAL_ORDERING);
ComplexSparseLU.setPivotGrowth(true);

// Set option of iterative refinement
ComplexSparseLU.setIterativeRefinement(true);

// Solve sparse system A*x = b1
System.out.println();
System.out.println("Solve sparse System Ax=b1");
System.out.println("=========================");
System.out.println();

sol = ComplexSparseLU.solve(b1);
new PrintMatrix("Solution").print(sol);
```

```
        Ferr = ComplexSparseLU.getForwardErrorBound();
        Berr = ComplexSparseLU.getRelativeBackwardError();

        System.out.println();
        System.out.println("Forward error bound: " + Ferr);
        System.out.println("Relative backward error: " + Berr);
        System.out.println();
        System.out.println();

        // Solve sparse system A^H*x = b2
        System.out.println();
        System.out.println("Solve sparse System A^Hx=b2");
        System.out.println("===========================");
        System.out.println();

        sol = ComplexSparseLU.solveConjugateTranspose(b2);
        new PrintMatrix("Solution").print(sol);

        Ferr = ComplexSparseLU.getForwardErrorBound();
        Berr = ComplexSparseLU.getRelativeBackwardError();

        System.out.println();
        System.out.println("Forward error bound: " + Ferr);
        System.out.println("Relative backward error: " + Berr);
        System.out.println();
        System.out.println();

        // Compute reciprocal pivot growth factor and condition number
        recip_pivot_growth
                = ComplexSparseLU.getReciprocalPivotGrowthFactor();
        conditionNumber = ComplexSparseLU.getConditionNumber();

        System.out.println("Pivot growth factor and condition number");
        System.out.println("========================================");
        System.out.println();

        System.out.println("Reciprocal pivot growth factor: "
                + recip_pivot_growth);
        System.out.println("Reciprocal condition number: "
                + conditionNumber);
        System.out.println();
    }
}
```

Output

```
Solve sparse System Ax=b1
=========================

Solution
    0
0  1+1i
1  2+2i
```

```
2  3+3i
3  4+4i
4  5+5i
5  6+6i


Forward error bound: 2.8393330592805326E-15
Relative backward error: 1.708035422500241E-16



Solve sparse System A^Hx=b2
===========================

Solution
    0
0  1+1i
1  2+2i
2  3+3i
3  4+4i
4  5+5i
5  6+6i


Forward error bound: 8.54834098797111E-15
Relative backward error: 1.0297720808117394E-16


Pivot growth factor and condition number
========================================

Reciprocal pivot growth factor: 0.7993827160493826
Reciprocal condition number: 0.07006544790967506
```

Cholesky class

`public class com.imsl.math.Cholesky implements Serializable, Cloneable`

Cholesky factorization of a matrix of type `double`.

Class `Cholesky` uses the Cholesky-Banachiewicz algortithm to factor the matrix A.

The Cholesky factorization of a matrix is $A = RR^T$, where R is a lower triangular matrix. Thus,

$$A = RR^T = \begin{pmatrix} R_{11} & 0 & 0 \\ R_{21} & R_{22} & 0 \\ R_{31} & R_{32} & R_{33} \end{pmatrix} \begin{pmatrix} R_{11} & R_{21} & R_{31} \\ 0 & R_{22} & R_{32} \\ 0 & 0 & R_{33} \end{pmatrix} = \begin{pmatrix} R_{11}^2 & & (Symmetric) \\ R_{21}R_{11} & R_{22}^2 + R_{22}^2 & \\ R_{31}R_{11} & R_{31}R_{21} + R_{32}R_{22} & R_{31}^2 + R_{32}^2 + R_{33}^2 \end{pmatrix}$$

which leads to the following for the entries of the lower triangular marix R:

$$R_{i,j} = \frac{1}{R_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} R_{i,k} R_{j,k} \right), \text{ for } i > j$$

and

$$R_{i,i} = \sqrt{A_{i,i} - \sum_{k=1}^{i-1} R_{i,k}^2} \; .$$

The method `update` is based on the LINPACK routine `SCHUD`; see Dongarra et al. (1979) and updates the RR^T Cholesky factorization of the real symmetric positive definite matrix A after a rank-one matrix is added. Given this factorization, `update` computes the factorization $\tilde{R}$ such that

$$A + xx^T = \tilde{R}\tilde{R}^T \; .$$

Similarly, the method `downdate`, based on the LINPACK routine `SCHDD`; see Dongarra et al. (1979), downdates the RR^T Cholesky factorization of the real symmetric positive definite matrix A after a rank-one matrix is subtracted. `downdate` computes the factorization $\tilde{R}$ such that

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

This is not always possible, since $A - xx^T$ may not be positive definite.

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

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

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

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

is obtained.

Let a be the solution of the linear system $Ra = x$ 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)* by *(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 θ_i.

The Givens rotations are then used to form

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

The matrix

$$\tilde{R}$$

is lower triangular and

$$\tilde{x} = x$$

because

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

.

Constructor

Cholesky

```
public Cholesky(double[][] a) throws SingularMatrixException,
Cholesky.NotSPDException
```

Description

Create the Cholesky factorization of a symmetric positive definite matrix of type `double`.

Parameter

`a` – a `double` square matrix to be factored

Exceptions

`IllegalArgumentException` Thrown when the row lengths of matrix a are not equal (for example, the matrix edges are “jagged”.)

`SingularMatrixException` Thrown when the input matrix A is singular.

`NotSPDException` Thrown when the input matrix is not symmetric, positive definite.

Methods

downdate

```
public void downdate(double[] x) throws Cholesky.NotSPDException
```

Description

Downdates the factorization by subtracting a rank-1 matrix. The object will contain the Cholesky factorization of $A - xx^T$, where A is the previously factored matrix.

Parameter

`x` – A `double` array which specifies the rank-1 matrix. `x` is not modified by this function.

Exception

`NotSPDException` if $A - xx^T$ is not symmetric positive-definite.

---

getR

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

Description

Returns the R matrix that results from the Cholesky factorization.

Returns

a `double` matrix which contains the lower triangular R matrix that results from the Cholesky factorization such that $A = RR^T$

---

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`.

Parameter

`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 $A + xx^T$, where A is the previously factored matrix.

Parameter

`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
```

Cholesky.NotSPDException class

```
static public class com.imsl.math.Cholesky.NotSPDException extends
com.imsl.IMSLException
```

The matrix is not symmetric, positive definite.

Constructor

Cholesky.NotSPDException

```
public Cholesky.NotSPDException()
```

Description

Constructs a `NotSPDException` object.

SparseCholesky class

```
public class com.imsl.math.SparseCholesky implements Serializable
```

Sparse Cholesky factorization of a matrix of type `SparseMatrix`.

Class `SparseCholesky` computes the Cholesky factorization of a sparse symmetric positive definite matrix A. This factorization can then be used to compute the solution of the linear system $Ax = b$.

Typically, the solution of a large sparse positive definite system $Ax = b$ is done in 4 steps:

1. In step one, an ordering algorithm is used to preserve sparsity in the Cholesky factor L of matrix A during the numerical factorization process. The new order can be described by a permutation matrix P.
2. Step two consists of setting up the data structure for the Cholesky factor L, where $PAP^T = LL^T$. This step is called the symbolic factorization phase of the computation. During symbolic factorization, only the sparsity pattern of sparse matrix A, i.e., the locations of the nonzero entries of matrix A are needed but not any of the elements themselves.
3. In step 3, the numerical factorization phase, the Cholesky factorization is done numerically.
4. Step 4 is the solution phase. Here, the numerical solution, x, to the original system is obtained by solving the two triangular systems $Ly_1 = Pb$, $L^T y_2 = y_1$ and the permutation $x = P^T y_2$.

Class `SparseCholesky` realizes all four steps by algorithms described in George and Liu (1981). Especially, step one, is a realization of a minimum degree ordering algorithm. The numerical factorization in its standard form is based on a sparse compressed storage scheme. Alternatively, a multifrontal method can be used. The multifrontal method requires more storage but will be faster than the standard method in certain cases. The multifrontal method is based on the routines in Liu (1987). For a detailed description of this method, see Liu (1990), also Duff and Reid(1983, 1984), Ashcraft (1987) et al. (1987), and Liu (1986, 1989, 1992). The numerical factorization method can be specified by using the `setNumericFactorizationMethod` .

The `solve` method will compute the symbolic and numeric factorizations if they have not already been computed or supplied by the user through the `factorSymbolically` , `factorNumerically` , `setNumericFactor` , or `setSymbolicFactor` methods. These factorizations are retained for later use by the `solve` method when different right-hand sides are to be solved.

There is a special situation where computations can be simplified. If an application generates different sparse symmetric positive definite coefficient matrices that all have the same sparsity pattern, then by using methods `getSymbolicFactor` and `setSymbolicFactor` the symbolic factorization needs only be computed once.

Fields

MULTIFRONTAL_METHOD

```
static final public int MULTIFRONTAL_METHOD
```

Indicates the multifrontal method will be used for numeric factorization.

STANDARD_METHOD

```
static final public int STANDARD_METHOD
```

Indicates the method of George/Liu (1981) will be used for numeric factorization.

Constructor

SparseCholesky

```
public SparseCholesky(SparseMatrix A)
```

Description

Constructs the matrix structure for the Cholesky factorization of a sparse symmetric positive definite matrix of type `SparseMatrix`.

Parameter

`A` – The `SparseMatrix` symmetric positive definite matrix to be factored. Only the lower triangular part of the input matrix is used.

Methods

factorNumerically

```
public void factorNumerically() throws SparseCholesky.NotSPDException
```

Description

Computes the numeric factorization of a sparse real symmetric positive definite matrix.

This method numerically factors the instance of the constructed matrix A, where A is of type `SparseMatrix` and is symmetric positive definite. The factorization is obtained in several steps:

1. First, matrix A is permuted to reduce fill-in, leading to a sparse symmetric positive definite matrix PAP^T.
2. Then, matrix PAP^T is symbolically and numerically factored.

Note that the symbolic factorization is not done if the symbolic factor has been supplied by the user through the `setSymbolicFactor` method.

Exception

`NotSPDException` is thrown if the input matrix is not symmetric, positive definite.

factorSymbolically

```
public void factorSymbolically() throws SparseCholesky.NotSPDException
```

Description

Computes the symbolic factorization of a sparse real symmetric positive definite matrix.

This method symbolically factors the instance of the constructed matrix A, where A is of type `SparseMatrix` and is symmetric positive definite. The factorization is obtained in several steps:

1. First, matrix A is permuted to reduce fill-in, leading to a sparse symmetric positive definite system $PAP^T = Pb$.
2. Then, matrix PAP^T is symbolically factored.

Exception

`NotSPDException` is thrown if the input matrix is not symmetric, positive definite.

getLargestDiagonalElement

```
public double getLargestDiagonalElement()
```

Description

Returns the largest diagonal element of the Cholesky factor.

Returns

a `double` value specifying the largest diagonal element of the Cholesky factor. Use of this method is only sensible if a numeric factorization of the input matrix was done beforehand.

getNumberOfNonzeros

```
public long getNumberOfNonzeros()
```

Description

Returns the number of nonzeros in the Cholesky factor.

Returns

a `long` value specifying the number of nonzeros (including the diagonal) of the Cholesky factor.

getNumericFactor

```
public SparseCholesky.NumericFactor getNumericFactor()
```

Description

Returns the numeric Cholesky factor.

Returns

a `NumericFactor` containing the numeric Cholesky factor.

getNumericFactorizationMethod

```
public int getNumericFactorizationMethod()
```

Description

Returns the method used in the numerical factorization of the permuted input matrix.

Returns

an `int` value equal to `STANDARD_METHOD` = 0 or `MULTIFRONTAL_METHOD` = 1 representing the method used in the numeric factorization of the permuted input matrix. See `setNumericFactorizationMethod` for more details.

getSmallestDiagonalElement

```
public double getSmallestDiagonalElement()
```

Description

Returns the smallest diagonal element of the Cholesky factor.

Returns

a `double` value specifying the smallest diagonal element of the Cholesky factor. Use of this method is only sensible if a numeric factorization of the input matrix was done beforehand.

getSymbolicFactor

```
public SparseCholesky.SymbolicFactor getSymbolicFactor()
```

Description

Returns the symbolic Cholesky factor.

Returns

a `SymbolicFactor` containing the symbolic Cholesky factor.

setNumericFactor

```
public void setNumericFactor(SparseCholesky.NumericFactor numericFactor)
```

Description

Sets the numeric Cholesky factor to use in solving of a sparse positive definite system of linear equations $Ax = b$.

Parameter

`numericFactor` – a `NumericFactor` object containing the numeric Cholesky factor. By default the numeric factorization is computed.

---

setNumericFactorizationMethod

```
public void setNumericFactorizationMethod(int method)
```

Description

Defines the method used in the numerical factorization of the permuted input matrix.

Parameter

`method` – an `int` value specifying the method to choose:

Method Name	*Description*
STANDARD_METHOD	standard method as described by George/Liu (1981). This is the default.
MULTIFRONTAL_METHOD	multifrontal method

Exception

`IllegalArgumentException` This exception is thrown when the value for method is not `STANDARD_METHOD` or `MULTIFRONTAL_METHOD`.

---

setSymbolicFactor

```
public void setSymbolicFactor(SparseCholesky.SymbolicFactor symbolicFactor)
```

Description

Sets the symbolic Cholesky factor to use in solving a sparse positive definite system of linear equations $Ax = b$.

Parameter

`symbolicFactor` – a `SymbolicFactor` containing the symbolic Cholesky factor. By default the symbolic factorization is computed.

---

solve

```
public double[] solve(double[] b) throws SparseCholesky.NotSPDException
```

Description

Computes the solution of a sparse real symmetric positive definite system of linear equations $Ax = b$.

This method solves the linear system $Ax = b$, where A is symmetric positive definite. The solution is obtained in several steps:

1. First, matrix A is permuted to reduce fill-in, leading to a sparse symmetric positive definite system $PAP^T = Pb$.
2. Then, matrix PAP^T is symbolically and numerically factored.
3. The final solution is obtained by solving the systems $Ly_1 = Pb, L^T y_2 = y_1$ and $x = P^T y_2$.

By default this method implements all of the above steps. The factorizations are retained for later use by subsequent solves. By choosing appropriate methods within this class, the computation can be reduced to the solution of the system $Ax = b$ for a given or precomputed symbolic or numeric factor.

Parameter

`b` – a `double` vector of length equal to the order of matrix `A` representing the new right-hand side.

Returns

a `double` vector of length equal to the order of matrix `A` representing the solution to the system of linear equations $Ax = b$.

Exception

`NotSPDException` is thrown if the input matrix is not symmetric, positive definite.

Example: Sparse Cholesky Factorization

The Cholesky Factorization of a sparse symmetric positive definite matrix is used to solve a linear system. Some additional information about the Cholesky factorization is also computed.

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

public class SparseCholeskyEx1 {

    public static void main(String args[]) throws Exception {

        SparseMatrix A = new SparseMatrix(5, 5);
        A.set(0, 0, 10.0);
        A.set(1, 1, 20.0);
        A.set(2, 0, 1.0);
        A.set(2, 2, 30.0);
        A.set(3, 2, 4.0);
        A.set(3, 3, 40.0);
        A.set(4, 0, 2.0);
        A.set(4, 1, 3.0);
        A.set(4, 3, 5.0);
        A.set(4, 4, 50.0);

        double b[] = {55.0, 83.0, 103.0, 97.0, 82.0};

        SparseCholesky cholesky = new SparseCholesky(A);

        // Choose Multifrontal method as numeric factorization method
        cholesky.setNumericFactorizationMethod(
                SparseCholesky.MULTIFRONTAL_METHOD);

        // Compute solution
        double solution[] = cholesky.solve(b);
        new PrintMatrix("Computed Solution").print(solution);

        // Print additional information about the factorization
        System.out.println("Smallest diagonal element of Cholesky factor: "
                + cholesky.getSmallestDiagonalElement());
        System.out.println("Largest diagonal element of Cholesky factor: "
                + cholesky.getLargestDiagonalElement());
        System.out.println("Number of nonzeros in Cholesky factor: "
                + cholesky.getNumberOfNonzeros());
    }
```

```
}
```

Output

```
Computed Solution
   0
0  5
1  4
2  3
3  2
4  1

Smallest diagonal element of Cholesky factor: 3.1622776601683795
Largest diagonal element of Cholesky factor: 7.010706098532443
Number of nonzeros in Cholesky factor: 11
```

SparseCholesky.NotSPDException class

`static public class com.imsl.math.SparseCholesky.NotSPDException extends com.imsl.IMSLException`

The matrix is not symmetric, positive definite.

Constructor

SparseCholesky.NotSPDException

`public SparseCholesky.NotSPDException()`

Description

Constructs a `NotSPDException` object.

SparseCholesky.SymbolicFactor class

`static public class com.imsl.math.SparseCholesky.SymbolicFactor implements Serializable`

The symbolic Cholesky factorization of a matrix.

Used by `getSymbolicFactor` and `setSymbolicFactor` to hold the symbolic Cholesky factorization of a matrix.

SparseCholesky.NumericFactor class

`static public class com.imsl.math.SparseCholesky.NumericFactor implements Serializable`

The numeric Cholesky factorization of a matrix.

Used by `getNumericFactor` and `setNumericFactor` to hold the numeric Cholesky factorization of a matrix.

ComplexSparseCholesky class

`public class com.imsl.math.ComplexSparseCholesky implements Serializable`

Sparse Cholesky factorization of a matrix of type `ComplexSparseMatrix`.

Class `ComplexSparseCholesky` computes the Cholesky factorization of a sparse Hermitian positive definite matrix A. This factorization can then be used to compute the solution of the linear system $Ax = b$.

Typically, the solution of a large sparse positive definite system $Ax = b$ is done in four steps.

1. In the first step, an ordering algorithm is used to preserve sparsity in the Cholesky factor L of matrix A during the numerical factorization process. The new order can be described by a permutation matrix P.
2. Step two consists of setting up the data structure for the Cholesky factor L, where $PAP^T = LL^T$. This step is called the symbolic factorization phase of the computation. During symbolic factorization, only the sparsity pattern of sparse matrix A, i.e., the locations of the nonzero entries of matrix A are needed but not any of the elements themselves.
3. In step 3, the numerical factorization phase, the Cholesky factorization is done numerically.
4. Step 4 is the solution phase. Here, the numerical solution, x, to the original system is obtained by solving the two triangular systems $Ly_1 = Pb$, $L^T y_2 = y_1$ and the permutation $x = P^T y_2$.

Class `ComplexSparseCholesky` realizes all four steps by algorithms described in George and Liu (1981). Especially, step one, is a realization of a minimum degree ordering algorithm. The numerical factorization in its standard form is based on a sparse compressed storage scheme. Alternatively, a multifrontal method can be used. The multifrontal method requires more storage but will be faster than

the standard method in certain cases. The multifrontal method is based on the routines in Liu (1987). For a detailed description of this method, see Liu (1990), also Duff and Reid(1983, 1984), Ashcraft (1987) et al. (1987), and Liu (1986, 1989, 1992). The numerical factorization method can be specified by using the `setNumericFactorizationMethod`.

The `solve` method will compute the symbolic and numeric factorizations if they have not already been computed or supplied by the user through the `factorSymbolically`, `factorNumerically`, `setNumericFactor`, or `setSymbolicFactor` methods. These factorizations are retained for later use by the `solve` method when different right-hand sides are to be solved.

There is a special situation where computations can be simplified. If an application generates different sparse Hermitian positive definite coefficient matrices that all have the same sparsity pattern, then by using methods `getSymbolicFactor` and `setSymbolicFactor` the symbolic factorization needs only be computed once.

Fields

MULTIFRONTAL_METHOD

```
static final public int MULTIFRONTAL_METHOD
```

Indicates the multifrontal method will be used for numeric factorization.

STANDARD_METHOD

```
static final public int STANDARD_METHOD
```

Indicates that the method of George/Liu (1981) is used for numeric factorization.

Constructor

ComplexSparseCholesky

```
public ComplexSparseCholesky(ComplexSparseMatrix A)
```

Description

Constructs the matrix structure for the Cholesky factorization of a sparse Hermitian positive definite matrix of type `ComplexSparseMatrix`.

Parameter

`A` – The `ComplexSparseMatrix` Hermitian positive definite matrix to be factored. Only the lower triangular part of the input matrix is used.

Methods

factorNumerically

`public void factorNumerically() throws ComplexSparseCholesky.NotSPDException`

Description

Computes the numeric factorization of a sparse Hermitian positive definite matrix.

This method numerically factors the instance of the constructed matrix A, where A is of type `ComplexSparseMatrix` and is Hermitian positive definite. The factorization is obtained in several steps:

1. First, matrix A is permuted to reduce fill-in, leading to a sparse Hermitian positive definite matrix PAP^T.
2. Then, matrix PAP^T is symbolically and numerically factored.

Note that the symbolic factorization is not done if the symbolic factor has been supplied by the user through the `setSymbolicFactor` method.

Exception

`NotSPDException` is thrown if the input matrix is not Hermitian, positive definite.

factorSymbolically

`public void factorSymbolically() throws ComplexSparseCholesky.NotSPDException`

Description

Computes the symbolic factorization of a sparse Hermitian positive definite matrix.

This method symbolically factors the instance of the constructed matrix A, where A is of type `ComplexSparseMatrix` and is Hermitian positive definite. The factorization is obtained in several steps:

1. First, matrix A is permuted to reduce fill-in, leading to a sparse Hermitian positive definite matrix PAP^T.
2. Then, matrix PAP^T is symbolically factored.

Exception

`NotSPDException` is thrown if the input matrix is not Hermitian, positive definite.

getLargestDiagonalElement

`public double getLargestDiagonalElement()`

Description

Returns the largest diagonal element of the Cholesky factor.

Returns

a `double` value specifying the largest diagonal element of the Cholesky factor. Use of this method is only sensible if a numeric factorization of the input matrix was done beforehand.

getNumberOfNonzeros

`public long getNumberOfNonzeros()`

Description

Returns the number of nonzeros in the Cholesky factor.

Returns

a `long` value specifying the number of nonzeros (including the diagonal) of the Cholesky factor.

getNumericFactor

`public ComplexSparseCholesky.NumericFactor getNumericFactor()`

Description

Returns the numeric Cholesky factor.

Returns

a `NumericFactor` containing the numeric Cholesky factor.

getNumericFactorizationMethod

`public int getNumericFactorizationMethod()`

Description

Returns the method used in the numerical factorization of the permuted input matrix.

Returns

an `int` value equal to `STANDARD_METHOD` = 0 or `MULTIFRONTAL_METHOD` = 1 representing the method used in the numeric factorization of the permuted input matrix. See `setNumericFactorizationMethod` for more details.

getSmallestDiagonalElement

`public double getSmallestDiagonalElement()`

Description

Returns the smallest diagonal element of the Cholesky factor.

Returns

a `double` value specifying the smallest diagonal element of the Cholesky factor. Use of this method is only sensible if a numeric factorization of the input matrix was done beforehand.

getSymbolicFactor

`public ComplexSparseCholesky.SymbolicFactor getSymbolicFactor()`

Description

Returns the symbolic Cholesky factor.

Returns

a `SymbolicFactor` containing the symbolic Cholesky factor.

setNumericFactor

`public void setNumericFactor(ComplexSparseCholesky.NumericFactor numericFactor)`

Description

Sets the numeric Cholesky factor to use in solving a sparse complex Hermitian positive definite system of linear equations $Ax = b$.

Parameter

`numericFactor` – a `NumericFactor` object containing the numeric Cholesky factor. By default the numeric factorization is computed.

setNumericFactorizationMethod

```
public void setNumericFactorizationMethod(int method)
```

Description

Defines the method used in the numerical factorization of the permuted input matrix.

Parameter

`method` – an `int` value specifying the method to choose:

Method Name	*Description*
STANDARD_METHOD	standard method as described by George/Liu (1981). This is the default.
MULTIFRONTAL_METHOD	multifrontal method

Exception

`IllegalArgumentException` This exception is thrown when the value for method is not `STANDARD_METHOD` or `MULTIFRONTAL_METHOD`.

setSymbolicFactor

```
public void setSymbolicFactor(ComplexSparseCholesky.SymbolicFactor
symbolicFactor)
```

Description

Sets the symbolic Cholesky factor to use in solving a sparse complex Hermitian positive definite system of linear equations $Ax = b$.

Parameter

`symbolicFactor` – a `SymbolicFactor` containing the symbolic Cholesky factor. By default the symbolic factorization is computed.

solve

```
public Complex[] solve(Complex[] b) throws
ComplexSparseCholesky.NotSPDException
```

Description

Computes the solution of a sparse Hermitian positive definite system of linear equations $Ax = b$.

This method solves the linear system $Ax = b$, where A is Hermitian positive definite. The solution is obtained in several steps:

1. First, matrix A is permuted to reduce fill-in, leading to a sparse Hermitian positive definite system $PAP^T = Pb$.
2. Then, matrix PAP^T is symbolically and numerically factored.
3. The final solution is obtained by solving the systems $Ly_1 = Pb, L^T y_2 = y_1$ and $x = P^T y_2$.

By default this method implements all of the above steps. The factorizations are retained for later use by subsequent solves. By choosing appropriate methods within this class, the computation can be reduced to the solution of the system $Ax = b$ for a given or precomputed symbolic or numeric factor.

Parameter

`b` – A `Complex` vector of length equal to the order of A containing the right-hand side.

Returns

a `Complex` vector of length equal to the order of matrix A containing the solution of the system $Ax = b$.

Exception

`NotSPDException` is thrown if the input matrix is not Hermitian, positive definite.

Example: Complex Sparse Cholesky Factorization

The Cholesky Factorization of a sparse hermitian positive definite matrix is used to solve a linear system. Some additional information about the Cholesky factorization is also computed.

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

public class ComplexSparseCholeskyEx1 {

    public static void main(String args[]) throws IMSLException {

        ComplexSparseMatrix A = new ComplexSparseMatrix(3, 3);

        A.set(0, 0, new Complex(2.0, 0.0));
        A.set(1, 0, new Complex(-1.0, -1.0));
        A.set(1, 1, new Complex(4.0, 0.0));
        A.set(2, 1, new Complex(1.0, -2.0));
        A.set(2, 2, new Complex(10.0, 0.0));

        Complex[] b = {
            new Complex(-2., 2.), new Complex(5., 15.), new Complex(36., 28.)
        };

        ComplexSparseCholesky cholesky = new ComplexSparseCholesky(A);

        // Choose Multifrontal method as numeric factorization method
        cholesky.setNumericFactorizationMethod(cholesky.MULTIFRONTAL_METHOD);

        // Compute solution
        Complex[] solution = cholesky.solve(b);

        PrintMatrix p = new PrintMatrix("Computed solution");
        p.print(solution);
```

```
        // Compute additional information about the factorization
        System.out.println("Smallest diagonal element of Cholesky factor: "
                + cholesky.getSmallestDiagonalElement());
        System.out.println("Largest diagonal element of Cholesky factor: "
                + cholesky.getLargestDiagonalElement());
        System.out.println("Number of nonzeros in Cholesky factor: "
                + cholesky.getNumberOfNonzeros());
    }
}
```

Output

```
Computed solution
    0
0  1+1i
1  2+2i
2  3+3i

Smallest diagonal element of Cholesky factor: 1.4142135623730951
Largest diagonal element of Cholesky factor: 2.8867513459481287
Number of nonzeros in Cholesky factor: 5
```

ComplexSparseCholesky.NotSPDException class

`static public class com.imsl.math.ComplexSparseCholesky.NotSPDException extends com.imsl.IMSLException`

The matrix is not Hermitian, positive definite.

Constructor

ComplexSparseCholesky.NotSPDException

`public ComplexSparseCholesky.NotSPDException()`

Description

Constructs a `NotSPDException` object.

ComplexSparseCholesky.NumericFactor class

```
static public class com.imsl.math.ComplexSparseCholesky.NumericFactor
implements Serializable
```

Data structures and functions for the numeric Cholesky factor.

Used by `getNumericFactor` and `setNumericFactor` to hold the numeric Cholesky factorization of a matrix.

ComplexSparseCholesky.SymbolicFactor class

```
static public class com.imsl.math.ComplexSparseCholesky.SymbolicFactor
implements Serializable
```

Data structures and functions for the symbolic Cholesky factor.

Used by `getSymbolicFactor` and `setSymbolicFactor` to hold the symbolic Cholesky factorization of a matrix.

QR class

```
public class com.imsl.math.QR implements Serializable, Cloneable
```

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.

Constructor

QR

`public QR(double[][] a)`

Description

Constructs the QR decomposition of a matrix with elements of type `double`.

Parameter

`a` – a `double` matrix to be factored

Exception

`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.

Parameter

`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 SingularMatrixException
```

Description

Returns the solution to the least-squares problem Ax = b.

Parameter

`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

Exception

`SingularMatrixException` Thrown when the upper triangular matrix R resulting from the QR factorization is singular.

solve

`public double[] solve(double[] b, double tol) throws 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

Exception

`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
```

SVD class

`public class com.imsl.math.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

$$U^T AV = \begin{cases} \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} & \text{if } n \geq p \\ [\Sigma \; 0] & \text{if } n \leq p \end{cases}$$

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 σ_k that is larger than a tolerance η. If τ 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 matrices U, V and Σ 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 $V_1 \Sigma_1^{-1} U_1^T$.

Constructors

SVD

```
public SVD(double[][] a) throws 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.

Parameter

`a` – a `double` matrix for which the singular value decomposition is to be computed

Exception

`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 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

Exceptions

`IllegalArgumentException` is thrown when the row lengths of input matrix a are not equal (for example, the matrix edges are “jagged”)

`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
```

SVD.DidNotConvergeException class

`static public class com.imsl.math.SVD.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge

Constructors

SVD.DidNotConvergeException

`public SVD.DidNotConvergeException(String message)`

Description

Constructs a `DidNotConvergeException` object.

Parameter

`message` – a `String` containing the error message

SVD.DidNotConvergeException

`public SVD.DidNotConvergeException(String key, Object[] arguments)`

Description

Constructs a `DidNotConvergeException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

GenMinRes class

`public class com.imsl.math.GenMinRes implements Serializable`

Linear system solver using the restarted Generalized Minimum Residual (GMRES) method.

`GenMinRes` implements restarted GMRES to generate an approximate solution to $Ax = b$. It is based on GMRES by Homer Walker (1988).

The GMRES method begins with an approximate solution x_0 and an initial residual $r_0 = b - Ax_0$. At iteration m, a correction z_m is determined in the Krylov subspace

$$\kappa_m(v) = \mathrm{span}(v, \mathrm{A}v, \ldots, \mathrm{A}^{\mathrm{m}-1}v)$$

$v = r_0$ which solves the least squares problem

$$\min_{z \in \kappa_m(r_0)} \|b - A(x_0 + z)\|_2$$

Then at iteration m, $x_m = x_0 + z_m$.

There are four distinct GMRES implementations, selectable through method `setMethod`. The first Gram-Schmidt implementation is essentially the original algorithm by Saad and Schultz (1986). The second Gram-Schmidt implementation, developed by Homer Walker and Lou Zhou, is simpler than the first implementation. The least squares problem is constructed in upper-triangular form and the residual vector updating at the end of a GMRES cycle is cheaper. The first Householder implementation is algorithm 2.2 of Walker (1988), but with more efficient correction accumulation at the end of each GMRES cycle. The second Householder implementation is algorithm 3.1 of Walker (1988). The products of Householder transformations are expanded as sums, allowing most work to be formulated as large scale matrix-vector operations.

The Gram-Schmidt implementations are less expensive than the Householder, the latter requiring about twice as many computations beyond the coefficient matrix/vector products. However, the Householder implementations may be more reliable near the limits of residual reduction. See Walker (1988) for details. Issues such as the cost of coefficient matrix/vector products, availability of effective preconditioners, and features of particular computing environments may serve to mitigate the extra expense of the Householder implementations.

Note that one can use the JAVA Logging API to generate intermediate output for the solver. Accumulated levels of detail correspond to JAVA's FINE and FINER logging levels with FINE yielding the smallest amount of information and FINER yielding the most. The levels of output yield the following:

Level	*Output*
FINE	A summary report is printed upon completion.
FINER	Lines of intermediate results giving the most important data for each step are printed after each step. A summary report is printed upon completion.

Fields

DIRECT_AT_RESTART_AND_TERMINATION

`static final public int DIRECT_AT_RESTART_AND_TERMINATION`

Indicates residual updating is to be done by direct evaluation upon restarting and at termination.

DIRECT_AT_RESTART_ONLY

`static final public int DIRECT_AT_RESTART_ONLY`

Indicates residual updating is to be done by direct evaluation upon restarting only.

FIRST_GRAM_SCHMIDT

`static final public int FIRST_GRAM_SCHMIDT`

Indicates the first Gram-Schmidt implementation method is to be used.

FIRST_HOUSEHOLDER

`static final public int FIRST_HOUSEHOLDER`

Indicates the first Householder implementation method is to be used.

LINEAR_AT_RESTART_AND_TERMINATION

`static final public int LINEAR_AT_RESTART_AND_TERMINATION`

Indicates residual updating is to be done by linear combination upon restarting and at termination.

LINEAR_AT_RESTART_ONLY

`static final public int LINEAR_AT_RESTART_ONLY`

Indicates residual updating is to be done by linear combination upon restarting only.

SECOND_GRAM_SCHMIDT

`static final public int SECOND_GRAM_SCHMIDT`

Indicates the second Gram-Schmidt implementation method is to be used.

SECOND_HOUSEHOLDER

```
static final public int SECOND_HOUSEHOLDER
```

Indicates the second Householder implementation method is to be used.

Constructor

GenMinRes

```
public GenMinRes(int n, GenMinRes.Function argF)
```

Description

GMRES linear system solver constructor.

Parameters

`n` – An `int` scalar value which defines the order of the system to be solved

`argF` – a `Function` that defines the user-supplied function which computes $z = Ap$. If `argF` implements `Preconditioner` then right preconditioning is performed using this user supplied function. Otherwise, no preconditioning is performed. Note that `argF` can be used to act upon the coefficients of matrix A stored in different storage modes. See the examples.

Methods

getGuess

```
public double[] getGuess()
```

Description

Returns the initial guess of the solution.

Returns

a `double` array of length `n` containing the initial guess of the solution.

getIterations

```
public int getIterations()
```

Description

Returns the actual number of GMRES iterations used.

Returns

an `int` scalar representing the number of iterations used.

getLogger

```
public Logger getLogger()
```

Description

Returns the logger object.

Returns

the logger object, if present, or null.

---

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the maximum number of iterations allowed.

Returns

an `int` specifying the maximum number of iterations allowed.

---

getMaxKrylovDim

```
public int getMaxKrylovDim()
```

Description

Returns the maximum Krylov subspace dimension, i.e., the maximum allowable number of GMRES iterations allowed before restarting.

Returns

An `int` scalar representing the maximum Krylov subspace dimension, i.e., the maximum allowable number of GMRES iterations allowed before restarting.

---

getMethod

```
public int getMethod()
```

Description

Returns the implementation method to be used.

Returns

an `int` scalar value specifying the implementation method to be used.

Return Value	*Method*
1 = `FIRST_GRAM_SCHMIDT`	Use the first Gram-Schmidt implementation. This is the default value used.
2 = `SECOND_GRAM_SCHMIDT`	Use the second Gram-Schmidt implementation.
3 = `FIRST_HOUSEHOLDER`	Use the first Householder implementation.
4 = `SECOND_HOUSEHOLDER`	Use the second Householder implementation.

---

getPreconditionerSolves

```
public int getPreconditionerSolves()
```

Description

Returns the total number of GMRES right preconditioner solves.

Returns

An `int` representing the number of GMRES right preconditioner solves.

getProducts

```
public int getProducts()
```

Description

Returns the total number of GMRES matrix-vector products used.

Returns

An `int` representing the number of GMRES matrix-vector products used.

getRelativeError

```
public double getRelativeError()
```

Description

Returns the stopping tolerance. The algorithm attempts to generate `x` such that $\|b - Ax\|_2 \leq t\|b\|_2$, where t = tolerance.

Returns

a `double` scalar value specifying the stopping tolerance.

getResidualNorm

```
public double getResidualNorm()
```

Description

Returns the final residual norm, $\|b - Ax\|_2$.

Returns

a `double` scalar value specifying the final residual norm.

getResidualUpdating

```
public int getResidualUpdating()
```

Description

Returns the residual updating method to be used.

Returns

an `int` scalar value specifying the residual updating method to be used.

Return Value	*Method*
1 = `LINEAR_AT_RESTART_ONLY`	Update by linear combination upon restarting only. This is the default value used.
2 = `LINEAR_AT_RESTART_AND_TERMINATION`	Update by linear combination upon restarting and at termination.
3 = `DIRECT_AT_RESTART_ONLY`	Update by direct evaluation upon restarting only.
4 = `DIRECT_AT_RESTART_AND_TERMINATION`	Update by direct evaluation upon restarting and at termination.

getVectorProducts

```
public GenMinRes.VectorProducts getVectorProducts()
```

Description

Returns the user-supplied functions for the inner product and, optionally, the norm used in the Gram-Schmidt implementations.

Returns

a `VectorProducts` that defines the user-supplied functions for the inner product and, optionally, the norm used in the Gram-Schmidt implementations.

setGuess

```
public void setGuess(double[] xguess)
```

Description

Set the initial guess of the solution. If this member function is not called, the elements of this array are set to 0.0.

Parameter

`xguess` – a `double` array of length `n` specifying the initial guess of the solution.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Set the maximum number of iterations allowed.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed. By default, `maxIterations` = 1000.

Exception

`IllegalArgumentException` is thrown if `maxIterations` is less than or equal to 0.

setMaxKrylovDim

```
public void setMaxKrylovDim(int kdmax)
```

Description

Set the maximum Krylov subspace dimension, i.e., the maximum allowable number of GMRES iterations allowed before restarting. If this member function is not called, `kdmax` is set to the minimum of `n` and 20.

Exception

`IllegalArgumentException` is thrown if `kdmax` is less than 1 or greater than `n`.

setMethod

```
public void setMethod(int iMethod)
```

Description

Set the implementation method to be used.

Parameter

`iMethod` – an `int` scalar value specifying the implementation method to be used. If this member function is not called, `iMethod` is set to `FIRST_GRAM_SCHMIDT`.

iMethod	*Description*
FIRST_GRAM_SCHMIDT	Use the first Gram-Schmidt implementation. This is the default value used.
SECOND_GRAM_SCHMIDT	Use the second Gram-Schmidt implementation.
FIRST_HOUSEHOLDER	Use the first Householder implementation.
SECOND_HOUSEHOLDER	Use the second Householder implementation.

Exception

`IllegalArgumentException` is thrown if `iMethod` is less than 1 or greater than 4.

setRelativeError

```
public void setRelativeError(double tolerance)
```

Description

Set the stopping tolerance. The algorithm attempts to generate `x` such that $\|b - Ax\|_2 \leq t\|b\|_2$, where t = tolerance. If this member function is not called, `tolerance` is set to 1.4901161193847656e-08.

Parameter

`tolerance` – a `double` scalar value specifying the stopping tolerance. By default, `tolerance` = 1.49e-08.

Exception

`IllegalArgumentException` is thrown if `tolerance` is less than or equal to 0.0

setResidualUpdating

```
public void setResidualUpdating(int rMethod)
```

Description

Set the residual updating method to be used.

Parameter

rMethod – an int scalar value specifying the residual updating method to be used. If this member function is not called, rMethod is set to LINEAR_AT_RESTART_ONLY.

rMethod	*Description*
LINEAR_AT_RESTART_ONLY	Update by linear combination upon restarting only. This is the default value used.
LINEAR_AT_RESTART_AND_TERMINATION	Update by linear combination upon restarting and at termination.
DIRECT_AT_RESTART_ONLY	Update by direct evaluation upon restarting only.
DIRECT_AT_RESTART_AND_TERMINATION	Update by direct evaluation upon restarting and at termination.

Exception

IllegalArgumentException is thrown if rMethod is less than 1 or greater than 4.

setVectorProducts

```
public void setVectorProducts(GenMinRes.VectorProducts argP)
```

Description

Sets the user-supplied functions for the inner product and, optionally, the norm to be used in the Gram-Schmidt implementations.

Parameter

argP – a VectorProduct specifying the user-defined function for the inner product and, optionally, the norm in the Gram-Schmidt implementations. If this member function is not called, the dot product will be used for the inner product and the L_2 norm will be used for the norm.

solve

```
public double[] solve(double[] b) throws SingularMatrixException,
GenMinRes.TooManyIterationsException
```

Description

Generate an approximate solution to $Ax = b$ using the Generalized Residual Method.

Parameter

b – a double array which defines the right-hand side of the linear system.

Returns

a double array containing the solution of the linear system.

Exception

IllegalArgumentException is thrown if if the length of b is not consistent with n.

Example 1: Solve a Small Linear System

A solution to a small linear system is found. The coefficient matrix is stored as a full matrix and no preconditioning is used. Typically, preconditioning is required to achieve convergence in a reasonable number of iterations.

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

public class GenMinResEx1 implements GenMinRes.Function {

    static private double a[][] = {
        {33.0, 16.0, 72.0},
        {-24.0, -10.0, -57.0},
        {18.0, -11.0, 7.0}
    };
    static private double b[] = {129.0, -96.0, 8.5};
    // If A were to be read in from some outside source the
    // code to read the matrix could reside in a constructor.

    public void amultp(double p[], double z[]) {
        double[] result;
        result = Matrix.multiply(a, p);
        System.arraycopy(result, 0, z, 0, z.length);
    }

    public static void main(String args[]) throws Exception {
        int n = 3;

        GenMinResEx1 atp = new GenMinResEx1();

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);

        // Solve Ax = b
        new PrintMatrix("x").print(gnmnrs.solve(b));
    }
}
```

Output

```
   x
    0
0  1
1  1.5
2  1
```

Example 2: Solve a Small Linear System with User Supplied Inner Product

A solution to a small linear system is found. The coefficient matrix is stored as a full matrix and no preconditioning is used. Typically, preconditioning is required to achieve convergence in a reasonable

number of iterations. The user supplies a function to compute the inner product and norm within the Gram-Schmidt implementation.

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

public class GenMinResEx2 implements GenMinRes.Function, GenMinRes.Norm {

    static private double a[][] = {
        {33.0, 16.0, 72.0},
        {-24.0, -10.0, -57.0},
        {18.0, -11.0, 7.0}
    };
    static private double b[] = {129.0, -96.0, 8.5};
    // If A were to be read in from some outside source the
    // code to read the matrix could reside in a constructor.

    public void amultp(double p[], double z[]) {
        double[] result;
        result = Matrix.multiply(a, p);
        System.arraycopy(result, 0, z, 0, z.length);
    }

    public double innerproduct(double[] x, double[] y) {
        int n = x.length;
        double tmp = 0.0;
        for (int i = 0; i < n; i++) {
            tmp += x[i] * y[i];
        }
        return tmp;
    }

    public double norm(double[] x) {
        int n = x.length;
        double tmp = 0.0;
        for (int i = 0; i < n; i++) {
            tmp += x[i] * x[i];
        }
        return Math.sqrt(tmp);
    }

    public static void main(String args[]) throws Exception {
        int n = 3;

        GenMinResEx2 atp = new GenMinResEx2();

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);
        gnmnrs.setVectorProducts(atp);

        // Solve Ax = b
        new PrintMatrix("x").print(gnmnrs.solve(b));
    }
}
```

Output

```
   x
    0
0  1
1  1.5
2  1
```

Example 3: Solve a Small Linear System Stored in Sparse Form

A solution to a small linear system in which the coefficient matrix has been stored in `SparseMatrix` form is found. An initial guess of ones is set before solving the system.

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

public class GenMinResEx3 implements GenMinRes.Function {

    static private SparseMatrix A;
    static private double a[] = {6.0, 10.0, 15.0, -3.0, 10.0,
        -1.0, -1.0, -3.0, -5.0, 1.0,
        10.0, -1.0, -2.0, -1.0, -2.0};
    static private double b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
    static private double xguess[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
    static private int irow[] = {5, 1, 2, 1, 3, 3, 4,
        4, 4, 4, 0, 5, 5, 1, 3};
    static private int jcol[] = {5, 1, 2, 2, 3, 4, 0,
        5, 3, 4, 0, 0, 1, 3, 0};

    public void amultp(double p[], double z[]) {
        double[] result;
        result = A.multiply(A, p);
        System.arraycopy(result, 0, z, 0, z.length);
    }

    public static void main(String args[]) throws Exception {
        int n = 6;

        A = new SparseMatrix(n, n);
        for (int i = 0; i < a.length; i++) {
            A.set(irow[i], jcol[i], a[i]);
        }

        GenMinResEx3 atp = new GenMinResEx3();

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);
        gnmnrs.setGuess(xguess);
        // Solve Ax = b
        new PrintMatrix("x").print(gnmnrs.solve(b));
    }
}
```

Output

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

Example 4: Solve a Small Linear System Stored in Sparse Form With Preconditioning

A solution to a small linear system in which the coefficient matrix has been stored in `SparseMatrix` form is found. An initial guess of ones is set before solving the system and preconditioning is used.

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

public class GenMinResEx4 implements GenMinRes.Preconditioner {

    static private SparseMatrix A;
    static private double a[] = {
        6.0, 10.0, 15.0, -3.0, 10.0,
        -1.0, -1.0, -3.0, -5.0, 1.0,
        10.0, -1.0, -2.0, -1.0, -2.0
    };
    static private double b[] = {10.0, 7.0, 45.0, 33.0, -34.0, 31.0};
    static private double xguess[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
    static private double diagin[] = {0.1, 0.1, 0.0666666666666667,
        0.1, 1.0, 0.16666666666666667};
    static private int irow[] = {5, 1, 2, 1, 3, 3, 4, 4, 4, 4, 0, 5, 5, 1, 3};
    static private int jcol[] = {5, 1, 2, 2, 3, 4, 0, 5, 3, 4, 0, 0, 1, 3, 0};

    public void amultp(double p[], double z[]) {
        double[] result;
        result = A.multiply(A, p);
        System.arraycopy(result, 0, z, 0, z.length);
    }

    public void preconditioner(double r[], double z[]) {
        int n = z.length;
        for (int i = 0; i < n; i++) {
            z[i] = diagin[i] * r[i];
        }
    }

    public static void main(String args[]) throws Exception {
        int n = 6;

        A = new SparseMatrix(n, n);
        for (int i = 0; i < a.length; i++) {
            A.set(irow[i], jcol[i], a[i]);
```

```
        }
        GenMinResEx4 atp = new GenMinResEx4();

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);
        gnmnrs.setGuess(xguess);
        // Solve Ax = b
        new PrintMatrix("x").print(gnmnrs.solve(b));
    }
}
```

Output

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

Example 5: The Second Householder Implementation

The coefficient matrix in this example corresponds to the five-point discretization of the 2-d Poisson equation with the Dirichlet boundary condition. Assuming the natural ordering of the unknowns, and moving all boundary terms to the right hand side, we obtain a block tridiagonal matrix. (Consider the tridiagonal matrix T which has the value 4.0 down the main diagonal and -1.0 along the upper and lower co-diagonals. Then the coefficient matrix is the block tridiagonal matrix consisting of T's down the main diagonal and -I along the upper and lower codiagonals where I is the identity matrix.) Discretizing on a 20 x 20 grid implies that the coefficient matrix is 400 x 400. In the solution, the second Householder implementation is selected and we choose to update the residual vector by direct evaluation.

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

public class GenMinResEx5 implements GenMinRes.Function {

    //Creates a new instance of GenMinResEx5
    public GenMinResEx5() {
    }

    public void amultp(double p[], double z[]) {
        int n = z.length;
        int k = (int) Math.sqrt(n);
        // Multiply by diagonal blocks
        for (int i = 0; i < n; i++) {
            z[i] = 4.0 * p[i];
        }
        for (int i = 0; i < n - 2; i++) {
            z[i] = -1.0 * p[i + 1] + z[i];
        }
```

```
        for (int i = 0; i < n - 2; i++) {
            z[i + 1] = -1.0 * p[i] + z[i + 1];
        }
        // Correct for terms not properly in block diagonal
        for (int i = k - 1; i < n - k; i = i + k) {
            z[i] += p[i + 1];
            z[i + 1] += p[i];
        }
        // Do the super and subdiagonal blocks, the -I's
        for (int i = 0; i < n - k; i++) {
            z[i] = -1.0 * p[i + k] + z[i];
        }
        for (int i = 0; i < n - k; i++) {
            z[i + k] = -1.0 * p[i] + z[i + k];
        }
    }

    public static void main(String args[]) throws Exception {
        int n = 400;
        double b[] = new double[n];
        double xguess[] = new double[n];

        GenMinResEx5 atp = new GenMinResEx5();

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);
        // Set right hand side and initial guess to ones
        for (int i = 0; i < n; i++) {
            b[i] = 1.0;
            xguess[i] = 1.0;
        }
        gnmnrs.setGuess(xguess);
        gnmnrs.setMethod(gnmnrs.SECOND_HOUSEHOLDER);
        gnmnrs.setResidualUpdating(gnmnrs.DIRECT_AT_RESTART_ONLY);
        // Solve Ax = b
        gnmnrs.solve(b);
        int iterations = gnmnrs.getIterations();
        System.out.println("The number of iterations used = " + iterations);
        double resnorm = gnmnrs.getResidualNorm();
        System.out.println("The final residual norm is " + resnorm);
    }
}
```

Output

```
The number of iterations used = 92
The final residual norm is 2.5264852954103667E-7
```

Example 6: The Second Householder Implementation With Preconditioning

The coefficient matrix in this example corresponds to the five-point discretization of the 2-d Poisson equation with the Dirichlet boundary condition. Assuming the natural ordering of the unknowns, and

moving all boundary terms to the right hand side, we obtain a block tridiagonal matrix. (Consider the tridiagonal matrix T which has the value 4.0 down the main diagonal and -1.0 along the upper and lower co-diagonals. Then the coefficient matrix is the block tridiagonal matrix consisting of T's down the main diagonal and -I along the upper and lower codiagonals where I is the identity matrix.) Discretizing on a 20 x 20 grid implies that the coefficient matrix is 400 x 400. In the solution, the second Householder implementation is selected and we choose to update the residual vector by direct evaluation. Preconditioning is used with the preconditioning matrix being a diagonal matrix with 4.0 down the main diagonal and -1.0 along the upper and lower co-diagonals. This `preconditioner` method solves this tridiagonal matrix.

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

public class GenMinResEx6 implements GenMinRes.Preconditioner {

    private double precondA[], precondB[], precondC[];

    // Creates a new instance of GenMinResEx6
    public GenMinResEx6(int n) throws Exception {
        precondA = new double[n];
        precondB = new double[n];
        precondC = new double[n];
        // Define the preconditioning matrix
        for (int i = 0; i < n; i++) {
            precondA[i] = 4.0;
            precondB[i] = -1.0;
            precondC[i] = -1.0;
        }
    }

    public void amultp(double p[], double z[]) {
        int m = z.length;
        int n = (int) Math.sqrt(m);
        // Multiply by diagonal blocks
        for (int i = 0; i < m; i++) {
            z[i] = 4.0 * p[i];
        }
        for (int i = 0; i < m - 2; i++) {
            z[i] -= p[i + 1];
        }
        for (int i = 0; i < m - 2; i++) {
            z[i + 1] -= p[i];
        }
        // Correct for terms not properly in block diagonal
        for (int i = n - 1; i < m - n; i = i + n) {
            z[i] += p[i + 1];
            z[i + 1] += p[i];
        }
        // Do the super and subdiagonal blocks, the -I's
        for (int i = 0; i < m - n; i++) {
            z[i] -= p[i + n];
        }
        for (int i = 0; i < m - n; i++) {
            z[i + n] -= p[i];
        }
```

```
    }

    /**
     * Solve the tridiagonal preconditioning matrix problem for z.
     */
    public void preconditioner(double r[], double z[]) {
        int n = z.length;
        double w[] = new double[n];
        double v[] = new double[n];
        double u[] = new double[n];
        w[0] = precondA[0];
        v[0] = precondC[0] / w[0];
        u[0] = r[0] / w[0];
        for (int i = 1; i < n; i++) {
            w[i] = precondA[i] - precondB[i] * v[i - 1];
            v[i] = precondC[i] / w[i];
            u[i] = (r[i] - precondB[i] * u[i - 1]) / w[i];
        }
        z[n - 1] = u[n - 1];
        for (int j = n - 2; j >= 0; j--) {
            z[j] = u[j] - v[j] * z[j + 1];
        }
    }

    public static void main(String args[]) throws Exception {
        int n = 400;
        double b[] = new double[n];
        double xguess[] = new double[n];

        GenMinResEx6 atp = new GenMinResEx6(n);

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);
        // Set right hand side and initial guess to ones
        for (int i = 0; i < n; i++) {
            b[i] = 1.0;
            xguess[i] = 1.0;
        }
        gnmnrs.setGuess(xguess);
        gnmnrs.setMethod(gnmnrs.SECOND_HOUSEHOLDER);
        gnmnrs.setResidualUpdating(gnmnrs.DIRECT_AT_RESTART_ONLY);
        // Solve Ax = b
        gnmnrs.solve(b);
        int iterations = gnmnrs.getIterations();
        System.out.println("The number of iterations used = " + iterations);
        double resnorm = gnmnrs.getResidualNorm();
        System.out.println("The final residual norm is " + resnorm);
    }
}
```

Output

```
The number of iterations used = 60
The final residual norm is 2.841414917241539E-7
```

Example 7: Solve a Small Linear System With Logging

A solution to a small linear system is found. The coefficient matrix is stored as a full matrix and no preconditioning is used. Typically, preconditioning is required to achieve convergence in a reasonable number of iterations. Logging is enabled so that intermediate output and a summary report are generated.

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

public class GenMinResEx7 implements GenMinRes.Function {

    static private double a[][] = {
        {33.0, 16.0, 72.0},
        {-24.0, -10.0, -57.0},
        {18.0, -11.0, 7.0}
    };
    static private double b[] = {129.0, -96.0, 8.5};
    // If A were to be read in from some outside source the
    // code to read the matrix could reside in a constructor.

    public void amultp(double p[], double z[]) {
        double[] result;
        result = Matrix.multiply(a, p);
        System.arraycopy(result, 0, z, 0, z.length);
    }

    public static void main(String args[]) throws Exception {
        int n = 3;

        GenMinResEx7 atp = new GenMinResEx7();

        // Construct a GenMinRes object
        GenMinRes gnmnrs = new GenMinRes(n, atp);
        Logger logger = gnmnrs.getLogger();
        Handler h = new java.util.logging.FileHandler("GenMinReslog.txt");
        logger.addHandler(h);
        logger.setLevel(Level.FINER);
        h.setFormatter(new com.imsl.IMSLFormatter());
        // Solve Ax = b
        new PrintMatrix("x").print(gnmnrs.solve(b));
    }
}
```

Output

```
   x
    0
0  1
1  1.5
2  1
```

GenMinRes.Function interface

```
public interface com.imsl.math.GenMinRes.Function
```

Public interface for the user supplied function to `GenMinRes`.

Method

amultp

```
public void amultp(double[] p, double[] z)
```

Description

Used to compute $z = Ap$ where `A` is the matrix of coefficients to solve and `p` and `z` are arrays of length `n`, the order of matrix `A`.

Parameters

- `p` – an input `double` array of length `n` generated during the implementation of the `solve` method.
- `z` – an output `double` array of length `n`.

GenMinRes.Preconditioner interface

```
public interface com.imsl.math.GenMinRes.Preconditioner implements
com.imsl.math.GenMinRes.Function
```

Public interface for the user supplied function to `GenMinRes` used for preconditioning.

Method

preconditioner

```
public void preconditioner(double[] r, double[] z)
```

Description

Used to compute $z = M^{-1}r$ where M is the preconditioning matrix and r and z are arrays of length `n`, the order of matrix M.

Parameters

r – an input double array of length n generated during the implementation of the solve method.

z – an output double array of length n.

GenMinRes.VectorProducts interface

```
public interface com.imsl.math.GenMinRes.VectorProducts
```

Public interface for the user supplied function to the GenMinRes object used for the inner product when the Gram-Schmidt implementation is used.

Method

innerproduct

```
public double innerproduct(double[] x, double[] y)
```

Description

Used to compute the inner product of 2 vectors for the Gram-Schmidt implementation. If this function is not implemented, the dot product is used for the inner product.

Parameters

x – first input double vector which is to take part in the inner product.

y – second input double vector which is to take part in the inner product.

Returns

a double, the value of the inner product of x and y.

GenMinRes.Norm interface

```
public interface com.imsl.math.GenMinRes.Norm implements
com.imsl.math.GenMinRes.VectorProducts
```

Public interface for the user supplied function to the GenMinRes object used for the norm $\|X\|$ when the Gram-Schmidt implementation is used.

Method

norm

```
public double norm(double[] x)
```

Description

Used to compute the norm $\|X\|$ in the Gram-Schmidt implementation. If this function is not implemented, the L_2 norm is used.

Parameter

`x` – input `double` vector for which the norm will be computed.

Returns

a `double`, the value of the norm of `x`.

GenMinRes.TooManyIterationsException class

```
static public class com.imsl.math.GenMinRes.TooManyIterationsException extends
com.imsl.IMSLException
```

Maximum number of iterations exceeded.

Constructors

GenMinRes.TooManyIterationsException

```
public GenMinRes.TooManyIterationsException(String message)
```

Description

Constructs a `TooManyIterationsException` object.

Parameter

`message` – a `String` containing the error message

GenMinRes.TooManyIterationsException

```
public GenMinRes.TooManyIterationsException(String key, Object[] arguments)
```

Description

Constructs a `TooManyIterationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

GenMinRes.Formatter class

```
static public class com.imsl.math.GenMinRes.Formatter extends
java.util.logging.Formatter
```

Simple formatter for GenMinRes logging

Constructor

GenMinRes.Formatter

```
public GenMinRes.Formatter()
```

Method

format

```
public String format(LogRecord record)
```

ConjugateGradient class

```
public class com.imsl.math.ConjugateGradient implements Serializable
```

Solves a real symmetric definite linear system using the conjugate gradient method with optional preconditioning.

Class `ConjugateGradient` solves the symmetric positive or negative definite linear system $Ax = b$ using the conjugate gradient method with optional preconditioning. This method is described in detail by Golub and Van Loan (1983, Chapter 10), and in Hageman and Young (1981, Chapter 7).

The preconditioning matrix M is a matrix that approximates A, and for which the linear system $Mz=r$ is easy to solve. These two properties are in conflict; balancing them is a topic of current research. If no preconditioning matrix is specified, M is set to the identity, i.e. $M = I$.

The number of iterations needed depends on the matrix and the error tolerance. As a rough guide,

$$\text{itmax} = \sqrt{n} \text{ for } n \gg 1,$$

where n is the order of matrix A.

See the references for details.

Let M be the preconditioning matrix, let b,p,r,x and z be vectors and let τ be the desired relative error. Then the algorithm used is as follows:

$\lambda = -1$

$p_0 = x_0$

$r_1 = b - Ap_0$

for $k = 1, \ldots, \texttt{itmax}$

$z_k = M^{-1} r_k$

if $k = 1$ then

$\beta_k = 1$

$p_k = z_k$

else

$\beta_k = (z_k^T r_k)/(z_{k-1}^T r_{k-1})$

$p_k = z_k + \beta_k p_{k-1}$

endif

$\alpha_k = (r_k^T z_k)/(p_k^T A p_k)$

$x_k = x_{k-1} + \alpha_k p_k$

$r_{k+1} = r_k - \alpha_k A p_k$

if $(\|Ap_k\|_2 \le \tau(1-\lambda)\|x_k\|_2)$ then

recompute λ

if $(\|Ap_k\|_2 \le \tau(1-\lambda)\|x_k\|_2)$ exit

endif

endfor

Here, λ is an estimate of $\lambda_{\max}(\Gamma)$, the largest eigenvalue of the iteration matrix $\Gamma = I - M^{-1}A$. The stopping criterion is based on the result (Hageman and Young 1981, pp. 148-151)

$$\frac{\|x_k - x\|_M}{\|x\|_M} \le \left(\frac{1}{1-\lambda_{\max}(\Gamma)}\right)\left(\frac{\|z_k\|_M}{\|x_k\|_M}\right),$$

where

$$\|x\|_M^2 = x^T M x\,.$$

It is also known that

$$\lambda_{\max}(T_1) \le \lambda_{\max}(T_2) \le \ldots \le \lambda_{\max}(\Gamma) < 1,$$

where the T_l are the symmetric, tridiagonal matrices

$$T_l = \begin{bmatrix} \mu_1 & \omega_2 & & & \\ \omega_2 & \mu_2 & \omega_3 & & \\ & \omega_3 & \mu_3 & \ddots & \\ & & \ddots & \ddots & \omega_l \\ & & & \omega_l & \mu_l \end{bmatrix}$$

with $\mu_1 = 1 - 1/\alpha_1$ and, for $k = 2, \ldots, l$,

$$\mu_k = 1 - \beta_k/\alpha_{k-1} - 1/\alpha_k \quad \text{and} \quad \omega_k = \sqrt{\beta_k}/\alpha_{k-1}.$$

Usually, the eigenvalue computation is needed for only a few of the iterations.

Constructor

ConjugateGradient

```
public ConjugateGradient(int n, ConjugateGradient.Function argF)
```

Description

Conjugate gradient constructor.

Parameters

`n` – an `int` scalar value defining the order of the matrix.

`argF` – a `Function` that defines the user-supplied function which computes $z = Ap$. If `argF` implements `Preconditioner` then right preconditioning is performed using this user supplied function. Otherwise, no preconditioning is performed. Note that `argF` can be used to act upon the coefficients of matrix A stored in different storage modes.

Methods

getIterations

```
public int getIterations()
```

Description

Returns the number of iterations needed by the conjugate gradient algorithm.

Returns

an `int` value indicating the number of iterations needed.

getJacobi

```
public double[] getJacobi()
```

Description

Returns the Jacobi preconditioning matrix.

Returns

a `double` vector `diagonal` containing the diagonal of the Jacobi preconditioner M, that is, `diagonal[i]`$=A_{i,i}$, A the input matrix.

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the maximum number of iterations allowed.

Returns

an `int` value specifying the maximum number of iterations allowed.

getRelativeError

```
public double getRelativeError(double errorRelative)
```

Description

Returns the relative error used for stopping the algorithm.

Returns

a `double` containing the relative error.

setJacobi

```
public void setJacobi(double[] diagonal)
```

Description

Defines a Jacobi preconditioner as the preconditioning matrix, that is, M is the diagonal of `A`.

Parameter

`diagonal` – a `double` vector containing the diagonal of A as the Jacobi preconditioner M, that is, `diagonal[i]`$=A_{i,i}$.

Exception

`IllegalArgumentException` is thrown if the length of vector `diagonal` is not equal to the order `n` of input matrix A.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Sets the maximum number of iterations allowed.

Parameter

`maxIterations` – an `int` value specifying the maximum number of iterations allowed. By default, `maxIterations` $= \max(1000, \sqrt{n})$.

Exception

`IllegalArgumentException` is thrown if `maxIterations` is less than or equal to 0.

setRelativeError

```
public void setRelativeError(double tolerance)
```

Description

Sets the relative error used for stopping the algorithm.

Parameter

`tolerance` – a `double` specifying the relative error. By default, `tolerance` = 1.49e-08, the square root of the precision.

Exception

`IllegalArgumentException` is thrown if `tolerance` is less than 0.

solve

```
public double[] solve(double[] b) throws
ConjugateGradient.SingularPreconditionMatrixException,
ConjugateGradient.NotDefinitePreconditionMatrixException,
SingularMatrixException, ConjugateGradient.NotDefiniteAMatrixException,
ConjugateGradient.NoConvergenceException,
ConjugateGradient.NotDefiniteJacobiPreconditionerException
```

Description

Solves a real symmetric positive or negative definite system $Ax = b$ using a conjugate gradient method with or without preconditioning.

Parameter

`b` – a `double` vector of length `n` containing the right-hand side.

Returns

a `double` vector of length `n` containing the approximate solution to the linear system.

Exceptions

`IllegalArgumentException` is thrown if the length of `b` is not consistent with the order `n` of `A`.

`SingularPreconditionMatrixException` is thrown if the preconditioning matrix is singular.

`NotDefinitePreconditionMatrixException` is thrown if the preconditioning matrix is not definite.

`SingularMatrixException` is thrown if input matrix A is singular.

`NotDefiniteAMatrixException` is thrown if matrix A is not definite.

`NoConvergenceException` is thrown if the algorithm is not convergent within `maxIterations` iterations.

`NotDefiniteJacobiPreconditionerException` is thrown if the Jacobi preconditioner is not definite.

Conjugate Gradient Example 1:

The solution to a positive definite linear system is found. The coefficient matrix is stored as a full matrix.

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

public class ConjugateGradientEx1 implements ConjugateGradient.Function {

    static private double[][] a = {
        {1.0, -3.0, 2.0},
        {-3.0, 10.0, -5.0},
        {2.0, -5.0, 6.0}
    };
    static private double[] b = {27.0, -78.0, 64.0};

    public void amultp(double[] p, double[] z) {
        double w[] = Matrix.multiply(a, p);
        System.arraycopy(w, 0, z, 0, z.length);
    }

    public static void main(String args[]) throws Exception {
        int n = 3;

        ConjugateGradientEx1 atp = new ConjugateGradientEx1();

        // Construct Cg object
        ConjugateGradient cg = new ConjugateGradient(n, atp);

        // Solve Ax=b
        new PrintMatrix("Solution").print(cg.solve(b));
    }
}
```

Output

```
Solution
   0
0   1
1  -4
2   7
```

Conjugate Gradient Example 2:

In this example, two different preconditioners are used to find the solution of a sparse positive definite linear system which occurs in a finite difference solution of Laplace's equation on a regular $c \times c$ grid, $c = 50$. The matrix is $A = E(c^2, c)$. For the first solution, Jacobi preconditioning with preconditioner $M = \text{diag}(A)$ is used. The required iteration number and maximum absolute error are printed. Next, a more complicated preconditioning matrix, consisting of the symmetric tridiagonal part of A, is used. Again, the iteration number and the maximum absolute error are printed. The iteration number is substantially reduced.

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

public class ConjugateGradientEx2 implements ConjugateGradient.Preconditioner {

    private SparseMatrix A;
    private SparseCholesky M;

    public ConjugateGradientEx2(int n, int c)
            throws SparseCholesky.NotSPDException {
        // Create matrix E(n,c), n>1, 1<c<n-1
        // See Osterby and Zlatev(1982), pp. 7-8
        A = new SparseMatrix(n, n);
        for (int j = 0; j < n; j++) {
            if (j - c >= 0) {
                A.set(j, j - c, -1.0);
            }
            if (j - 1 >= 0) {
                A.set(j, j - 1, -1.0);
            }
            A.set(j, j, 4.0);
            if (j + 1 < n) {
                A.set(j, j + 1, -1.0);
            }
            if (j + c < n) {
                A.set(j, j + c, -1.0);
            }
        }

        // Create and factor preconditioning matrix
        SparseMatrix C = new SparseMatrix(n, n);
        for (int j = 0; j < n; j++) {
            if (j - 1 >= 0) {
                C.set(j, j - 1, -1.0);
            }
            C.set(j, j, 4.0);
            if (j + 1 < n) {
                C.set(j, j + 1, -1.0);
            }
        }
        M = new SparseCholesky(C);
        M.factorSymbolically();
        M.factorNumerically();
    }

    public void amultp(double[] p, double[] z) {
        double w[] = A.multiply(p);
        System.arraycopy(w, 0, z, 0, w.length);
    }

    public void preconditioner(double[] r, double[] z) {
        try {
            double w[] = M.solve(r);
            System.arraycopy(w, 0, z, 0, w.length);
        } catch (SparseCholesky.NotSPDException e) {
            // No danger of NotSPDException because the system is only
            // solved here for given Cholesky factor. This exception
```

```
            // would have been thrown in the constructor already.
        }
    }

    public static void main(String args[]) throws Exception {
        int n = 2500;
        int c = 50;

        ConjugateGradientEx2 atp = new ConjugateGradientEx2(n, c);

        // Set a predetermined answer and diagonal
        double[] expected = new double[n];
        double[] diagonal = new double[n];
        for (int i = 0; i < n; i++) {
            expected[i] = (double) (i % 5);
            diagonal[i] = 4.0;
        }

        // Get right-hand side
        double[] b = new double[n];
        atp.amultp(expected, b);

        // Solve system with Jacobi preconditioning
        ConjugateGradient cgJacobi = new ConjugateGradient(n, atp);
        cgJacobi.setJacobi(diagonal);
        double solution[] = cgJacobi.solve(b);

        // Compute inf-norm of computed solution - exact solution,
        // print results
        double norm = 0.0;
        for (int i = 0; i < n; i++) {
            norm = Math.max(norm, Math.abs(solution[i] - expected[i]));
        }
        System.out.println("Jacobi preconditioning");
        System.out.println("Iterations= " + cgJacobi.getIterations()
                + ", norm= " + norm);
        System.out.println();

        /*
         * Solve the same system, with Cholesky preconditioner
         */
        ConjugateGradient cgCholesky = new ConjugateGradient(n, atp);
        solution = cgCholesky.solve(b);

        norm = 0.0;
        for (int i = 0; i < n; i++) {
            norm = Math.max(norm, Math.abs(solution[i] - expected[i]));
        }
        System.out.println("More general preconditioning");
        System.out.println("Iterations= " + cgCholesky.getIterations()
                + ", norm= " + norm);
    }
}
```

Output

```
Jacobi preconditioning
Iterations= 187, norm= 4.463440728130763E-10

More general preconditioning
Iterations= 127, norm= 5.137250624898115E-10
```

ConjugateGradient.SingularPreconditionMatrixException class

```
static public class
com.imsl.math.ConjugateGradient.SingularPreconditionMatrixException extends
com.imsl.IMSLException
```

The Precondition matrix is singular.

Constructors

ConjugateGradient.SingularPreconditionMatrixException

```
public ConjugateGradient.SingularPreconditionMatrixException(String message)
```

Description

Constructs a `SingularPreconditionMatrixException` object.

Parameter

`message` – a `String` containing the error message

ConjugateGradient.SingularPreconditionMatrixException

```
public ConjugateGradient.SingularPreconditionMatrixException(String key,
Object[] arguments)
```

Description

Constructs a `SingularPreconditionMatrixException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ConjugateGradient.NotDefinitePreconditionMatrixException class

```
static public class
com.imsl.math.ConjugateGradient.NotDefinitePreconditionMatrixException extends
com.imsl.IMSLException
```

The Precondition matrix is indefinite.

Constructors

ConjugateGradient.NotDefinitePreconditionMatrixException

```
public ConjugateGradient.NotDefinitePreconditionMatrixException(String message)
```

Description

Constructs a `NotDefinitePreconditionMatrixException` object.

Parameter

`message` – a `String` containing the error message

ConjugateGradient.NotDefinitePreconditionMatrixException

```
public ConjugateGradient.NotDefinitePreconditionMatrixException(String key,
Object[] arguments)
```

Description

Constructs a `NotDefinitePreconditionMatrixException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ConjugateGradient.NotDefiniteAMatrixException class

```
static public class com.imsl.math.ConjugateGradient.NotDefiniteAMatrixException
extends com.imsl.IMSLException
```

The input matrix A is indefinite, that is the matrix is not positive or negative definite.

Constructors

ConjugateGradient.NotDefiniteAMatrixException

`public ConjugateGradient.NotDefiniteAMatrixException(String message)`

Description

Constructs a `NotDefiniteAMatrixException` object.

Parameter

`message` – a `String` containing the error message

ConjugateGradient.NotDefiniteAMatrixException

`public ConjugateGradient.NotDefiniteAMatrixException(String key, Object[] arguments)`

Description

Constructs a `NotDefiniteAMatrixException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ConjugateGradient.NoConvergenceException class

```
static public class com.imsl.math.ConjugateGradient.NoConvergenceException
extends com.imsl.IMSLException
```

The conjugate gradient method did not converge within the allowed maximum number of iterations.

Constructors

ConjugateGradient.NoConvergenceException

`public ConjugateGradient.NoConvergenceException(String message)`

Description

Constructs a `NoConvergenceException` object.

Parameter

`message` – a `String` containing the error message

ConjugateGradient.NoConvergenceException

```
public ConjugateGradient.NoConvergenceException(String key, Object[] arguments)
```

Description

Constructs a `NoConvergenceException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ConjugateGradient.NotDefiniteJacobiPreconditionerException class

```
static public class
com.imsl.math.ConjugateGradient.NotDefiniteJacobiPreconditionerException
extends com.imsl.IMSLException
```

The Jacobi preconditioner is not strictly positive or negative definite.

Constructors

ConjugateGradient.NotDefiniteJacobiPreconditionerException

```
public ConjugateGradient.NotDefiniteJacobiPreconditionerException(String
message)
```

Description

Constructs a `NotDefiniteJacobiPreconditionerException` object.

Parameter

`message` – a `String` containing the error message

ConjugateGradient.NotDefiniteJacobiPreconditionerException

```
public ConjugateGradient.NotDefiniteJacobiPreconditionerException(String key,
Object[] arguments)
```

Description

Constructs a `NotDefiniteJacobiPreconditionerException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ConjugateGradient.Function interface

```
public interface com.imsl.math.ConjugateGradient.Function
```

Public interface for the user supplied function to `ConjugateGradient`.

Method

amultp

```
public void amultp(double[] p, double[] z)
```

Description

A user-supplied function which computes $z=Ap$.

Parameters

`p` – an input `double` vector of length dimension of A.

`z` – an output `double` vector containing the matrix-vector product Ap.

ConjugateGradient.Preconditioner interface

```
public interface com.imsl.math.ConjugateGradient.Preconditioner implements
com.imsl.math.ConjugateGradient.Function
```

Public interface for the user supplied function to `ConjugateGradient` used for preconditioning.

Method

preconditioner

```
public void preconditioner(double[] r, double[] z)
```

Description

Used to compute $z = M^{-1}r$ where M is the preconditioning matrix and r and z are arrays of length `n`, the order of matrix M.

Parameters

`r` – an input `double` array of length `n` generated during the implementation of the `solve` method.

`z` – an output `double` array of length `n`.

SingularMatrixException class

```
public class com.imsl.math.SingularMatrixException extends
com.imsl.IMSLException
```

The matrix is singular.

Constructor

SingularMatrixException

```
public SingularMatrixException()
```

Chapter 3: Eigensystem Analysis

Types

class Eigen 144
class SymEigen 148

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

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

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 τ. The performance index is defined to be

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

where ε is again the machine precision.

The performance index τ is related to the error analysis because

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

where E is the "nearby" matrix discussed above.

While the exact value of τ 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 τ 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 τ 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 λ_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 $\kappa_j \varepsilon \left\| A \right\|$. To compute an approximate bound for the relative accuracy of an eigenvalue, divide this bound by $\left| \lambda_j \right|$.

Eigen class

```
public class com.imsl.math.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, τ, 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).

Constructors

Eigen

```
public Eigen()
```

Description

Constructs the eigenvalues and the eigenvectors of a real square matrix.

Methods

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the maximum number of iterations.

Returns

an `int` containing the maximum number of iterations.

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.

Parameter

`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.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Set the maximum number of iterations allowed.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed. `maxIterations` must be greater than 0. By default, `maxIterations` = 50.

solve

```
public void solve(double[][] a, boolean computeVectors) throws
Eigen.DidNotConvergeException
```

Description

Solves for the eigenvalues and (optionally) the eigenvectors of a real square matrix.

Parameters

`a` – is the `double` square matrix for which the eigenvalues and (optionally) eigenvectors are to be found

`computeVectors` – is true if the eigenvectors are to be computed

Exception

`DidNotConvergeException` is thrown when the algorithm fails to converge on the eigenvalues of the matrix.

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();
        eigen.solve(a, true);
        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
```

Eigen.DidNotConvergeException class

`static public class com.imsl.math.Eigen.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge

Constructors

Eigen.DidNotConvergeException

`public Eigen.DidNotConvergeException(String message)`

Description

Constructs a `DidNotConvergeException` object.

Parameter

`message` – a `String` containing the error message

Eigen.DidNotConvergeException

`public Eigen.DidNotConvergeException(String key, Object[] arguments)`

Description

Constructs a `DidNotConvergeException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SymEigen class

`public class com.imsl.math.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, λ = the array of eigenvalues, and x_j is the associated eigenvector with jth eigenvalue.

Also, let ε be the machine precision. The performance index, τ, is defined to be

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

While the exact value of τ 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).

Constructors

SymEigen

```
public SymEigen(double[][] a)
```

Description

Constructs the eigenvalues and the eigenvectors for a real symmetric matrix.

Parameter

`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

Exception

`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.

Parameter

`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.

Exception

`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 4: Interpolation and Approximation

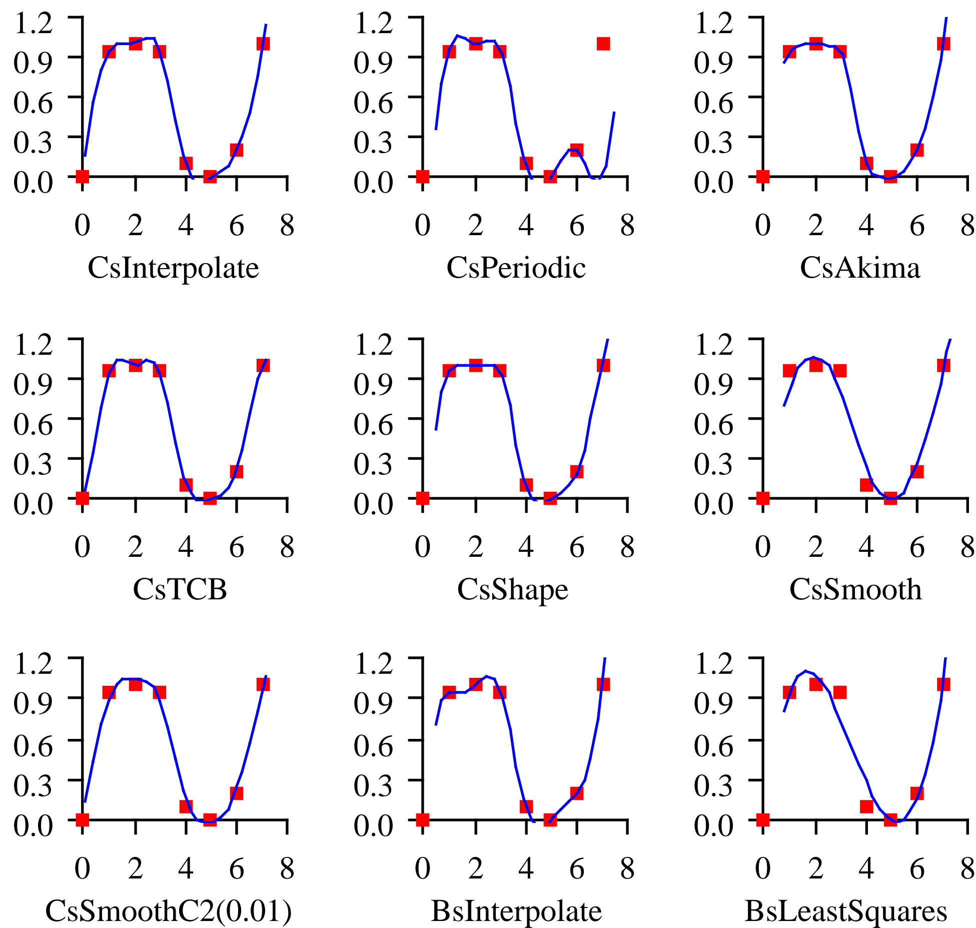
Types

class Spline 155
class CsAkima 159
class CsTCB 160
class CsInterpolate 166
class CsPeriodic 168
class CsShape 170
class CsSmooth 172
class CsSmoothC2 174
class BSpline 176
class BsInterpolate 180
class BsLeastSquares 182
class Spline2D 184
class Spline2DInterpolate 188
class Spline2DLeastSquares 197
class RadialBasis 202

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 five 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 nine 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 `CsTCB` allows the user to specify various datapoint parameters (such as tension, continuity, and bias).

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 BSpline is the abstract base class for univariate B-splines. B-splines provide a particularly convenient and suitable basis for a given class of smooth piecewise polynomial (ppoly) functions. Such a class is specified by giving its breakpoint sequence, its order k, and the required smoothness across each of the interior breakpoints. The corresponding B-spline basis is specified by giving its knot sequence $\mathbf{t} \in \mathbf{R}$. The specification rule is as follows: if the class is to have all derivatives up to and including the j-th derivative continuous across the interior breakpoint ξ_i, then the number ξ_i should occur *k-j-1* times in the knot sequence. Assuming that ξ_0 and ξ_{n-1} are the endpoints of the interval of interest, choose the first k knots equal to ξ_0 and the last k knots equal to ξ_{n-1}. This can be done because the B-splines are defined to be right continuous near ξ_0 and left continuous near ξ_{n-1}.

When the above construction is completed, a knot sequence $\mathbf{t}$ of length M is generated, and there are m = *M-k* B-splines of order k, for example $B_0, \ldots, B_{m-1}$, spanning the ppoly functions on the interval with the indicated smoothness. That is, each ppoly function in this class has a unique representation $p = a_0 B_0 + a_1 B_1 + \cdots + a_{m-1} B_{m-1}$ as a linear combination of B-splines. A B-spline is a particularly compact ppoly function. B_i is a nonnegative function that is nonzero only on the interval $[\mathbf{t}_i, \mathbf{t}_{i+k}]$ More precisely, the support of the i-th B-spline is $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. No ppoly function in the same class (other than the zero function) has smaller support (i.e., vanishes on more intervals) than a B-spline. This makes B-splines particularly attractive basis functions since the influence of any particular B-spline coefficient extends only over a few intervals. When it is necessary to emphasize the dependence of the B-spline on its parameters, we will use the notation $B_{i,j,\mathbf{t}}$ to denote the i-th B-spline of order k for the knot sequence $\mathbf{t}$.

Class BsInterpolate extends BSpline and creates a B-spline by interpolating data points.

Class BsLeastSquares extends BSpline and creates a B-spline by computing a least squares spline approximation to data points.

Class Spline2D is the abstract base class for the two-dimensional, tensor-product splines.

Class Spline2DInterpolate computes a Spline2D using interpolation from two-dimensional, tensor-product data.

Class Spline2DLeastSquares computes a Spline2D using least squares.

Class RadialBasis computes an approximation to scattered data in $\mathbf{R}^M$ using radial-basis functions.

Spline class

```
abstract public class com.imsl.math.Spline implements Serializable, Cloneable
```

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 ξ 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.

Fields

EPSILON_LARGE

```
static final protected double EPSILON_LARGE
```

The largest relative spacing for double.

breakPoint

```
protected double[] breakPoint
```

The breakpoint array of length n, where n is the number of piecewise polynomials.

coef

```
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]`).

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.

Parameter

`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 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

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

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.

Parameter

- `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.

Parameter

- `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

CsAkima class

`public class com.imsl.math.CsAkima extends com.imsl.math.Spline`

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 ξ 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.)

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.

Exception

`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
```

CsTCB class

`public class com.imsl.math.CsTCB extends com.imsl.math.Spline`

Extension of the Spline class to handle a tension-continuity-bias (TCB) cubic spline, also known as a Kochanek-Bartels spline and is a generalization of the Catmull-Rom spline.

Let $x =$`xData`, $y =$`yData`, and $n =$ the length of `xData` and `yData`. Class `CsTCB` computes the Kochanek-Bartels spline, a piecewise cubic Hermite spline interpolant to the set of data points $\{x_i, y_i\}$ for $i = 0, \ldots, n-1$. The breakpoints of the spline are the abscissas. As with all of the univariate interpolation functions, the abscissas need not be sorted.

The $\{x_i\}$ values are the knots, so the i-th interval is $[x_i, x_{i+1}]$. (To simplify the explanation, it is assumed that the data points are given in increasing order.) The cubic Hermite in the i-th segment has a starting value of y_i and an ending value of y_{i+1}. Its incoming tangent is

$$DS_i = \frac{1}{2}(1-t_i)(1-c_i)(1+b_i)\frac{y_i - y_{i-1}}{x_{i+1} - x_i} + \frac{1}{2}(1-t_i)(1+c_i)(1-b_i)\frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

where t_i is the i-th tension value, c_i is the i-th continuity value, and b_i is the i-th bias value. Its outgoing tangent is

$$DD_i = \frac{1}{2}(1-t_i)(1+c_i)(1+b_i)\frac{y_i - y_{i-1}}{x_{i+1} - x_i} + \frac{1}{2}(1-t_i)(1-c_i)(1-b_i)\frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

The value of the tangent at the endpoint, `left`, is given as:

$$\frac{y_0 - y_{-1}}{x_1 - x_0}$$

The value of the tangent at the endpoint, `right`, is given as:

$$\frac{y_n - y_{n-1}}{x_n - x_{n-1}}$$

By default the values of the tangents at the leftmost and rightmost endpoints are zero. These values can be reset via the `setLeftEndTangent` and `setRightEndTangent` methods.

The spline has a continuous first derivative (C^{-1}) if at each data point the left and right tangents are equal. This is true if the continuity parameters, c_i, are all zero. For any values of the parameters the spline is continuous (C^0).

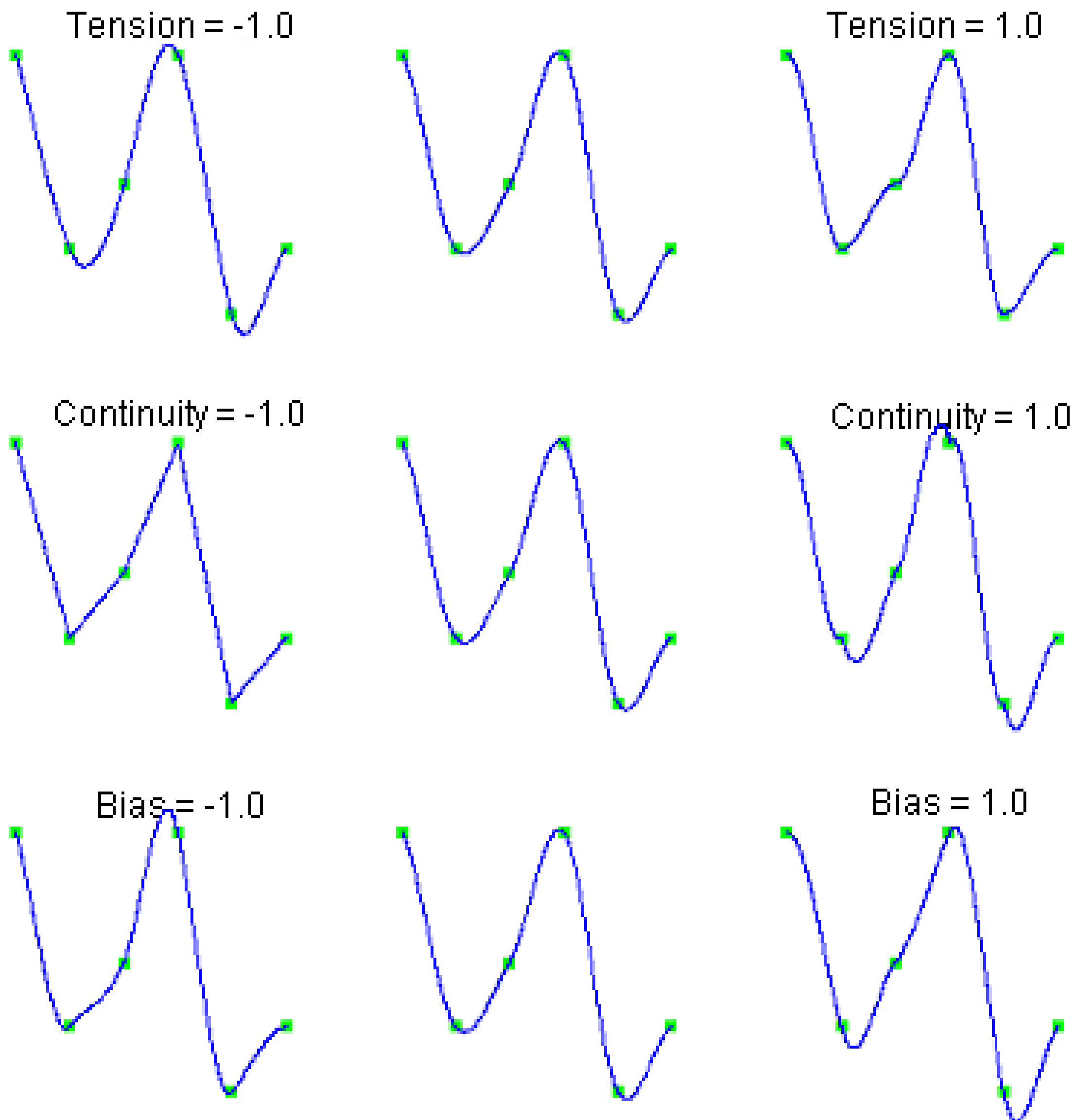
If $t_i = c_i = b_i = 0$ for all i, then the curve is the Catmull-Rom spline.

The following chart shows the same data points interpolated with different parameter values. All of the tension, continuity, and bias parameters are zero except for the labeled parameter, which has the indicated value at all data points.

Tension controls how sharply the spline bends at the data points. The tension values can be set via the `setTension` method. If `tension` values are near +1, the curve tightens. If the `tension` values are near -1, the curve slackens.

The continuity parameter controls the continuity of the first derivative. The continuity values can be set via the `setContinuity` method. If the `continuity` value is zero, the spline's first derivative is continuous, so the spline is C^{-1}.

The bias parameter controls the weighting of the left and right tangents. If zero, the tangents are equally weighted. If the bias parameter is near +1, the left tangent dominates. If the bias parameter is near -1, the right tangent dominates. The bias values can be set via the `setBias` method.



Constructor

CsTCB

```
public CsTCB(double[] xData, double[] yData)
```

Description

Constructs the tension-continuity-bias (TCB) cubic spline interpolant to the given data points.

Parameters

`xData` – a `double` array containing the x-coordinates of the data. Values must be distinct. `xData` and `yData` must be of the same length.

`yData` – a `double` array containing the y-coordinates of the data. `xData` and `yData` must be of the same length.

Methods

compute

```
public void compute()
```

Description

Computes the tension-continuity-bias (TCB) cubic spline interpolant.

getLeftEndTangent

```
public double getLeftEndTangent()
```

Description

Returns the value of the tangent at the leftmost endpoint.

Returns

A `double` value of the tangent at the leftmost endpoint.

getRightEndTangent

```
public double getRightEndTangent()
```

Description

Returns the value of the tangent at the rightmost endpoint.

Returns

A `double` value of the tangent at the rightmost endpoint.

setBias

```
public void setBias(double[] bias)
```

Description

Sets the bias values at the data points.

Parameter

`bias` – A `double` array of length `xData.length` which contains bias values in the interval [-1,1]. For each point, if the bias value is zero, the left and right side tangents are equally weighted. If the value is near +1, the left-side tangent dominates. If the value is near -1, the right side tangent dominates. By default, all values of `bias` are zero.

---

setContinuity

```
public void setContinuity(double[] continuity)
```

Description

Sets the continuity values at the data points.

Parameter

`continuity` – A `double` array of length `xData.length` which contains continuity values in the interval [-1,1]. For each point, if the continuity value is zero, the curve is C^1 at that point. Otherwise, the curve has a corner at that point, but is still continuous (C^0). By default, all values of `continuity` are zero.

---

setLeftEndTangent

```
public void setLeftEndTangent(double left)
```

Description

Sets the value of the tangent at the left endpoint.

Parameter

`left` – A `double` value of the tangent at the leftmost endpoint. The default value is zero.

---

setRightEndTangent

```
public void setRightEndTangent(double right)
```

Description

Sets the value of the tangent at the right endpoint.

Parameter

`right` – A `double` value of the tangent at the rightmost endpoint. The default value is zero.

---

setTension

```
public void setTension(double[] tension)
```

Description

Sets the tension values at the data points.

Parameter

`tension` – A `double` array of length `xData.length` which contains tension values in the interval [-1,1]. For each point, if the tension value is near +1, the curve is tightened at that point. If it is near -1, the curve is slack. By default, all values of `tension` are zero.

Example: The Kochanek-Bartels cubic spline interpolant

This example interpolates to a set of points. At x = 3, the continuity and tension parameters are -1. At all other points, they are zero. Interpolated values are then printed.

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

public class CsTCBEx1 {

    public static void main(String args[]) {

        double[] xdata = {0., 1., 2., 3., 4., 5.};
        double[] ydata = {5., 2., 3., 5., 1., 2.};
        double[] continuity = {0., 0., 0., -1., 0., 0.};
        double[] tension = {0., 0., 0., -1., 0., 0.};

        CsTCB cs = new CsTCB(xdata, ydata);
        cs.setContinuity(continuity);
        cs.setTension(tension);
        cs.compute();
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
        nf.setMinimumFractionDigits(4);
        System.out.println("      x     " + "   value   ");

        for (int k = 0; k < 11; k++) {
            double x = (double) k / 2.0;
            double y = cs.value(x);
            System.out.println("   " + nf.format(x) + "     " + nf.format(y));
        }
    }
}
```

Output

```
      x        value
   0.0000      5.0000
   0.5000      3.4375
   1.0000      2.0000
   1.5000      2.1875
   2.0000      3.0000
   2.5000      3.6875
   3.0000      5.0000
   3.5000      2.1875
   4.0000      1.0000
   4.5000      1.2500
   5.0000      2.0000
```

CsInterpolate class

```
public class com.imsl.math.CsInterpolate extends com.imsl.math.Spline
```

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 ξ 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).

Fields

FIRST_DERIVATIVE

```
static final public int FIRST_DERIVATIVE
```

NOT_A_KNOT

```
static final public int NOT_A_KNOT
```

SECOND_DERIVATIVE

```
static final public 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
```

CsPeriodic class

```
public class com.imsl.math.CsPeriodic extends com.imsl.math.Spline
```

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 ξ be the breakpoint vector for the above spline interpolant. Then, the maximum absolute error satisfies

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

where

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

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

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
```

CsShape class

`public class com.imsl.math.CsShape extends com.imsl.math.Spline`

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.

Constructor

CsShape

```
public CsShape(double[] xData, double[] yData) throws
CsShape.TooManyIterationsException, 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
```

CsShape.TooManyIterationsException class

`static public class com.imsl.math.CsShape.TooManyIterationsException extends com.imsl.IMSLException`

Too many iterations.

Constructors

CsShape.TooManyIterationsException

`public CsShape.TooManyIterationsException()`

Description

Constructs a `TooManyIterationsException` object.

CsShape.TooManyIterationsException

`public CsShape.TooManyIterationsException(Object[] arguments)`

Description

Constructs a `TooManyIterationsException` object.

Parameter

`arguments` – an `Object` array containing arguments used within the error message string

CsShape.TooManyIterationsException

```
public CsShape.TooManyIterationsException(String key, Object[] arguments)
```

Description

Constructs a `TooManyIterationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

CsSmooth class

```
public class com.imsl.math.CsSmooth extends com.imsl.math.Spline
```

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

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

where σ 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 σ. `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 σ so that the smoothing spline (S_σ) 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).

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 0.10785822561423902
```

CsSmoothC2 class

`public class com.imsl.math.CsSmoothC2 extends com.imsl.math.Spline`

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_σ 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 σ 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).

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 0.06458434076780883
```

BSpline class

`abstract public class com.imsl.math.BSpline implements Serializable, Cloneable`

BSpline represents and evaluates univariate B-splines.

B-splines provide a particularly convenient and suitable basis for a given class of smooth ppoly functions. Such a class is specified by giving its breakpoint sequence, its order *k*, and the required smoothness across each of the interior breakpoints. The corresponding B-spline basis is specified by giving its knot sequence $\mathbf{t} \in \mathbf{R}^M$. The specification rule is as follows: If the class is to have all derivatives up to and including the j-th derivative continuous across the interior breakpoint ξ_i, then the number ξ_i should occur *k* - *j* - 1 times in the knot sequence. Assuming that ξ_1 and ξ_n are the endpoints of the interval of interest, choose the first *k* knots equal to ξ_1 and the last *k* knots equal to ξ_n. This can be done because the B-splines are defined to be right continuous near ξ_1 and left continuous near ξ_n.

When the above construction is completed, a knot sequence **t** of length *M* is generated, and there are *m*: = *M*-*k* B-splines of order *k*, for example $B_0, \ldots, B_{m-1}$, spanning the ppoly functions on the interval with the indicated smoothness. That is, each ppoly function in this class has a unique representation $p = a_0B_0 + a_1B_1 + \ldots + a_{m-1}B_{m-1}$ as a linear combination of B-splines. A B-spline is a particularly compact piecewise polynomial function. B_i is a nonnegative function that is nonzero only on the interval $[\mathbf{t}_i, \mathbf{t}_{i+k}]$. More precisely, the support of the i-th B-spline is $[t_i, t_{i+k}]$. No piecewise polynomial function in the same class (other than the zero function) has smaller support (i.e., vanishes on more intervals) than a B-spline. This makes B-splines particularly attractive basis functions since the influence of any particular B-spline coefficient extends only over a few intervals.

Fields

coef

```
protected double[] coef
```

The B-spline coefficient array.

knot

```
protected double[] knot
```

The knot array of length *n* + *order*, where *n* is the number of coefficients in the B-spline.

order

```
protected int order
```

Order of the spline.

Constructor

BSpline

```
public BSpline()
```

Methods

derivative

`public double derivative(double x)`

Description

Returns the value of the first derivative of the B-spline at a point.

Parameter

`x` – a `double` specifying a point at which the derivative is to be evaluated

Returns

a `double` containing the value of the first derivative of the B-spline at the point `x`

derivative

`public double derivative(double x, int ideriv)`

Description

Returns the value of the derivative of the B-spline at a point.

Parameters

`x` – a `double` specifying a 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 B-spline at the point `x`

derivative

`public double[] derivative(double[] x, int ideriv)`

Description

Returns the value of the derivative of the B-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 the B-spline at each point of the array `x`

getKnots

`public double[] getKnots()`

Description

Returns a copy of the knot sequence.

Returns

a double array containing a copy of the knot sequence.

getSpline

```
public Spline getSpline()
```

Description

Returns a Spline representation of the B-spline.

Returns

a Spline representation of the BSpline

integral

```
public double integral(double a, double b)
```

Description

Returns the value of an integral of the B-spline.

Parameters

a – a double specifying the lower limit of integration

b – a double specifying the upper limit of integration

Returns

a double which specifies the B-spline integral value from a to b

value

```
public double value(double x)
```

Description

Returns the value of the B-spline at a point.

Parameter

x – a double specifying the point at which the B-spline is to be evaluated

Returns

a double giving the value of the B-spline at the point x

value

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

Description

Returns the value of the B-spline at each point of an array.

Parameter

x – a double array of points at which the B-spline is to be evaluated

Returns

a double array containing the value of the B-spline at each point of the array x

BsInterpolate class

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

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, $B_{j-k+1}, \ldots, B_j$ where j satisfies $\mathbf{t}_j \leq x_i < \mathbf{t}_{j+1}$ be well-defined (that is, $j-k+1 \geq 0$).

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
```

BsLeastSquares class

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

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

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

Fields

nCoef

```
protected int nCoef
```

Number of B-spline coefficients.

weight

```
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[]) {
        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
```

Spline2D class

```
abstract public class com.imsl.math.Spline2D implements Serializable, Cloneable
```

Represents and evaluates tensor-product splines.

The simplest method of obtaining multivariate interpolation and approximation functions is to take univariate methods and form a multivariate method via tensor products. In the case of two-dimensional spline interpolation, the derivation proceeds as follows: Let t_x be a knot sequence for splines of order k_x,

and t_y be a knot sequence for splines of order k_y. Let $N_x + k_x$ be the length of t_x, and $N_y + k_x$ be the length of t_y. Then, the tensor-product spline has the following form:

$$\sum_{m=0}^{N_y-1} \sum_{n=0}^{N_x-1} c_{nm} B_{n,k_x,t_x}(x) B_{m,k_y,t_y}(y)$$

Given two sets of points

$$\{x_i\}_{i=1}^{N_x}$$

and

$$\{y_j\}_{j=1}^{N_y}$$

for which the corresponding univariate interpolation problem can be solved, the tensor-product interpolation problem finds the coefficients c_{nm} so that

$$\sum_{m=0}^{N_y-1} \sum_{n=0}^{N_x-1} c_{nm} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j) = f_{ij}$$

This problem can be solved efficiently by repeatedly solving univariate interpolation problems as described in de Boor (1978, p. 347).

Constructor

Spline2D

```
public Spline2D()
```

Methods

derivative

```
public double derivative(double x, double y, int xPartial, int yPartial)
```

Description

Returns the value of the partial derivative of the tensor-product spline at the point *(x, y)*.

Parameters

`x` – a `double` scalar specifying the *x*-coordinate of the evaluation point for the tensor-product spline.

`y` – a `double` scalar specifying the *y*-coordinate of the evaluation point for the tensor-product spline.

`xPartial` – an `int` scalar specifying the *x*-partial derivative.

`yPartial` – an `int` scalar specifying the *y*-partial derivative.

Returns

a `double` scalar containing the value of the partial derivative

$$\frac{\partial^{i+j} s}{\partial^i x \, \partial^j y}$$

where i = `xPartial` and j = `yPartial`, at (x, y).

derivative

```
public double[][] derivative(double[] xVec, double[] yVec, int xPartial, int
yPartial)
```

Description

Returns the values of the partial derivative of the tensor-product spline of an array of points.

Parameters

`xVec` – a `double` array specifying the x-coordinates at which the spline is to be evaluated.

`yVec` – a `double` array specifying the y-coordinates at which the spline is to be evaluated.

`xPartial` – an `int` scalar specifying the x-partial derivative.

`yPartial` – an `int` scalar specifying the y-partial derivative.

Returns

a `double` matrix containing the values of the partial derivatives

$$\frac{\partial^{i+j} s}{\partial^i x \, \partial^j y}$$

where i = `xPartial` and j = `yPartial`, at each (x, y).

getCoefficients

```
public double[][] getCoefficients()
```

Description

Returns the coefficients for the tensor-product spline.

Returns

a `double` matrix containing the coefficients.

getXKnots

```
public double[] getXKnots()
```

Description

Returns the knot sequences in the x-direction.

Returns

a `double` array containing the knot sequences of the spline in the x-direction.

getYKnots

```
public double[] getYKnots()
```

Description

Returns the knot sequences in the y-direction.

Returns

a `double` array containing the knot sequences of the spline in the y-direction.

integral

```
public double integral(double a, double b, double c, double d)
```

Description

Returns the value of an integral of a tensor-product spline on a rectangular domain.

If s is the spline, then the `integral` method returns

$$\int_a^b \int_c^d s(x,y)\,dy\,dx$$

This method uses the (univariate integration) identity (22) in de Boor (1978, p. 151)

$$\int_{t_0}^{x} \sum_{i=0}^{n-1} \alpha_i B_{i,k}(\tau)\,d\tau = \sum_{i=0}^{r-1} \left[\sum_{j=0}^{i} \alpha_j \frac{t_{j+k} - t_j}{k} \right] B_{i,k+1}(x)$$

where $t_0 \le x \le t_r$.

It assumes (for all knot sequences) that the first and last k knots are stacked, that is, $t_0 = \ldots = t_{k-1}$ and $t_n = \ldots = t_{n+k-1}$, where k is the order of the spline in the x or y direction.

Parameters

`a` – a `double` specifying the lower limit for the first variable of the tensor-product spline.

`b` – a `double` specifying the upper limit for the first variable of the tensor-product spline.

`c` – a `double` specifying the lower limit for the second variable of the tensor-product spline.

`d` – a `double` specifying the upper limit for the second variable of the tensor-product spline.

Returns

a `double`, the integral of the tensor-product spline over the rectangle `[a, b]` by `[c, d]`.

value

```
public double value(double x, double y)
```

Description

Returns the value of the tensor-product spline at the point *(x, y)*.

Parameters

`x` – a `double` scalar specifying the x-coordinate of the evaluation point for the tensor-product spline.

`y` – a `double` scalar specifying the y-coordinate of the evaluation point for the tensor-product spline.

Returns

a `double` scalar containing the value of the tensor-product spline.

value

```
public double[][] value(double[] xVec, double[] yVec)
```

Description

Returns the values of the tensor-product spline of an array of points.

Parameters

xVec – a `double` array specifying the x-coordinates at which the spline is to be evaluated.

yVec – a `double` array specifying the y-coordinates at which the spline is to be evaluated.

Returns

a `double` matrix containing the values evaluated.

Spline2DInterpolate class

```
public class com.imsl.math.Spline2DInterpolate extends com.imsl.math.Spline2D
```

Computes a two-dimensional, tensor-product spline interpolant from two-dimensional, tensor-product data.

The class `Spline2DInterpolate` computes a tensor-product spline interpolant. The tensor-product spline interpolant to data $\{(x_i, y_j, f_{ij})\}$, where $0 \le i \le (n_x - 1)$ and $0 \le j \le (n_y - 1)$ has the form

$$\sum_{m=0}^{n_y-1} \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,t_x}(x) B_{m,k_y,t_y}(y)$$

where k_x and k_y are the orders of the splines. These numbers are defaulted to be 4, but can be set to any positive integer using `xOrder` and `yOrder` in the constructor. Likewise, t_x and t_y are the corresponding knot sequences (`xKnots` and `yKnots`). These default values are selected by `Spline2DInterpolate`. The algorithm requires that

$$t_x(k_x - 1) \le x_i \le t_x(n_x) \qquad 0 \le i \le n_x - 1$$

$$t_y(k_y - 1) \le y_j \le t_y(n_y - 1) \qquad 0 \le j \le n_y - 1$$

Tensor-product spline interpolants in two dimensions can be computed quite efficiently by solving (repeatedly) two univariate interpolation problems.

The computation is motivated by the following observations. It is necessary to solve the system of equations

$$\sum_{m=0}^{n_y-1} \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,t_x}(x_i)$$

note that for each fixed i from 0 to $n_x - 1$, we have n_y linear equations in the same number of unknowns as can be seen below:

$$\sum_{m=0}^{n_y-1} h_{mi} B_{m,k_y,t_y}(y_j) = f_{ij}$$

$$\sum_{m=0}^{n_y-1} \sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,t_x}(x_i) B_{m,k_y,t_y}(y_j) = f_{ij}$$

Setting

$$h_{mi} = \sum_{n=0}^{n_x-1} c_{nm} B_{m,k_x,t_x}(x_i)$$

note that for each fixed i from 0 to $n_x - 1$, we have $n_y - 1$ linear equations in the same number of unknowns as can be seen below:

$$\sum_{m=0}^{n_y-1} h_{mi} B_{n,k_y,t_y}(y_j) = f_{ij}$$

The same matrix appears in all of the equations above:

$$\left[B_{m,k_y,t_y}(y_j)\right] \quad 0 \le m, j \le n_y - 1$$

Thus, only factor this matrix once and then apply this factorization to the n_x right-hand sides. Once this is done and h_{mi} is computed, then solve for the coefficients c_{nm} using the relation

$$\sum_{n=0}^{n_x-1} c_{nm} B_{n,k_x,t_x}(x_i) = h_{mi}$$

for n from 0 to $n_y - 1$, which again involves one factorization and n_y solutions to the different right-hand sides. The class `Spline2DInterpolate` is based on the routine SPLI2D by de Boor (1978, p. 347).

Constructors

Spline2DInterpolate

```
public Spline2DInterpolate(double[] xData, double[] yData, double[][] fData)
```

Description

Constructor for `Spline2DInterpolate`.

Parameters

`xData` – a `double` array containing the data points in the *x*-direction.

`yData` – a `double` array containing the data points in the *y*-direction.

`fData` – a `double` matrix of size `xData.length` by `yData.length` containing the values to be interpolated.

Spline2DInterpolate

```
public Spline2DInterpolate(double[] xData, double[] yData, double[][] fData,
int xOrder, int yOrder)
```

Description

Constructor for `Spline2DInterpolate`.

Parameters

`xData` – a `double` array containing the data points in the *x*-direction.

`yData` – a `double` array containing the data points in the *y*-direction.

`fData` – a `double` matrix of size `xData.length` by `yData.length` containing the values to be interpolated.

`xOrder` – an `int` scalar value specifying the order of the spline in the *x*-direction. `xOrder` must be at least 1. Default: `xOrder` = 4, tensor-product cubic spline.

`yOrder` – an `int` scalar value specifying the order of the spline in the *y*-direction. `yOrder` must be at least 1. Default: `yOrder` = 4, tensor-product cubic spline.

Spline2DInterpolate

```
public Spline2DInterpolate(double[] xData, double[] yData, double[][] fData,
int xOrder, int yOrder, double[] xKnots, double[] yKnots)
```

Description

Constructor for `Spline2DInterpolate`.

Parameters

`xData` – a `double` array containing the data points in the *x*-direction.

`yData` – a `double` array containing the data points in the *y*-direction.

`fData` – a `double` matrix of size `xData.length` by `yData.length` containing the values to be interpolated.

`xOrder` – an `int` scalar value specifying the order of the spline in the *x*-direction. `xOrder` must be at least 1. Default: `xOrder` = 4, tensor-product cubic spline.

`yOrder` – an `int` scalar value specifying the order of the spline in the *y*-direction. `yOrder` must be at least 1. Default: `yOrder` = 4, tensor-product cubic spline.

`xKnots` – a `double` array of size `xData.length + xOrder` specifying the knot sequences of the spline in the *x*-direction. Default knot sequences are selected by the class.

`yKnots` – a `double` array of size `yData.length + yOrder` specifying the knot sequences of the spline in the *y*-direction. Default knot sequences are selected by the class.

Example 1

A tensor-product spline interpolant to a function

$$f(x,y) = x^3 + y^2$$

is computed. The values of the interpolant and the error on a 4 x 4 grid are displayed.

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

public class Spline2DInterpolateEx1 {

    private static double F(double x, double y) {
        return (x * x * x + y * y);
    }

    public static void main(String args[]) {
        int nData = 11;
        int outData = 2;
        double[][] fData = new double[nData][nData];
        double[] xData = new double[nData];
        double[] yData = new double[nData];
        double x, y, z;

        // Set up grid
        for (int i = 0; i < nData; i++) {
            xData[i] = yData[i] = (double) i / ((double) (nData - 1));
        }

        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nData; j++) {
                fData[i][j] = F(xData[i], yData[j]);
            }
        }

        // Compute tensor-product interpolant
        Spline2DInterpolate spline
                = new Spline2DInterpolate(xData, yData, fData);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
        nf.setMinimumFractionDigits(4);

        // Print results
        System.out.println("   x        y      F(x, y)    "
                + "Interpolant   Error");
        for (int i = 0; i < outData; i++) {
            x = (double) i / (double) (outData);
            for (int j = 0; j < outData; j++) {
                y = (double) j / (double) (outData);
                z = spline.value(x, y);
                System.out.println(nf.format(x) + "    " + nf.format(y) + "    "
                        + nf.format(F(x, y)) + "        " + nf.format(z)
                        + "      " + nf.format(Math.abs(F(x, y) - z)));
            }
```

```
        }
    }
}
```

Output

```
   x        y     F(x, y)   Interpolant   Error
0.0000   0.0000   0.0000        0.0000     0.0000
0.0000   0.5000   0.2500        0.2500     0.0000
0.5000   0.0000   0.1250        0.1250     0.0000
0.5000   0.5000   0.3750        0.3750     0.0000
```

Example 2

A tensor-product spline interpolant to a function

$$f(x,y) = x^3 + y^2$$

is computed. The values of the interpolant and the error on a 4 x 4 grid are displayed. Notice that the first interpolant with order = 3 does not reproduce the cubic data, while the second interpolant with order = 6 does reproduce the data.

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

public class Spline2DInterpolateEx2 {

    private static double F(double x, double y) {
        return (x * x * x + y * y);
    }

    public static void main(String args[]) {
        int nData = 7;
        int outData = 4;
        double[][] fData = new double[nData][nData];
        double[] xData = new double[nData];
        double[] yData = new double[nData];
        double x, y, z;

        // Set up grid
        for (int i = 0; i < nData; i++) {
            xData[i] = yData[i] = (double) i / ((double) (nData - 1));
        }

        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nData; j++) {
                fData[i][j] = F(xData[i], yData[j]);
            }
        }

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
```

```
        nf.setMinimumFractionDigits(4);

        for (int order = 3; order < 7; order += 3) {
            // Compute tensor-product interpolant
            Spline2DInterpolate spline
                    = new Spline2DInterpolate(xData, yData,
                            fData, order, order);

            // Print results
            System.out.println("\nThe order of the spline is " + order);
            System.out.println("   x        y     F(x, y)   "
                    + "Interpolant   Error");

            for (int i = 0; i < outData; i++) {
                x = (double) i / (double) (outData);

                for (int j = 0; j < outData; j++) {
                    y = (double) j / (double) (outData);
                    z = spline.value(x, y);

                    System.out.println(nf.format(x) + "   " + nf.format(y)
                            + "   " + nf.format(F(x, y)) + "       "
                            + nf.format(z) + "     "
                            + nf.format(Math.abs(F(x, y) - z)));
                }
            }
        }
    }
}
```

Output

```
The order of the spline is 3
   x        y     F(x, y)   Interpolant   Error
0.0000   0.0000   0.0000       0.0000     0.0000
0.0000   0.2500   0.0625       0.0625     0.0000
0.0000   0.5000   0.2500       0.2500     0.0000
0.0000   0.7500   0.5625       0.5625     0.0000
0.2500   0.0000   0.0156       0.0158     0.0002
0.2500   0.2500   0.0781       0.0783     0.0002
0.2500   0.5000   0.2656       0.2658     0.0002
0.2500   0.7500   0.5781       0.5783     0.0002
0.5000   0.0000   0.1250       0.1250     0.0000
0.5000   0.2500   0.1875       0.1875     0.0000
0.5000   0.5000   0.3750       0.3750     0.0000
0.5000   0.7500   0.6875       0.6875     0.0000
0.7500   0.0000   0.4219       0.4217     0.0002
0.7500   0.2500   0.4844       0.4842     0.0002
0.7500   0.5000   0.6719       0.6717     0.0002
0.7500   0.7500   0.9844       0.9842     0.0002

The order of the spline is 6
   x        y     F(x, y)   Interpolant   Error
0.0000   0.0000   0.0000       0.0000     0.0000
```

```
0.0000   0.2500   0.0625        0.0625     0.0000
0.0000   0.5000   0.2500        0.2500     0.0000
0.0000   0.7500   0.5625        0.5625     0.0000
0.2500   0.0000   0.0156        0.0156     0.0000
0.2500   0.2500   0.0781        0.0781     0.0000
0.2500   0.5000   0.2656        0.2656     0.0000
0.2500   0.7500   0.5781        0.5781     0.0000
0.5000   0.0000   0.1250        0.1250     0.0000
0.5000   0.2500   0.1875        0.1875     0.0000
0.5000   0.5000   0.3750        0.3750     0.0000
0.5000   0.7500   0.6875        0.6875     0.0000
0.7500   0.0000   0.4219        0.4219     0.0000
0.7500   0.2500   0.4844        0.4844     0.0000
0.7500   0.5000   0.6719        0.6719     0.0000
0.7500   0.7500   0.9844        0.9844     0.0000
```

Example 3

A spline interpolant s to a function

$$f(x,y) = x^3y^2$$

is constructed. Then, the values of the partial derivative

$$\frac{\partial^3 s(x,y)}{\partial x^2 2y}$$

and the error are computed on a 4 x 4 grid.

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

public class Spline2DInterpolateEx3 {

    private static double F(double x, double y) {
        return (x * x * x * y * y);
    }

    private static double F21(double x, double y) {
        return (6.0 * x * 2.0 * y);
    }

    public static void main(String args[]) {
        int nData = 11;
        int outData = 2;
        double[][] fData = new double[nData][nData];
        double[] xData = new double[nData];
        double[] yData = new double[nData];
        double x, y, z;

        // Set up grid
        for (int i = 0; i < nData; i++) {
            xData[i] = yData[i] = (double) i / ((double) (nData - 1));
        }
```

```
        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nData; j++) {
                fData[i][j] = F(xData[i], yData[j]);
            }
        }

        // Compute tensor-product interpolant
        Spline2DInterpolate spline
                = new Spline2DInterpolate(xData, yData, fData);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
        nf.setMinimumFractionDigits(4);

        // Print results
        System.out.println("   x        y     F21(x, y)   "
                + "21InterpDeriv   Error");
        for (int i = 0; i < outData; i++) {
            x = (double) (1 + i) / (double) (outData + 1);

            for (int j = 0; j < outData; j++) {
                y = (double) (1 + j) / (double) (outData + 1);
                z = spline.derivative(x, y, 2, 1);
                System.out.println(nf.format(x) + "   " + nf.format(y) + "    "
                        + nf.format(F21(x, y)) + "        " + nf.format(z)
                        + "       " + nf.format(Math.abs(F21(x, y) - z)));
            }
        }
    }
}
```

Output

```
   x        y     F21(x, y)   21InterpDeriv   Error
0.3333   0.3333    1.3333        1.3333       0.0000
0.3333   0.6667    2.6667        2.6667       0.0000
0.6667   0.3333    2.6667        2.6667       0.0000
0.6667   0.6667    5.3333        5.3333       0.0000
```

Example 4

This example integrates a two-dimensional, tensor-product spline over the rectangle [0, x] by [0, y].

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

public class Spline2DInterpolateEx4 {

    // Define function
    private static double F(double x, double y) {
        return (x * x * x + y * y);
    }
```

```
    // The integral of F from 0 to x and 0 to y
    private static double FI(double x, double y) {
        return (y * x * x * x * x / 4.0 + x * y * y * y / 3.0);
    }

    public static void main(String args[]) {
        int nData = 11, outData = 2;
        double[][] fData = new double[nData][nData];
        double[] xData = new double[nData];
        double[] yData = new double[nData];
        double x, y, z;

        // Set up grid
        for (int i = 0; i < nData; i++) {
            xData[i] = yData[i] = (double) i / ((double) (nData - 1));
        }

        for (int i = 0; i < nData; i++) {
            for (int j = 0; j < nData; j++) {
                fData[i][j] = F(xData[i], yData[j]);
            }
        }

        // Compute tensor-product interpolant
        Spline2DInterpolate spline
                = new Spline2DInterpolate(xData, yData, fData);

        // Print results
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
        nf.setMinimumFractionDigits(4);

        System.out.println("   x       y     FI(x, y)  Integral   Error");
        for (int i = 0; i < outData; i++) {
            x = (double) (1 + i) / (double) (outData + 1);
            for (int j = 0; j < outData; j++) {
                y = (double) (1 + j) / (double) (outData + 1);
                z = spline.integral(0.0, x, 0.0, y);
                System.out.println(nf.format(x) + "  " + nf.format(y)
                        + "    " + nf.format(FI(x, y)) + "    "
                        + nf.format(z) + "    "
                        + nf.format(Math.abs(FI(x, y) - z)));
            }
        }
    }
}
```

Output

```
   x       y     FI(x, y)  Integral   Error
0.3333  0.3333    0.0051    0.0051    0.0000
0.3333  0.6667    0.0350    0.0350    0.0000
0.6667  0.3333    0.0247    0.0247    0.0000
0.6667  0.6667    0.0988    0.0988    0.0000
```

Spline2DLeastSquares class

```
public class com.imsl.math.Spline2DLeastSquares extends com.imsl.math.Spline2D
```

Computes a two-dimensional, tensor-product spline approximant using least squares.

The Spline2DLeastSquares class computes a tensor-product spline least-squares approximation to weighted tensor-product data. The input consists of data vectors to specify the tensor-product grid for the data, two vectors with the weights, the values of the surface on the grid, and the specification for the tensor-product spline. The grid is specified by the two vectors x = `xData` and y = `yData` of length n = `xData.length` and m = `yData.length`, respectively. A two-dimensional array f = `fData` contains the data values which are to be fit. The two vectors w_x = `xWeights` and w_y = `yWeights` contain the weights for the weighted least-squares problem. The information for the approximating tensor-product spline can be provided using the `setXOrder`, `setYOrder`, `setXKnots` and `setYKnots` methods. This information is contained in k_x = `xOrder`, t_x = `xKnots`, and N = `xSplineSpaceDim` for the spline in the first variable, and in k_y = `yOrder`, t_y = `yKnots` and M = `ySplineSpaceDim` for the spline in the second variable. This class computes coefficients for the tensor-product spline by solving the normal equations in tensor-product form as discussed in de Boor (1978, Chapter 17). The interested reader might also want to study the paper by Grosse (1980).

As the computation proceeds, we obtain coefficients c minimizing

$$\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} w_x(i)\, w_y(j) \left[\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} c_{kl} B_{kl}(x_i, y_i) - f_{ij}\right]^2$$

where the function B_{kl} is the tensor-product of two B-splines of order k_x and k_y. Specifically, we have

$$B_{kl}(x,y) = B_{k,k_x,t_x}(x)\, B_{l,k_y,t_y}(y)$$

The spline

$$\sum_{k=0}^{N-1}\sum_{l=0}^{M-1} c_{kl} B_{kl}$$

and its partial derivatives can be evaluated using the `value` method.

Constructor

Spline2DLeastSquares

```
public Spline2DLeastSquares(double[] xData, double[] yData, double[][] fData,
int xSplineSpaceDim, int ySplineSpaceDim)
```

Description

Constructor for `Spline2DLeastSquares`.

Parameters

`xData` – a `double` array containing the data points in the *x*-direction.

`yData` – a `double` array containing the data points in the *y*-direction.

`fData` – a `double` matrix of size `xData.length` by `yData.length` containing the values to be approximated.

`xSplineSpaceDim` – an `int` scalar value specifying the linear dimension of the spline subspace for the *x* variable. It should be smaller than `xData.length` and greater than or equal to `xOrder` (whose default value is 4).

`ySplineSpaceDim` – an `int` scalar value specifying the linear dimension of the spline subspace for the *y* variable. It should be smaller than `yData.length` and greater than or equal to `yOrder` (whose default value is 4).

Methods

---

compute

`public void compute()`

Description

Computes a two-dimensional, tensor-product spline approximant using least squares.

---

getErrorSumOfSquares

`public double getErrorSumOfSquares()`

Description

Returns the weighted error sum of squares.

Returns

a `double` scalar containing the weighted error sum of squares.

---

getXOrder

`public int getXOrder()`

Description

Returns the order of the spline in the *x*-direction.

Returns

an `int` scalar containing the order of the spline in the *x*-direction.

---

getXWeights

`public double[] getXWeights()`

Description

Returns the weights for the least-squares fit in the *x*-direction.

Returns

a `double` array containing the weights for the least-squares fit in the *x*-direction.

getYOrder

```
public int getYOrder()
```

Description

Returns the order of the spline in the *y*-direction.

Returns

an `int` scalar containing the order of the spline in the *y*-direction.

getYWeights

```
public double[] getYWeights()
```

Description

Returns the weights for the least-squares fit in the *y*-direction.

Returns

a `double` array containing the weights for the least-squares fit in the *y*-direction.

setXKnots

```
public void setXKnots(double[] xKnots)
```

Description

Sets the knot sequences of the spline in the *x*-direction.

Parameter

`xKnots` – a `double` array of size `xSplineSpaceDim + xOrder` specifying the knot sequences of the spline in the *x*-direction. Default knot sequences are selected by the class.

setXOrder

```
public void setXOrder(int xOrder)
```

Description

Sets the order of the spline in the *x*-direction.

Parameter

`xOrder` – an `int` scalar value specifying the order of the spline in the *x*-direction. `xOrder` must be at least 1. Default: `xOrder` = 4, implying a tensor-product cubic spline.

setXWeights

```
public void setXWeights(double[] xWeights)
```

Description

Sets the weights for the least-squares fit in the *x*-direction.

Parameter

`xWeights` – a `double` array of size `xData.length` specifying the weights for the least-squares fit in the *x*-direction. Default: all weights are equal to 1.

setYKnots

```
public void setYKnots(double[] yKnots)
```

Description

Sets the knot sequences of the spline in the *y*-direction.

Parameter

`yKnots` – a `double` array of size `ySplineSpaceDim + yOrder` specifying the knot sequences of the spline in the *y*-direction. Default knot sequences are selected by the class.

setYOrder

```
public void setYOrder(int yOrder)
```

Description

Sets the order of the spline in the *y*-direction.

Parameter

`yOrder` – an `int` scalar value specifying the order of the spline in the *y*-direction. `yOrder` must be at least 1. Default: `yOrder` = 4, implying a tensor-product cubic spline.

setYWeights

```
public void setYWeights(double[] yWeights)
```

Description

Sets the weights for the least-squares fit in the *y*-direction.

Parameter

`yWeights` – a `double` array of size `yData.length` specifying the weights for the least-squares fit in the *y*-direction. Default: all weights are equal to 1.

Example

The data for this example comes from the function $e^x sin(x+y)$ on the rectangle [0, 3] x [0, 5]. This function is first sampled on a 50 x 25 grid. Next, an attempt to recover it by using tensor-product cubic splines is performed. The values of the function $e^x sin(x+y)$ are printed on a 2 x 2 grid and compared with the values of the tensor-product spline least-squares fit.

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

public class Spline2DLeastSquaresEx1 {

    private static double F(double x, double y) {
        return (Math.exp(x) * Math.sin(x + y));
    }
```

```
public static void main(String args[]) {
    int nxData = 50, nyData = 25, outData = 2;
    double[] xData = new double[nxData];
    double[] yData = new double[nyData];
    double[][] fData = new double[nxData][nyData];
    double x, y, z;

    // Set up grid
    for (int i = 0; i < nxData; i++) {
        xData[i] = 3. * (double) i / ((double) (nxData - 1));
    }
    for (int i = 0; i < nyData; i++) {
        yData[i] = 5. * (double) i / ((double) (nyData - 1));
    }

    // Compute function values on grid
    for (int i = 0; i < nxData; i++) {
        for (int j = 0; j < nyData; j++) {
            fData[i][j] = F(xData[i], yData[j]);
        }
    }

    // Compute tensor-product approximant
    Spline2DLeastSquares spline
            = new Spline2DLeastSquares(xData, yData, fData, 5, 7);

    spline.compute();
    x = spline.getErrorSumOfSquares();

    NumberFormat nf = NumberFormat.getInstance();
    nf.setMaximumFractionDigits(4);
    nf.setMinimumFractionDigits(4);

    // Print results
    System.out.println("The error sum of squares is "
            + nf.format(x) + "\n");

    double[][] output = new double[outData * outData][5];
    for (int i = 0, idx = 0; i < outData; i++) {
        x = (double) i / (double) (outData);
        for (int j = 0; j < outData; j++) {
            y = (double) j / (double) (outData);
            z = spline.value(x, y);
            output[idx][0] = x;
            output[idx][1] = y;
            output[idx][2] = F(x, y);
            output[idx][3] = z;
            output[idx][4] = Math.abs(F(x, y) - z);
            idx++;
        }
    }

    String[] labels = {"x", "y", "F(x, y)", "Fitted Values", "Error"};
    PrintMatrixFormat pmf = new PrintMatrixFormat();
    pmf.setNumberFormat(nf);
```

```
        pmf.setColumnLabels(labels);
        new PrintMatrix().print(pmf, output);
    }
}
```

Output

```
The error sum of squares is 3.7532

     x       y      F(x, y)  Fitted Values  Error
0  0.0000  0.0000  0.0000      -0.0204     0.0204
1  0.0000  0.5000  0.4794       0.5002     0.0208
2  0.5000  0.0000  0.7904       0.8158     0.0253
3  0.5000  0.5000  1.3874       1.3842     0.0031
```

RadialBasis class

`public class com.imsl.math.RadialBasis implements Serializable, Cloneable`

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.

Two `update` methods allow the user to specify weights for each data point in the approximation scheme. In this way the user can influence the fit of the radial basis function. For example, if weights are in the range [0,1] then 0-weighted points are effectively removed from computations and 1-weighted points will have more influence than any others. When the number of centers equals the number of data points, the RBF fit will be "exact", otherwise it will be an approximation (useful for large or noisy data sets). Provided the ratios of the weights are not too extreme, weights will not appreciably change the accuracy of the fit to the data, but they will affect the shape of the approximating function away from the data: Greater weights result in greater influence at greater distances.

Constructor

RadialBasis

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

Description

Creates a new instance of RadialBasis.

Parameters

`nDim` – an `int` specifying the number of dimensions.

`nCenters` – an `int` specifying the number of centers.

Methods

getANOVA

```
public 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

a `RadialBasis.Function` which is the current radial function.

gradient

```
public double[] gradient(double[] x)
```

Description

Returns the gradient of the radial basis approximation at a point.

Parameter

`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.

Parameter

`radialFunction` – a `RadialBasis.Function` to be used in the approximation. The default is Hardy Multiquadric with $\delta = 1$.

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)
```

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 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.

---

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.

---

value

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

Description

Returns the value of the radial basis approximation at a point.

Parameter

`x` – is a `double` array containing the locations of the data point at which the approximation is to be computed.

Returns

a `double` containing 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.

Parameter

`x` – a `double` matrix of size *n* by *nDim* containing the points at which the radial basis is to be evaluated.

Returns

a `double` array giving the value of the radial basis at the point `x`

Example: Radial Basis Function Approximation

Data is generated from the function

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

where d is the dimension, is evaluated at a set of randomly chosen points. Random noise is added to the values and a radial basis approximation to the noisy data is computed. The radial basis fit, using the default radial basis function Hardy multiquadric with $\delta = 1$, is then compared to the original function at another set of randomly chosen 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 $[-1,1]^d$ hyper-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 $[-1,1]^d$ hyper-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);
        }
        aveError /= nTest;
```

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

    // The function to approximate
    static private 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 0.026192967462953198
maximum error is 0.13197595135821816
```

Example: "Custom" Radial Basis Function Approximation

Data is generated from the function

$$e^{\frac{y}{2.0}} \sin x \cos \frac{y}{2.0}$$

where a number of (x,y) pairs make up a set of randomly chosen points. Random noise is added to the values, a "custom" Polyharmonic Spline radial basis function is specified

$$\varphi(r) = \begin{cases} r^k & k = 1,3,5,\ldots \\ r^k \ln r & k = 2,4,6,\ldots \end{cases}$$

and a radial basis approximation of the noisy data is computed. The radial basis fit is then compared to the original function at another set of randomly chosen points. Both the average normalized error and the maximum normalized error are computed and printed.

In this example, the order of the Polyharmonic Spline, k=2. The function is sampled at 200 random points and the error is computed at 10000 random points.

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

public class RadialBasisEx2 {

    // The function to approximate
    static private double fcn(double x[]) {
        return Math.exp((x[1]) / 2.0) * Math.sin(x[0]) - Math.cos((x[1]) / 2.0);
    }

    static public class PolyHarmonicSpline implements RadialBasis.Function {
```

```
    private int order = 3;
    private boolean isEven = false;

    public PolyHarmonicSpline(int order) {
        this.isEven = order % 2 == 0;
        this.order = order;
    }

    public double f(double x) {
        if (this.isEven) {
            return Math.pow(x, order) * Math.log(x);
        }
        return Math.pow(x, order);
    }

    public double g(double x) {
        if (order == 1) {
            return 1;
        }
        if (this.isEven) {
            return order * Math.pow(x, order - 1) * Math.log(x)
                    + Math.pow(x, order - 1);
        }
        return order * Math.pow(x, order - 1);
    }

    public int getOrder() {
        return order;
    }

    public boolean isEvenOrder() {
        return isEven;
    }
}

public static void main(String args[]) {
    int nDim = 2;

    // 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(123457);
    rand.setMultiplier(16807);
    double noise[] = new double[nData * nDim];
    for (int k = 0; k < nData; k++) {
        for (int i = 0; i < nDim; i++) {
            noise[k * 2 + i] = 1.0d - 2.0d * rand.nextDouble();
            xData[k][i] = 3 * noise[k * 2 + i];
        }
        // noisy sample
        fData[k] = fcn(xData[k]) + noise[k * 2] / 10;
    }

    // Compute the radial basis approximation using 100 centers
    int nCenters = 100;
```

```
        RadialBasis rb = new RadialBasis(nDim, nCenters);
        rb.setRadialFunction(new PolyHarmonicSpline(2));
        rb.update(xData, fData);

        // Compute the error at a randomly selected set of points
        int nTest = 10000;
        double maxError = 0.0;
        double aveError = 0.0;
        double maxMagnitude = 0.0;
        double x[][] = new double[nTest][nDim];
        noise = new double[nTest * nDim];

        for (int i = 0; i < nTest; i++) {
            for (int j = 0; j < nDim; j++) {
                noise[i * 2 + j] = 1.0d - 2.0d * (double) rand.nextDouble();
                x[i][j] = 3 * noise[i * 2 + j];
            }
            double error = Math.abs(fcn(x[i]) - rb.value(x[i]));
            maxMagnitude = Math.max(Math.abs(fcn(x[i])), maxMagnitude);
            aveError += error;
            maxError = Math.max(error, maxError);
        }
        aveError /= nTest;

        System.out.println("Average normalized error is "
                + aveError / maxMagnitude);
        System.out.println("Maximum normalized error is "
                + maxError / maxMagnitude);
        System.out.println("Using even order equation: "
                + ((PolyHarmonicSpline) rb.getRadialFunction()).isEvenOrder());
    }
}
```

Output

```
Average normalized error is 0.01805587205103261
Maximum normalized error is 0.2575653108755503
Using even order equation: true
```

Example: Multiquadric Radial Basis Function Approximation

Data is generated from the function

$$e^{\frac{y}{2.0}} \sin x \cos \frac{y}{2.0}$$

where a number of (x,y) pairs make up a set of randomly chosen points. Random noise is added to the values, a Hardy multiquadric radial basis function is specified $\sqrt{r^2+\delta^2}$ and a radial basis approximation of the noisy data is computed. The radial basis fit is then compared to the original function at another set of randomly chosen points. Both the average normalized error and the maximum normalized error are computed and printed.

In this example, the parameter of the Hardy multiquadric radial basis function $\delta = 5.5$. The function is sampled at 100 random points and the error is computed at 10000 random points.

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

public class RadialBasisEx3 {

    public static void main(String args[]) {
        int nDim = 2;

        // Sample, with noise, the function at 100 randomly choosen points
        int nData = 100;
        double xData[][] = new double[nData][nDim];
        double fData[] = new double[nData];
        Random rand = new Random(123457);
        rand.setMultiplier(16807);
        double noise[] = new double[nData * nDim];
        for (int k = 0; k < nData; k++) {
            for (int i = 0; i < nDim; i++) {
                noise[k * 2 + i] = 1.0d - 2.0d * (double) rand.nextDouble();
                xData[k][i] = 3 * noise[k * 2 + i];
            }
            // noisy sample
            fData[k] = fcn(xData[k]) + noise[k * 2] / 10;
        }

        // Compute the radial basis approximation using 100 centers
        int nCenters = 100;
        RadialBasis rb = new RadialBasis(nDim, nCenters);
        rb.setRadialFunction(new RadialBasis.HardyMultiquadric(5.5));
        rb.update(xData, fData);

        // Compute the error at a randomly selected set of points
        int nTest = 10000;
        double maxError = 0.0;
        double aveError = 0.0;
        double maxMagnitude = 0.0;
        double x[][] = new double[nTest][nDim];
        noise = new double[nTest * nDim];

        for (int i = 0; i < nTest; i++) {
            for (int j = 0; j < nDim; j++) {
                noise[i * 2 + j] = 1.0d - 2.0d * rand.nextDouble();
                x[i][j] = 3 * noise[i * 2 + j];
            }
            double error = Math.abs(fcn(x[i]) - rb.value(x[i]));
            maxMagnitude = Math.max(Math.abs(fcn(x[i])), maxMagnitude);
            aveError += error;
            maxError = Math.max(error, maxError);
        }
        aveError /= nTest;

        System.out.println("Average normalized error is "
                + aveError / maxMagnitude);
        System.out.println("Maximum normalized error is "
                + maxError / maxMagnitude);
    }
```

```
    // The function to approximate
    static private double fcn(double x[]) {
        return Math.exp((x[1]) / 2.0) * Math.sin(x[0]) - Math.cos((x[1]) / 2.0);
    }
}
```

Output

```
Average normalized error is 0.011085255049181973
Maximum normalized error is 0.05481720536391322
```

Example: Gaussian Radial Basis Function Approximation

Data is generated from the function

$$e^{\frac{y}{2.0}} \sin x \cos \frac{y}{2.0}$$

where a number of (x,y) pairs make up a set of randomly chosen points. Random noise is added to the values, a Gaussian radial basis function is specified e^{-ax^2} and a radial basis approximation of the noisy data is computed. The radial basis fit is then compared to the original function at another set of randomly chosen points. Both the average normalized error and the maximum normalized error are computed and printed.

In this example, the parameter of the Gaussian radial basis function $a = 0.1$. The function is sampled at 100 random points and the error is computed at 10000 random points.

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

public class RadialBasisEx4 {

    public static void main(String args[]) {
        int nDim = 2;

        // Sample, with noise, the function at 100 randomly choosen points
        int nData = 100;
        double xData[][] = new double[nData][nDim];
        double fData[] = new double[nData];
        Random rand = new Random(123457);
        rand.setMultiplier(16807);
        double noise[] = new double[nData * nDim];
        for (int k = 0; k < nData; k++) {
            for (int i = 0; i < nDim; i++) {
                noise[k * 2 + i] = 1.0d - 2.0d * (double) rand.nextDouble();
                xData[k][i] = 3 * noise[k * 2 + i];
            }
            // noisy sample
            fData[k] = fcn(xData[k]) + noise[k * 2] / 10;
        }

        // Compute the radial basis approximation using 100 centers
        int nCenters = 100;
```

```
        RadialBasis rb = new RadialBasis(nDim, nCenters);
        rb.setRadialFunction(new RadialBasis.Gaussian(.1));
        rb.update(xData, fData);

        // Compute the error at a randomly selected set of points
        int nTest = 10000;
        double maxError = 0.0;
        double aveError = 0.0;
        double maxMagnitude = 0.0;
        double x[][] = new double[nTest][nDim];
        noise = new double[nTest * nDim];

        for (int i = 0; i < nTest; i++) {
            for (int j = 0; j < nDim; j++) {
                noise[i * 2 + j] = 1.0d - 2.0d * rand.nextDouble();
                x[i][j] = 3 * noise[i * 2 + j];
            }
            double error = Math.abs(fcn(x[i]) - rb.value(x[i]));
            maxMagnitude = Math.max(Math.abs(fcn(x[i])), maxMagnitude);
            aveError += error;
            maxError = Math.max(error, maxError);
        }
        aveError /= nTest;

        System.out.println("Average normalized error is "
                + aveError / maxMagnitude);
        System.out.println("Maximum normalized error is "
                + maxError / maxMagnitude);
    }

    // The function to approximate
    static private double fcn(double x[]) {
        return Math.exp((x[1]) / 2.0) * Math.sin(x[0]) - Math.cos((x[1]) / 2.0);
    }
}
```

Output

```
Average normalized error is 0.01095472360691534
Maximum normalized error is 0.02300757841627234
```

RadialBasis.Function interface

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

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

Methods

f

```
public double f(double x)
```

Description

A radial basis function.

Parameter

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

Returns

a `double`, the value of the function at x

g

```
public double g(double x)
```

Description

The derivative of the radial basis function used to calculate the `gradient` of the radial basis approximation.

Parameter

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

Returns

a `double`, the value of the function at x

RadialBasis.HardyMultiquadric class

```
static public class com.imsl.math.RadialBasis.HardyMultiquadric implements
com.imsl.math.RadialBasis.Function
```

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

Constructor

RadialBasis.HardyMultiquadric

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

Description

Creates a Hardy multiquadric basis function $\sqrt{r^2+\delta^2}$.

Parameter

`delta` – a `double` specifying the value of the function parameter. Increasing the multiquadric parameter decreases fitting-error but generally increases computational effort. The default value is 1.0.

Methods

f

```
public double f(double x)
```

Description

A Hardy multiquadric basis function.

Parameter

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

Returns

a `double`, the value of the function at x

g

```
public double g(double x)
```

Description

The derivative of the Hardy multiquadric basis function used to calculate the `gradient` of the radial basis approximation.

Parameter

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

Returns

a `double`, the value of the function at x

RadialBasis.Gaussian class

```
static public class com.imsl.math.RadialBasis.Gaussian implements
com.imsl.math.RadialBasis.Function
```

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

Constructor

RadialBasis.Gaussian

`public RadialBasis.Gaussian(double a)`

Description

Creates a Gaussian basis function e^{-ax^2}.

Parameter

`a` – a `double` specifying the value of the function parameter. Decreasing the Gaussian parameter decreases fitting-error but may increase computational effort.

Methods

f

`public double f(double x)`

Description

A Gaussian basis function.

Parameter

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

Returns

a `double`, the value of the function at x

g

`public double g(double x)`

Description

The derivative of the Gaussian basis function used to calculate the `gradient` of the radial basis approximation.

Parameter

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

Returns

a `double`, the value of the function at x

Chapter 5: Quadrature

Types

class Quadrature 218
class HyperRectangleQuadrature 225

Usage Notes

Univariate Quadrature

Class Quadrature computes approximations to integrals of the form

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

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.

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

Return Values from User-Supplied Functions

All values returned by user-supplied functions must be valid real numbers. It is the user's responsibility to check that the values returned by a user-supplied function do not contain NaN, infinity, or negative infinity values.

Example: Minimum of a smooth function

The minimum of $e^x - 5x$ is found using function evaluations only.

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

public class OptimizationIntroEx1 {
    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) {
                double y = Math.exp(x) - 5.*x;
                if(!Double.isNaN(y)) {
                       return y;
                } else {
                       return 0.0;
                }
            }
        };
        System.out.println("Minimum is " + zf.computeMin(fcn));
    }
}
```

Quadrature class

`public class com.imsl.math.Quadrature implements Serializable, Cloneable`

`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).

If the function to be integrated has endpoint singularities then extrapolation should be enabled. As described above, the integral's value is approximated by applying a quadrature rule to a series of subdivisions of the interval. The sequence of approximate values converges to the integral's value. The ε-algorithm can be used to extrapolate from the initial terms of the sequence to its limit. Without extrapolation, the quadrature approximation sequence converges slowly if the function being integrated has endpoint singularities. The ε-algorithm accelerates convergence of the sequence in this case. The class `com.imsl.math.EpsilonAlgorithm` (p. 580) implements the ε-algorithm. With extrapolation, this class is similar to the subroutine `QAGS` by Piessens et al. (1983).

The desired absolute error, ε, can be set using `setAbsoluteError`. The desired relative error, ρ, can be

set using `setRelativeError`. The method `eval` computes the approximate integral value $R \approx \int_a^b f(x)dx$. It also computes an error estimate E, which can be retrieved using `getErrorEstimate`. These are related by the following equation:

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

Constructor

Quadrature

`public Quadrature()`

Description

Constructs a Quadrature object.

Methods

eval

`public 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
0	No error.
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 underestimated.
3	Extremely bad integrand behavior occurs at some points of the integration interval.
4	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 which can be obtained.
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 void setAbsoluteError(double errorAbsolute)`

Description

Sets the absolute error tolerance.

Parameter

`errorAbsolute` – a `double` scalar value specifying the absolute error tolerance. The default value is 1.0536712127723714E-8.

setExtrapolation

`public void setExtrapolation(boolean doExtrapolation)`

Description

If true, the epsilon-algorithm for extrapolation is enabled. The default is false (extrapolation is not used).

Parameter

`doExtrapolation` – a `boolean`, true if the epsilon-algorithm for extrapolation is to be enabled, false otherwise

setMaxSubintervals

```
public void setMaxSubintervals(int maxSubintervals)
```

Description

Sets the maximum number of subintervals allowed. The default value is 500.

Parameter

`maxSubintervals` – an `int` specifying the maximum number of subintervals to be allowed. The default is 500.

setRelativeError

```
public void setRelativeError(double errorRelative)
```

Description

Sets the relative error tolerance.

Parameter

`errorRelative` – a `double` scalar value specifying the relative error tolerance. The default value is 1.0536712127723714E-8.

setRule

```
public 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.

Parameter

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

Example 1: Integral of exp(2x)

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(2.0 * x);
            }
        };

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

        double expect = (Math.exp(6) - Math.exp(2)) / 2.0;
        System.out.println("result = " + result);
        System.out.println("expect = " + expect);
    }
}
```

Output

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

Example 2: Integral of exp(-x) from 0 to infinity

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
```

Quadrature.Function interface

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

Public interface function for the Quadrature class.

Method

f

`public double f(double x)`

Description

Returns the value of the function at the given point.

Parameter

`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

HyperRectangleQuadrature class

```
public class com.imsl.math.HyperRectangleQuadrature implements Serializable,
Cloneable
```

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` (p. 1349) .

Constructors

HyperRectangleQuadrature

```
public HyperRectangleQuadrature(RandomSequence sequence)
```

Description

Constructs a HyperRectangleQuadrature object.

HyperRectangleQuadrature

```
public HyperRectangleQuadrature(int dim)
```

Description

Constructs a HyperRectangleQuadrature object.

Methods

eval

`public double eval(HyperRectangleQuadrature.Function objectF)`

Description

Returns the value of the integral over the unit cube.

Parameter

`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 void setAbsoluteError(double errorAbsolute)`

Description

Sets the absolute error tolerance.

Parameter

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

setRelativeError

`public void setRelativeError(double errorRelative)`

Description

Sets the relative error tolerance.

Parameter

`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
```

HyperRectangleQuadrature.Function interface

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

Public interface function for the HyperRectangleQuadrature class.

Method

f

```
public double f(double[] x)
```

Description

Returns the value of the function at the given point.

Parameter

`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`

Chapter 6: Differential Equations

Types

class ODE 230
class OdeRungeKutta 236
class OdeAdamsGear 240
class FeynmanKac 251

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 classes `OdeRungeKutta` and `OdeAdamsGear` solve the IVP for ODEs of the form

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

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,

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

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. In such cases, use `OdeAdamsGear`. For more discussion about stiff systems, see Gear (1971, Chapter 11) or Shampine and Gear (1979).

Partial Differential Equations

The `FeynmanKac` class solves the Feynman-Kac equation, a single partial differential equation, on a finite interval $[x_{\min}, x_{\max}]$. This equation often arises in applications from financial engineering and that is the primary focus of the documentation examples. The equation, initial conditions and Feynman-Kac boundary values are given by

$$f_t + \mu(x,t) + \frac{\sigma^2(x,t)}{2} f_{xx} - \kappa(x,t) f = \phi(f,x,t),$$

$$f(x,T) = p(x), \{f_t = \frac{\partial f}{\partial t}, \text{etc.}\},$$

$$a(x,t)f + b(x,t)f_x + c(x,t)f_{xx} = d(x,t),\ x = x_{\min} \& x_{\max}.$$

The solution is approximated by a piece-wise series of Hermite quintic polynomials on a grid of the interval $[x_{\min}, x_{\max}]$ that yields a twice differentiable solution.
To assist in the evaluation of the approximate solution and its derivatives there is method `getSplineValue`.

ODE class

```
abstract public class com.imsl.math.ODE implements Serializable, Cloneable
```

ODE represents and solves an initial-value problem for ordinary differential equations.

Fields

AFTER_SUCCESSFUL_STEP

```
static final public int AFTER_SUCCESSFUL_STEP
```

Used by method `examineStep` to indicate examining after a successful step

AFTER_UNSUCCESSFUL_STEP

```
static final public int AFTER_UNSUCCESSFUL_STEP
```

Used by method `examineStep` to indicate examining after an unsuccessful step

BEFORE_STEP

```
static final public int BEFORE_STEP
```

Used by method `examineStep` to indicate examining before the next step

ERROR_NORM_ABS

```
static final public int ERROR_NORM_ABS
```

Used by method `setNorm` to indicate that the error norm to be used is to be the absolute error, equals $max(|e_i|)$

ERROR_NORM_EUCLIDEAN

```
static final public int ERROR_NORM_EUCLIDEAN
```

Used by method `setNorm` to indicate that the error norm to be used is to be the scaled Euclidean norm defined as

$$s = \sqrt{\sum_{i=1}^{neq} \frac{{e_i}^2}{{w_i}^2}}$$

where $w_i = e_i/max(|y_i(t)|, 1.0)$ and neq is the number of equations

ERROR_NORM_MAX

```
static final public int ERROR_NORM_MAX
```

Used by method `setNorm` to indicate that the error norm to be used is to be the maximum of $e_i/max(|y_i(t)|, floor)$ where `floor` is set via `setFloor`

ERROR_NORM_MINABSREL

```
static final public int ERROR_NORM_MINABSREL
```

Used by method `setNorm` to indicate that the error norm to be used is to be the minimum of the absolute error and the relative error, equals the maximum of $e_i/max(|y_i(t)|, 1)$

Constructor

ODE

```
public ODE()
```

Methods

examineStep

```
protected void examineStep(int state, double t, double[] y)
```

Description

Called before and after each internal step. This method can be over-ridden by the user to examine intermediate values of `t` and `y`.

Parameters

`state` – an `int`, one of `BEFORE_STEP`, `AFTER_SUCCESSFUL_STEP` or `AFTER_UNSUCCESSFUL_STEP`.

`t` – `double` representing the independent variable.

`y` – `double` array containing the dependent variables.

getFloor

```
public double getFloor()
```

Description

Returns the value used in the norm computation.

Returns

a `double` used in the norm computation.

getInitialStepsize

```
public double getInitialStepsize()
```

Description

Returns the initial internal step size.

Returns

a `double` specifying the initial internal step size.

getMaxSteps

```
public int getMaxSteps()
```

Description

Returns the maximum number of internal steps allowed.

Returns

an `int` specifying the maximum number of internal steps allowed.

getMaximumStepsize

```
public double getMaximumStepsize()
```

Description

Returns the maximum internal step size.

Returns

a `double` specifying the maximum internal step size.

getMinimumStepsize

```
public double getMinimumStepsize()
```

Description

Returns the minimum internal step size.

Returns

a `double` specifying the minimum internal step size.

getNorm

```
public int getNorm()
```

Description

Returns the switch for determining the error norm.

Returns

an `int` specifying the switch for determining the error norm. In the following, e_i is the absolute value for an estimate of the error in $y_i(t)$.

normMethod	Constraint
ERROR_NORM_MINABSREL	Minimum of the absolute error and the relative error, equals the maximum of $e_i/max(\|y_i(t)\|, 1)$
ERROR_NORM_ABS	Absolute error, equals $max(e_i)$
ERROR_NORM_MAX	Maximum of $e_i/max(\|y_i(t)\|, floor)$
ERROR_NORM_EUCLIDEAN	Scaled Euclidean norm defined as $$s = \sqrt{\sum_{i=1}^{neq} \frac{{e_i}^2}{{w_i}^2}}$$ where $w_i = e_i/max(\|y_i(t)\|, 1.0)$ and neq is the number of equations

getScale

```
public double getScale()
```

Description

Returns the scaling factor.

Returns

a `double` specifying the scaling factor.

getTolerance

```
public double getTolerance()
```

Description

Returns the error tolerance.

Returns

a `double` specifying the error tolerance.

setFloor

```
public void setFloor(double floor)
```

Description

Sets the value used in the norm computation.

Parameter

`floor` – a `double` used in the norm computation. `floor` must be greater than zero.
Default: `floor` = 1.0

setInitialStepsize

```
public void setInitialStepsize(double stepsize)
```

Description

Sets the initial internal step size.

Parameter

`stepsize` – a `double` specifying the initial internal step size. `stepsize` must be greater than or equal to zero. Default: `stepsize` = 0.0

setMaxSteps

```
public void setMaxSteps(int maxSteps)
```

Description

Sets the maximum number of internal steps allowed.

Parameter

`maxSteps` – an `int` specifying the maximum number of internal steps allowed. `maxSteps` must be greater than zero. Default: `maxSteps` = 500

setMaximumStepsize

```
public void setMaximumStepsize(double stepsize)
```

Description

Sets the maximum internal step size.

Parameter

`stepsize` – a `double` specifying the maximum internal step size. `stepsize` must be greater than zero.
Default: See the `setMaximumStepsize` method in the subclasses for the default values used.

setMinimumStepsize

```
public void setMinimumStepsize(double stepsize)
```

Description

Sets the minimum internal step size.

Parameter

`stepsize` – a `double` specifying the minimum internal step size. `stepsize` must be greater than or equal to zero. Default: `stepsize` = 0.0

setNorm

```
public void setNorm(int normMethod)
```

Description

Sets the switch for determining the error norm.

Parameter

`normMethod` – an `int` specifying the switch for determining the error norm.
Default: `normMethod` = `ERROR_NORM_MINABSREL`
In the following table, e_i is the absolute value for an estimate of the error in $y_i(t)$.

normMethod	Constraint
`ERROR_NORM_MINABSREL`	Minimum of the absolute error and the relative error, equals the maximum of $e_i/max(\lvert y_i(t)\rvert, 1)$
`ERROR_NORM_ABS`	Absolute error, equals $max(e_i)$
`ERROR_NORM_MAX`	Maximum of $e_i/max(\lvert y_i(t)\rvert, floor)$
`ERROR_NORM_EUCLIDEAN`	Scaled Euclidean norm defined as $$s = \sqrt{\sum_{i=1}^{neq} \frac{{e_i}^2}{{w_i}^2}}$$ where $w_i = e_i/max(\lvert y_i(t)\rvert, 1.0)$ and neq is the number of equations

setScale

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

Description

Sets the scaling factor.

Parameter

scale – a double specifying the scaling factor. scale must be greater than zero.
Default: scale = 1.0

setTolerance

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

Description

Sets the error tolerance.

Parameter

tolerance – a double specifying the error tolerance. tolerance must be greater than zero.
Default: tolerance = 1.0e-6

vnorm

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

Description

Returns the norm of a vector. This method can be over-ridden by the user to supply a different norm than those available through setNorm.

Parameters

v – a double array containing the vector whose norm is to be computed

y – a double array containing the values of the dependent variable

ymax – a double array containing the maximum y values computed thus far

Returns

a double scalar value representing the norm of the vector v

OdeRungeKutta class

```
public class com.imsl.math.OdeRungeKutta extends com.imsl.math.ODE
```

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 $\frac{dy}{dt} = y' = f(t,y)$ with given initial data. The class attempts to keep the global error proportional to a user-specified tolerance. This class 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.

Constructor

OdeRungeKutta

`public OdeRungeKutta(OdeRungeKutta.Function function)`

Description

Constructs an ODE solver to solve the initial value problem *dy/dt = f(t,y)*

Parameter

`function` – implementation of interface Function that defines the right-hand side function *f(t,y)*

Methods

setMaximumStepsize

`public void setMaximumStepsize(double stepsize)`

Description

Sets the maximum internal step size.

Parameter

`stepsize` – a `double` specifying the maximum internal step size. `stepsize` must be greater than zero. Default: `stepsize` = 2

solve

```
public void solve(double t, double tEnd, double[] y) throws
OdeRungeKutta.ToleranceTooSmallException, OdeRungeKutta.DidNotConvergeException
```

Description

Integrates the ODE system from `t` to `tEnd`. On all but the first call to solve, the value of `t` must equal the value of `tEnd` from the previous call.

Parameters

`t` – `double` specifying the independent variable

`tEnd` – `double` specifying the value of `t` at which the solution is desired

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

Exceptions

`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.

`ToleranceTooSmallException` is thrown if the computation does not converge on some step.

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}
```

OdeRungeKutta.Function interface

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

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

Method

f

`public void f(double t, double[] y, double[] yprime)`

Description

Returns the value of the function at the given point.

Parameters

`t` – 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 (`t`,`y`)

OdeRungeKutta.ToleranceTooSmallException class

```
static public class com.imsl.math.OdeRungeKutta.ToleranceTooSmallException
extends com.imsl.IMSLException
```

Tolerance is too small or the problem is stiff.

Constructors

OdeRungeKutta.ToleranceTooSmallException

```
public OdeRungeKutta.ToleranceTooSmallException(String message)
```

Description

Constructs a `ToleranceTooSmallException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameter

`message` – a `String` containing the error message

OdeRungeKutta.ToleranceTooSmallException

```
public OdeRungeKutta.ToleranceTooSmallException(String key, Object[] arguments)
```

Description

Constructs a `ToleranceTooSmallException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

OdeRungeKutta.DidNotConvergeException class

`static public class com.imsl.math.OdeRungeKutta.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge within the maximum number of steps allowed (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.

Constructors

OdeRungeKutta.DidNotConvergeException

`public OdeRungeKutta.DidNotConvergeException(String message)`

Description

Constructs a `DidNotConvergeException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameter

`message` – a `String` containing the error message

OdeRungeKutta.DidNotConvergeException

`public OdeRungeKutta.DidNotConvergeException(String key, Object[] arguments)`

Description

Constructs a `DidNotConvergeException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

OdeAdamsGear class

`public class com.imsl.math.OdeAdamsGear extends com.imsl.math.ODE`

Extension of the ODE class to solve a stiff initial-value problem for ordinary differential equations using the Adams-Gear methods.

Class `OdeAdamsGear` finds an approximation to the solution of a system of first-order differential equations of the form

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

with given initial conditions for y at the starting value for t. The class attempts to keep the global error proportional to a user-specified tolerance. The proportionality depends on the differential equation and the range of integration. The code is based on using backward difference formulas not exceeding order five as outlined in Gear (1971) and implemented by Hindmarsh (1974). There is an optional use of the code that employs implicit Adams formulas. This use is intended for nonstiff problems with expensive functions $y' = f(t, y)$.

Fields

METHOD_ADAMS

```
static final public int METHOD_ADAMS
```

The Adams integration method

METHOD_BDF

```
static final public int METHOD_BDF
```

The BDF integration method

SOLVE_CHORD_COMPUTED_DIAGONAL

```
static final public int SOLVE_CHORD_COMPUTED_DIAGONAL
```

A chord method and a diagonal matrix based on a directional directive

SOLVE_CHORD_COMPUTED_JACOBIAN

```
static final public int SOLVE_CHORD_COMPUTED_JACOBIAN
```

A chord or modified Newton method and a divided differences Jacobian

SOLVE_CHORD_USER_JACOBIAN

```
static final public int SOLVE_CHORD_USER_JACOBIAN
```

A chord or modified Newton method and a user-supplied Jacobian

SOLVE_FUNCTION_ITERATION

```
static final public int SOLVE_FUNCTION_ITERATION
```

A function iteration or successive substitution method

Constructor

OdeAdamsGear

```
public OdeAdamsGear(OdeAdamsGear.Function function)
```

Description

Constructs an ODE solver to solve the initial value problem $dy/dt = f(t,y)$

Parameter

`function` – implementation of interface Function that defines the right-hand side function $f(t,y)$

Methods

getIntegrationMethod

```
public int getIntegrationMethod()
```

Description

Returns the integration method used.

Returns

an `int` indicating the integration method used. One of the following is returned:

Value	Description
`METHOD_ADAMS`	Use the implicit Adams method.
`METHOD_BDF`	Use backward differentiation formula (BDF) methods.

getMaxOrder

```
public int getMaxOrder()
```

Description

Returns the highest order formula to use of implicit `METHOD_ADAMS` type or `METHOD_BDF` type.

Returns

an `int` specifying the highest order formula to use of implicit `METHOD_ADAMS` type or `METHOD_BDF` type.

getMaximumFunctionEvaluations

```
public int getMaximumFunctionEvaluations()
```

Description

Returns the maximum number of function evaluations of y' allowed.

Returns

an `int` specifying the maximum number of function evaluations of y' allowed.

getNumberOfFcnEvals

```
public int getNumberOfFcnEvals()
```

Description

Returns the number of function evaluations of y' made.

Returns

an `int` specifying the number of function evaluations of y' made.

getNumberOfJacobianEvals

```
public int getNumberOfJacobianEvals()
```

Description

Returns the number of Jacobian matrix evaluations used.

Returns

an `int` specifying the number of Jacobian matrix evaluations used.

getNumberOfSteps

```
public int getNumberOfSteps()
```

Description

Returns the number of internal steps taken.

Returns

an `int` specifying the number of internal steps taken.

getSolveMethod

```
public int getSolveMethod()
```

Description

Returns the method for solving the formula equations.

Returns

an `int` indicating the method used for solving the formula equations. One of the following is returned:

Value	Description
SOLVE_FUNCTION_ITERATION	Use a function iteration or successive substitution method.
SOLVE_CHORD_USER_JACOBIAN	Use a chord or modified Newton method and a user-supplied Jacobian.
SOLVE_CHORD_COMPUTED_JACOBIAN	Use a chord or modified Newton method and a Jacobian approximated by divided differences.
SOLVE_CHORD_COMPUTED_DIAGONAL	Use a chord method and a diagonal matrix based on a directional directive.

setIntegrationMethod

```
public void setIntegrationMethod(int intMethod)
```

Description

Indicates which integration method is to be used.

Parameter

`intMethod` – an `int` specifying the integration method to be used. `intMethod` must be one of the values specified in the table which follows. Default: `intMethod` = `METHOD_BDF`

intMethod	Description
METHOD_ADAMS	Use the implicit Adams method.
METHOD_BDF	Use backward differentiation formula (BDF) methods.

setMaxOrder

```
public void setMaxOrder(int maxOrder)
```

Description

Sets the highest order formula to use of implicit `METHOD_ADAMS` type or `METHOD_BDF` type.

Parameter

`maxOrder` – an `int` specifying the highest order formula to use of implicit `METHOD_ADAMS` type or `METHOD_BDF` type. `maxOrder` must be greater than zero. Default: `maxOrder` = 12 for `METHOD_ADAMS` and `maxOrder` = 5 for `METHOD_BDF`.

setMaximumFunctionEvaluations

```
public void setMaximumFunctionEvaluations(int maxfcn)
```

Description

Sets the maximum number of function evaluations of y' allowed.

Parameter

`maxfcn` – an `int` specifying the maximum number of function evaluations of y' allowed. `maxfcn` must be greater than zero. Default: No limit is enforced.

setMaximumStepsize

```
public void setMaximumStepsize(double stepsize)
```

Description

Sets the maximum internal step size.

Parameter

`stepsize` – a `double` specifying the maximum internal step size. `stepsize` must be greater than zero. Default: `stepsize` = 1.7976931348623157e+308

setSolveMethod

```
public void setSolveMethod(int solveMethod)
```

Description

Indicates which method to use for solving the formula equations.

Parameter

solveMethod – an int specifying the method to be used for solving the formula equations. Note that if the problem is stiff and a chord or modified Newton method is most efficient, use SOLVE_CHORD_USER_JACOBIAN or SOLVE_CHORD_COMPUTED_JACOBIAN. solveMethod must be one of the values specified in the table which follows.
Default: solveMethod = SOLVE_CHORD_COMPUTED_JACOBIAN.

solveMethod	Description
SOLVE_FUNCTION_ITERATION	Use a function iteration or successive substitution method.
SOLVE_CHORD_USER_JACOBIAN	Use a chord or modified Newton method and a user-supplied Jacobian.
SOLVE_CHORD_COMPUTED_JACOBIAN	Use a chord or modified Newton method and a divided differences Jacobian.
SOLVE_CHORD_COMPUTED_DIAGONAL	Use a chord method and a diagonal matrix based on a directional directive.

solve

```
public void solve(double t, double tEnd, double[] y) throws
OdeAdamsGear.DidNotConvergeException,
OdeAdamsGear.MaxFcnEvalsExceededException,
OdeAdamsGear.ToleranceTooSmallException, OdeAdamsGear.SingularMatrixException
```

Description

Integrates the ODE system from t to tEnd. On all but the first call to solve, the value of t must equal the value of tEnd from the previous call.

Parameters

t – a double specifying the independent variable

tEnd – a double specifying the value of t at which the solution is desired

y – on input, a double array containing the initial values. On output, a double array containing the approximate solution.

Exceptions

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.

ToleranceTooSmallException is thrown if the computation does not converge on some step.

MaxFcnEvalsExceededException is thrown if the maximum number of function evaluations allowed has been exceeded.

SingularMatrixException is thrown if the factorization function encounters a singular matrix during LU decomposition.

Example: Adams-Gear ordinary differential equation solver

A mildly stiff ordinary differential equation problem is solved using a solver which implements the Adams-Gear method. The solution at time t=240 is printed.

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

public class OdeAdamsGearEx1 {

    public static void main(String args[]) throws com.imsl.IMSLException {
        final double k1 = 294.0;
        final double k2 = 3.0;
        final double k3 = 0.01020408;

        OdeAdamsGear.Function f = new OdeAdamsGear.Function() {
            public double[] f(double t, double y[]) {
                double[] yprime = new double[y.length];
                yprime[0] = -y[0] - y[0] * y[1] + k1 * y[1];
                yprime[1] = -k2 * y[1] + k3 * (1.0 - y[1]) * y[0];
                return yprime;
            }
        };

        double t = 0.0;
        double tend = 240.0;
        double y[] = {1.0, 0.0};
        OdeAdamsGear q = new OdeAdamsGear(f);
        q.setNorm(OdeAdamsGear.ERROR_NORM_ABS);
        q.setSolveMethod(OdeAdamsGear.SOLVE_CHORD_COMPUTED_JACOBIAN);
        q.setTolerance(1.e-3);
        q.solve(t, tend, y);

        // Print Results
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
        nf.setMinimumFractionDigits(4);

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

Output

```
Result = {0.3924, 0.0013}
```

OdeAdamsGear.Function interface

`public interface com.imsl.math.OdeAdamsGear.Function`

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

Method

f

`public double[] f(double t, double[] y)`

Description

Computes the value of the function $y^{'} = f(t,y)$ at the given point.

Parameters

`t` – a `double`, the point at which the function is to be evaluated. (Input)

`y` – a `double` array which contains the dependent variable values. (Input)

Returns

a `double` array of length `y.length` which contains the value of the function $\frac{dy}{dt} = y^{'} = f(t,y)$. (Output)

OdeAdamsGear.Jacobian interface

`public interface com.imsl.math.OdeAdamsGear.Jacobian implements com.imsl.math.OdeAdamsGear.Function`

Public interface for the user supplied function to evaluate the Jacobian matrix.

Method

jacobian

`public double[][] jacobian(double t, double[] y, double[] yprime)`

Description

Used to compute the Jacobian of the function at `t`.

Parameters

`t` – a `double`, the point at which the function is to be evaluated. (Input)

`y` – a `double` array which contains the dependent variable values. (Input)

`yprime` – a `double` array which contains the value of the function $\frac{dy}{dt} = y' = f(t,y)$. (Input)

Returns

a `double` `y.length` by `y.length` matrix containing the value of the Jacobian of the function at `t`. (Output)

OdeAdamsGear.ToleranceTooSmallException class

```
static public class com.imsl.math.OdeAdamsGear.ToleranceTooSmallException
extends com.imsl.IMSLException
```

Tolerance is too small or the problem is stiff.

Constructors

OdeAdamsGear.ToleranceTooSmallException

```
public OdeAdamsGear.ToleranceTooSmallException(String message)
```

Description

Constructs a `ToleranceTooSmallException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameter

`message` – a `String` containing the error message

OdeAdamsGear.ToleranceTooSmallException

```
public OdeAdamsGear.ToleranceTooSmallException(String key, Object[] arguments)
```

Description

Constructs a `ToleranceTooSmallException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

OdeAdamsGear.MaxFcnEvalsExceededException class

`static public class com.imsl.math.OdeAdamsGear.MaxFcnEvalsExceededException extends com.imsl.IMSLException`

Maximum function evaluations exceeded.

Constructors

OdeAdamsGear.MaxFcnEvalsExceededException

`public OdeAdamsGear.MaxFcnEvalsExceededException(String message)`

Description

Constructs a `MaxFcnEvalsExceededException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameter

`message` – a `String` containing the error message

OdeAdamsGear.MaxFcnEvalsExceededException

`public OdeAdamsGear.MaxFcnEvalsExceededException(String key, Object[] arguments)`

Description

Constructs a `MaxFcnEvalsExceededException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

OdeAdamsGear.DidNotConvergeException class

`static public class com.imsl.math.OdeAdamsGear.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge within the maximum number of steps allowed (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.

Constructors

OdeAdamsGear.DidNotConvergeException

```
public OdeAdamsGear.DidNotConvergeException(String message)
```

Description

Constructs a `DidNotConvergeException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameter

`message` – a `String` containing the error message

OdeAdamsGear.DidNotConvergeException

```
public OdeAdamsGear.DidNotConvergeException(String key, Object[] arguments)
```

Description

Constructs a `DidNotConvergeException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

OdeAdamsGear.SingularMatrixException class

```
static public class com.imsl.math.OdeAdamsGear.SingularMatrixException extends
com.imsl.IMSLException
```

The interpolation matrix is singular. This exception is thrown if the factorization function encounters a singular matrix during LU decomposition.

Constructors

OdeAdamsGear.SingularMatrixException

```
public OdeAdamsGear.SingularMatrixException(String message)
```

Description

Constructs a `SingularMatrixException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameter

`message` – a `String` containing the error message

OdeAdamsGear.SingularMatrixException

```
public OdeAdamsGear.SingularMatrixException(String key, Object[] arguments)
```

Description

Constructs a `SingularMatrixException` with the specified detailed message. The detailed message is a `String` that describes this particular exception.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac class

```
public class com.imsl.math.FeynmanKac implements Serializable, Cloneable
```

Solves the generalized Feynman-Kac PDE.

Class `FeynmanKac` solves the generalized Feynman-Kac PDE on a rectangular grid with boundary conditions using a finite element Galerkin method. The generalized Feynman-Kac differential equation has the form

$$f_t + \mu(x,t) f_x + \frac{\sigma^2(x,t)}{2} f_{xx} - \kappa(x,t) f = \phi(f,x,t),$$

where the initial data satisfies $f(x,T) = p(x)$. The derivatives are $f_t = \frac{\partial f}{\partial t}$, etc. Method `computeCoefficients` uses a finite element Galerkin method over the rectangle

$$[x_{\min}, x_{\max}] \times [\bar{T}, T]$$

in (x,t) to compute the approximate solution. The interval $[x_{\min}, x_{\max}]$ is decomposed with a grid

$$(x_{\min} =) x_1 < x_2 < \ldots < x_m (= x_{\max}).$$

On each subinterval the solution is represented by

$$f(x,t) = f_i b_0(z) + f_{i+1} b_0(1-z) + h_i f_i' b_1(z) - h_i f_{i+1}' b_1(1-z) + h_i^2 f_i'' b_2(z) + h_i^2 f_{i+1}'' b_2(1-z).$$

The values

$$f_i, f_i', f_i'', f_{i+1}, f_{i+1}', f_{i+1}''$$

are time-dependent coefficients associated with each interval. The basis functions b_0, b_1, b_2 are given for

$$x \in [x_i, x_{i+1}],\ h_i = x_{i+1} - x_i,\ z = (x - x_i)/h_i,\ z \in [0,1],$$

by

$$\begin{aligned} b_0(z) &= -6z^5 + 15z^4 - 10z^3 + 1 = (1-z)^3(6z^2 + 3z + 1), \\ b_1(z) &= -3z^5 + 8z^4 - 6z^3 + z = (1-z)^3 z(3z+1), \\ b_2(z) &= \frac{1}{2}(-z^5 + 3z^4 - 3z^3 + z^2) = \frac{1}{2}(1-z)^3 z^2. \end{aligned}$$

By adding the piece-wise definitions the unknown solution function may be arranged as the series

$$f(x,t) = \sum_{i=1}^{3m} y_i \beta_i(x),\ x \in [x_{\min}, x_{\max}],$$

where the time-dependent coefficients are defined by re-labeling:

$$y_{3i-2} = f_i,\ y_{3i-1} = f_i',\ y_{3i} = f_i'',\ i = 1, \ldots, m.$$

The Galerkin principle is then applied. Using the provided initial and boundary conditions leads to an Index 1 differential-algebraic equation for the coefficients y_i, $i = 1, \ldots, 3m$.

This system is integrated using the variable order, variable step algorithm `DDASLX` noted in Hanson, R. and Krogh, F. (2008), *Solving Constrained Differential-Algebraic Systems Using Projections*. Solution values and their time derivatives at a grid preceding time T, expressed in units of time remaining, are given back by methods `getSplineCoefficients` and `getSplineCoefficientsPrime`, respectively. For further details of deriving and solving the system see Hanson, R. (2008), *Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements*. To evaluate f or its partials $f_x, f_{xx}, f_{xxx}, f_t, f_{tx}, f_{txx}, f_{txxx}$ at any time point in the grid, use method `getSplineValue`.

One useful application of the `FeynmanKac` class is financial analytics. This is illustrated in Example 2, which solves a diffusion model for call options which, in the special case $\alpha = 2$ reduces to the Black-Scholes (BS) model. Another useful application for the `FeynmanKac` class is the calculation of the Greeks, i.e. various derivatives of Feynman-Kac solutions applicable to, e.g., the pricing of options and related financial derivatives. In Example 5, the `FeynmanKac` class is used to calculate eleven of the Greeks for the same diffusion model introduced in Example 2 in the special BS case. These Greeks are also calculated using the BS closed form Greek equations (see http://en.wikipedia.org/wiki/The_Greeks). The Feynman-Kac and BS solutions are output and compared. Example 5 illustrates that the `FeynmanKac` class can be used to explore the Greeks for a much wider class of financial models than can BS.

Fields

METHOD_OF_PETZOLD

```
static final public int METHOD_OF_PETZOLD
```

Used by method `setStepControlMethod` to indicate that the step control algorithm of the original Petzold code is used in the integration.

METHOD_OF_SOEDERLIND

```
static final public int METHOD_OF_SOEDERLIND
```

Used by method `setStepControlMethod` to indicate that the step control method by Soederlind is used in the integration.

Constructor

FeynmanKac

```
public FeynmanKac(FeynmanKac.PdeCoefficients pdeCoeffs)
```

Description

Constructs a PDE solver to solve the Feynman-Kac PDE.

Parameter

`pdeCoeffs` – Implementation of interface PdeCoefficients that computes the values of the Feynman-Kac coefficients at a given point `(t,x)`.

Methods

computeCoefficients

```
public void computeCoefficients(int numLeftBounds, int numRightBounds,
FeynmanKac.Boundaries pdeBounds, double[] xGrid, double[] tGrid) throws
FeynmanKac.ToleranceTooSmallException, FeynmanKac.TooManyIterationsException,
FeynmanKac.ErrorTestException, FeynmanKac.CorrectorConvergenceException,
FeynmanKac.IterationMatrixSingularException,
FeynmanKac.TimeIntervalTooSmallException,
FeynmanKac.TcurrentTstopInconsistentException, FeynmanKac.TEqualsToutException,
FeynmanKac.InitialConstraintsException,
FeynmanKac.ConstraintsInconsistentException, SingularMatrixException,
FeynmanKac.BoundaryInconsistentException
```

Description

Determines the coefficients of the Hermite quintic splines that represent an approximate solution for the Feynman-Kac PDE.

Parameters

`numLeftBounds` – an `int`, the number of left boundary conditions. It is required that $1 \leq$ `numLeftBounds` ≤ 3.

`numRightBounds` – an `int`, the number of right boundary conditions. It is required that $1 \leq$ `numRightBounds` ≤ 3.

`pdeBounds` – Implementation of interface Boundaries that computes the boundary coefficients and terminal condition for given `(t,x)`.

`xGrid` – a `double` array containing the breakpoints for the Hermite quintic splines used in the `x` discretization. The length of `xGrid` must be at least 2, `xGrid.length >'= 2`, and the elements in `xGrid` must be in strictly increasing order.

`tGrid` – a `double` array containing the set of time points (in time-remaining units) where an approximate solution is returned. The elements in array `tGrid` must be positive and in strictly increasing order.

Exceptions

`ToleranceTooSmallException` is thrown if the absolute or relative error tolerances used in the integrator are too small.

`TooManyIterationsException` is thrown if the integrator needs too many iteration steps.

`ErrorTestException` is thrown if the error test used in the integrator failed repeatedly.

`CorrectorConvergenceException` is thrown if the corrector failed to converge repeatedly.

`IterationMatrixSingularException` thrown if one of the iteration matrices used in the integrator is singular.

`TimeIntervalTooSmallException` is thrown if the distance between an intermediate starting and end point for the integration is too small.

`TcurrentTstopInconsistentException` is thrown if during the integration the current integration time and given stepsize is inconsistent with the endpoint of the integration.

`TEqualsToutException` is thrown if during the integration process the actual integration time and the end time of the integration are identical.

`InitialConstraintsException` is thrown if at the initial integration point some of the constraints are inconsistent.

`ConstraintsInconsistentException` is thrown if during the integration process the constraints for the actual time point and given stepsize are inconsistent.

`SingularMatrixException` is thrown if one of the matrices used outside the integrator is singular.

`BoundaryInconsistentException` is thrown if the boundary conditions are inconsistent.

getAbsoluteErrorTolerances

```
public double[] getAbsoluteErrorTolerances()
```

Description

Returns absolute error tolerances.

Returns

a `double` array of length `3*xGrid.length` containing absolute error tolerances for the solutions.

getGaussLegendreDegree

```
public int getGaussLegendreDegree()
```

Description

Returns the number of quadrature points used in the Gauss-Legendre quadrature formula.

Returns

an `int`, the degree of the polynomial used in the Gauss-Legendre quadrature.

getInitialStepsize

```
public double getInitialStepsize()
```

Description

Returns the starting step size for the integration.

Returns

a `double`, the starting step size used in the integrator.

getMaxSteps

```
public int getMaxSteps()
```

Description

Returns the maximum number of internal steps allowed.

Returns

an `int` specifying the maximum number of internal steps allowed between each output point of the integration.

getMaximumBDFOrder

```
public int getMaximumBDFOrder()
```

Description

Returns the maximum order of the BDF formulas.

Returns

an `int`, the maximum order of the backward differentiation formulas used in the integrator.

getMaximumStepsize

```
public double getMaximumStepsize()
```

Description

Returns the maximum internal step size used by the integrator.

Returns

a `double`, the maximum internal step size.

getRelativeErrorTolerances

```
public double[] getRelativeErrorTolerances()
```

Description

Returns relative error tolerances.

Returns

a `double` array of length `3*xGrid.length` containing relative error tolerances for the solutions.

getSplineCoefficients

```
public double[][] getSplineCoefficients()
```

Description

Returns the coefficients of the Hermite quintic splines that represent an approximate solution of the Feynman-Kac PDE.

Returns

a `double` array of dimension `(tGrid.length+1) by (3*xGrid.length)` containing the coefficients of the Hermite quintic spline representation of the approximate solution for the Feynman-Kac PDE at time points `0, tGrid[0],...,tGrid[tGrid.length-1]`. Setting `ntGrid = tGrid.length` and `nxGrid = xGrid.length` the approximate solution is given by

$$f(x,t) = \sum_{j=0}^{3*\text{nxGrid}-1} y_{ij}\beta_j(x) \text{ for } t = \text{tGrid}[i-1], i = 1,\ldots,\text{ntGrid}.$$

The representation for the initial data at `t=0` is

$$p(x) = \sum_{j=0}^{3*\text{nxGrid}-1} y_{0j}\beta_j(x)\,.$$

The `(ntGrid+1) by (3*nxGrid)` matrix

$$(y_{ij})_{i=0,\ldots,\text{ntGrid}}^{j=0,\ldots,3*\text{nxGrid}-1}$$

is stored row-wise in the returned array.

getSplineCoefficientsPrime

```
public double[][] getSplineCoefficientsPrime()
```

Description

Returns the first derivatives of the Hermite quintic spline coefficients that represent an approximate solution of the Feynman-Kac PDE.

Returns

a `double` array of dimension `(tGrid.length+1) by (3*xGrid.length)` containing the first derivatives (in time) of the coefficients of the Hermite quintic spline representation of the approximate solution for the Feynman-Kac PDE at time points `0, tGrid[0],...,tGrid[tGrid.length-1]`. The approximate solution itself is given by

$$f_t(x,\bar{t}) = \sum_{j=0}^{3*\text{nxGrid}-1} y'_{ij}\beta_j(x) \quad \text{for } \bar{t} = \text{tGrid}[i-1], i = 1,\ldots,\text{ntGrid},$$

and

$$f_t(x,\bar{t}) = \sum_{j=0}^{3*\text{nxGrid}-1} y'_{0j}\beta_j(x) \quad \text{for } \bar{t} = 0\,.$$

The (ntGrid+1) by (3*nxGrid) matrix

$$(y'_{ij})_{i=0,\ldots,\text{ntGrid}}^{j=0,\ldots,3*\text{nxGrid}-1}$$

is stored row-wise in the returned array.

getSplineValue

```
public double[] getSplineValue(double[] evaluationPoints, double[]
coefficients, int ideriv)
```

Description

Evaluates for time value 0 or a time value in tGrid the derivative of the Hermite quintic spline interpolant at evaluation points within the range of `xGrid`.

Parameters

`evaluationPoints` – a `double` array containing the points in `x`-direction at which the Hermite quintic spline representing the approximate solution to the Feynman-Kac PDE or one of its derivatives is to be evaluated. It is required that all elements in array `evaluationPoints` are greater than or equal to `xGrid[0]` and less than or equal to `xGrid[xGrid.length-1]`.

`coefficients` – a `double` array of length `3*xGrid.length` containing the coefficients of the Hermite quintic spline representing the approximate solution f or f_t to the Feynman-Kac PDE. These coefficients are the rows of the arrays `splineCoeffs` and `splineCoeffsPrime` returned by methods `getSplineCoefficients` and `getSplineCoefficientsPrime`. If the user wants to compute approximate solutions f or f_x, f_{xx}, f_{xxx} to the Feynman-Kac PDE at time point `0`, one must assign row `splineCoeffs[0]` to array `coefficients`. If the user wants to compute these approximate solutions for time points `t=tGrid[i], i=0,...,tGrid.length-1`, one must assign row `splineCoeffs[i+1]` to array `coefficients`. The same reasoning applies to the computation of approximate solutions f_t and $f_{tx}, f_{txx}, f_{txxx}$ and assignment of rows of array `getSplineCoefficientsPrime` to array `coefficients`.

`ideriv` – an `int` specifying the derivative to be computed. It must be 0, 1, 2 or 3.

Returns

a `double` array containing the derivative of order `ideriv` of the Hermite quintic spline representing the approximate solution f or f_t to the Feynman Kac PDE at `evaluationPoints`. If `ideriv=0`, then the spline values are returned. If `ideriv=1`, then the first derivative is returned, etc.

getStepControlMethod

```
public int getStepControlMethod()
```

Description

Returns the step control method used in the integration of the Feynman-Kac PDE.

Returns

an `int` scalar specifying which step control method to be used.

Return Value	*Description*
METHOD_OF_SOEDERLIND	Method of Soederlind
METHOD_OF_PETZOLD	Method from the original Petzold code DASSL

getTimeBarrier

```
public double getTimeBarrier()
```

Description

Returns the barrier set for integration in the time direction.

Returns

a `double`, the time point beyond which the integrator should not integrate.

setAbsoluteErrorTolerances

```
public void setAbsoluteErrorTolerances(double atol)
```

Description

Sets the absolute error tolerances.

Parameter

`atol` – a `double` scalar specifying the absolute error tolerances for the row-wise solutions returned by method `getSplineCoefficients`. The tolerance value `atol` is applied to all `3*xGrid.length` solution components. `atol` must be greater than or equal to zero. Also, not all entries in `atol` and `rtol` are allowed to be equal to 0 simultaneously. Default value is 1.0e-5 for each solution component.

setAbsoluteErrorTolerances

```
public void setAbsoluteErrorTolerances(double[] atol)
```

Description

Sets the absolute error tolerances.

Parameter

`atol` – a `double` array of length `3*xGrid.length` specifying the absolute error tolerances for the row-wise solutions returned by method `getSplineCoefficients`. All entries in `atol` must be greater than or equal to zero. Also, not all entries in `atol` and `rtol` are allowed to be equal to 0 simultaneously. Default value is 1.0e-5 for each solution component.

setForcingTerm

```
public void setForcingTerm(FeynmanKac.ForcingTerm forceTerm)
```

Description

Sets the user-supplied method that computes approximations to the forcing term $\phi(x)$ and its derivative $\partial\phi/\partial y$ used in the FeynmanKac PDE.

Parameter

`forceTerm` – a `ForcingTerm` interface specifying the user defined function used for computation of the forcing term $\phi(f,x,t)$ and its derivative $\partial\phi/\partial y$. If this member function is not called it is assumed that $\phi(f,x,t)$ is identically zero.

setGaussLegendreDegree

```
public void setGaussLegendreDegree(int degree)
```

Description

Sets the number of quadrature points used in the Gauss-Legendre quadrature formula.

Parameter

`degree` – an `int`, the degree of the polynomial used in the Gauss-Legendre quadrature. It is required that `degree` is greater than or equal to 6. The default value is 6.

setInitialData

```
public void setInitialData(FeynmanKac.InitialData initData)
```

Description

Sets the user-supplied method for adjustment of initial data or as an opportunity for output during the integration steps.

Parameter

`initData` – an `InitialData` object specifying the user-defined function for adjustment of initial data or the object can be used as an opportunity for output during the integration steps. If this member function is not called, no adjustment of initial data or output during the integration steps is done.

setInitialStepsize

```
public void setInitialStepsize(double initStepsize)
```

Description

Sets the starting stepsize for the integration.

Parameter

`initStepsize` – a `double`, the starting stepsize used by the integrator. Must be less than zero since the integration is internally done from `t=0` to `t=tGrid[tGrid.length-1]` in a negative direction. The default is `initStepsize = -1.1102230246252e-16`.

setMaxSteps

```
public void setMaxSteps(int maxSteps)
```

Description

Sets the maximum number of internal steps allowed.

Parameter

`maxSteps` – an `int` specifying the maximum number of internal steps allowed between each output point of the integration. `maxSteps` must be positive. The default value is 500000.

setMaximumBDFOrder

```
public void setMaximumBDFOrder(int maxBDFOrder)
```

Description

Sets the maximum order of the BDF formulas.

Parameter

maxBDFOrder – an int specifying the maximum order of the backward differentiation formulas used in the integrator. It is required that maxBDFOrder is greater than zero and smaller than 6. The default is maxBDFOrder = 5.

setMaximumStepsize

```
public void setMaximumStepsize(double maximumStepsize)
```

Description

Sets the maximum internal step size used by the integrator.

Parameter

maximumStepsize – a positive scalar of type double, the maximum internal step size. Default value is Double.MAX_VALUE, the largest possible machine number.

setRelativeErrorTolerances

```
public void setRelativeErrorTolerances(double rtol)
```

Description

Sets the relative error tolerances.

Parameter

rtol – a double scalar specifying the relative error tolerances for the solution. The tolerance value rtol is applied to all 3*xGrid.length solution components. rtol must be greater than or equal to zero. Also, not all entries in atol and rtol are allowed to be equal to 0 simultaneously. Default value is 1.0e-5 for each solution component.

setRelativeErrorTolerances

```
public void setRelativeErrorTolerances(double[] rtol)
```

Description

Sets the relative error tolerances.

Parameter

rtol – a double array of length 3*xGrid.length specifying the relative error tolerances for the solution. All entries in rtol must be greater than or equal to zero. Also, not all entries in atol and rtol are allowed to be equal to 0 simultaneously. Default value is 1.0e-5 for each solution component.

setStepControlMethod

```
public void setStepControlMethod(int stepControlMethod)
```

Description

Sets the step control method used in the integration of the Feynman-Kac PDE.

Parameter

`stepControlMethod` – an `int` scalar specifying the step control method to be used in the integration. If this member function is not called `stepControlMethod` is set to `METHOD_OF_SOEDERLIND` by default.

`stepControlMethod`	*Description*
`METHOD_OF_SOEDERLIND`	Use method of Soederlind.
`METHOD_OF_PETZOLD`	Use method from the original Petzold code DASSL.

setTimeBarrier

`public void setTimeBarrier(double timeBarrier)`

Description

Sets a barrier for the integration in the time direction.

Parameter

`timeBarrier` – a `double`, controls whether the integrator should integrate in the time direction beyond a special point, `timeBarrier`, and then interpolate to get the Hermite quintic spline coefficients and its derivatives at the points in `tGrid`. It is required that `timeBarrier` be greater than or equal to `tGrid[tGrid.length-1]`. The default is `timeBarrier = tGrid[tGrid.length-1]`.

setTimeDependence

`public void setTimeDependence(boolean[] timeFlag)`

Description

Sets the time dependence of the coefficients, boundary conditions and function ϕ in the Feynman Kac equation.

Parameter

`timeFlag` – a `boolean` vector of length 7 indicating time dependencies in the Feynman-Kac PDE.

Index	*Time dependency of*
0	σ'
1	σ
2	μ
3	κ
4	Left boundary conditions
5	Right boundary conditions
6	ϕ

`timeFlag[i] = true` indicates that the associated value is time-dependent, `timeFlag[i] = false` indicates that the associated value is not time-dependent. By default, `timeFlag[i] = false` for $i = 0, \ldots, 6$.

Example 1: American Option vs. European Option on a Vanilla Put

The value of the American Option on a Vanilla Put can be no smaller than its European counterpart. That is due to the American Option providing the opportunity to exercise at any time prior to expiration. This example compares this difference - or premium value of the American Option - at two time values using the Black-Scholes model. The example is based on Wilmott et al. (1996, p. 176), and uses the non-linear forcing or weighting term described in Hanson, R. (2008), *Integrating Feynman-Kac Equations Using Hermite Quintic Finite Elements*, for evaluating the price of the American Option. The coefficients, payoff, boundary conditions and forcing term for American and European options are defined through interfaces `PdeCoefficients`, `Boundaries` and `ForcingTerm`, respectively. One breakpoint is set exactly at the strike price. The sets of parameters in the computation are:

1. Strike price $K = 10.0$
2. Volatility $\sigma = 0.4$
3. Times until expiration = $\{1/4, 1/2\}$
4. Interest rate $r = 0.1$
5. $x_{\min} = 0.0, x_{\max} = 30.0$
6. $nx = 61, n = 3 \times nx = 183$

The payoff function is the "vanilla option", $p(x) = \max(K - x, 0)$.

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

public class FeynmanKacEx1 {

    public static void main(String args[]) throws Exception {
        // Compute American Option Premium for Vanilla Put
        // The strike price
        double KS = 10.0;
        // The sigma value
        double sigma = 0.4;
        // Time values for the options
        int nt = 2;
        double[] tGrid = {0.25, 0.5};
        // Values of the underlying where evaluations are made
        double[] evaluationPoints = {0.0, 2.0, 4.0, 6.0, 8.0, 10.0,
            12.0, 14.0, 16.0};
        // Value of the interest rate
        double r = 0.1;
        // Values of the min and max underlying values modeled
        double xMin = 0.0, xMax = 30.0;
        // Define parameters for the integration step.
        int nxGrid = 61;
        int nv = 9;
        int nint = nxGrid - 1, n = 3 * nxGrid;
```

```
double[] xGrid = new double[nxGrid];
double dx;
int nlbcd = 2, nrbcd = 3;
// Define an equally-spaced grid of points for the
// underlying price
dx = (xMax - xMin) / ((double) nint);
xGrid[0] = xMin;
xGrid[nxGrid - 1] = xMax;
for (int i = 2; i <= nxGrid - 1; i++) {
    xGrid[i - 1] = xGrid[i - 2] + dx;
}

FeynmanKac.Boundaries boundaries = new FeynmanKac.Boundaries() {

    public void leftBoundaries(double time, double[][] bndCoeffs) {
        bndCoeffs[0][0] = 0.0;
        bndCoeffs[0][1] = 1.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = -1.0;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 0.0;
        bndCoeffs[1][2] = 1.0;
        bndCoeffs[1][3] = 0.0;
    }

    public void rightBoundaries(double time, double[][] bndCoeffs) {
        bndCoeffs[0][0] = 1.0;
        bndCoeffs[0][1] = 0.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = 0.0;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
        bndCoeffs[1][3] = 0.0;
        bndCoeffs[2][0] = 0.0;
        bndCoeffs[2][1] = 0.0;
        bndCoeffs[2][2] = 1.0;
        bndCoeffs[2][3] = 0.0;
    }

    public double terminal(double x) {
        double zero = 0.0;
        // Strike price
        double strikePrice = 10.0;
        // The payoff function
        double value = Math.max(strikePrice - x, zero);

        return value;
    }
};

FeynmanKac.PdeCoefficients pdeCoefficients
        = new FeynmanKac.PdeCoefficients() {
            // The coefficient sigma(x)

            public double sigma(double x, double t) {
```

```
                double sigma = 0.4;
                return (sigma * x);
            }

            // The coefficient derivative d(sigma) / dx
            public double sigmaPrime(double x, double t) {
                double sigma = 0.4;
                return sigma;
            }

            // The coefficient mu(x)
            public double mu(double x, double t) {
                double interestRate = 0.1;
                double dividend = 0.0;
                return ((interestRate - dividend) * x);
            }

            // The coefficient kappa(x)
            public double kappa(double x, double t) {
                double interestRate = 0.1;
                return interestRate;
            }
        };

FeynmanKac.ForcingTerm forceTerm = new FeynmanKac.ForcingTerm() {

    public void force(int interval, double[] y, double time,
            double width, double[] xlocal, double[] qw,
            double[][] u, double[] phi, double[][] dphi) {
        final int local = 6;
        final double zero = 0.0, one = 1.0;
        double[] yl = new double[local];
        double[] bf = new double[local];
        double value, strikePrice, interestRate;
        double rt, mu;
        int ndeg = xlocal.length;

        for (int i = 0; i < local; i++) {
            yl[i] = y[3 * interval - 3 + i];
            phi[i] = zero;
        }
        strikePrice = 10.0;
        interestRate = 0.1;
        value = 1.0e-5;
        mu = 2.0;
        // This is the local definition of the forcing term
        for (int j = 1; j <= local; j++) {
            for (int l = 1; l <= ndeg; l++) {
                bf[0] = u[0][l - 1];
                bf[1] = u[1][l - 1];
                bf[2] = u[2][l - 1];
                bf[3] = u[6][l - 1];
                bf[4] = u[7][l - 1];
                bf[5] = u[8][l - 1];
                rt = 0.0;
                for (int k = 0; k < local; k++) {
```

```
                    rt += yl[k] * bf[k];
                }
                rt = value / (rt + value
                        - (strikePrice - xlocal[l - 1]));
                phi[j - 1] += qw[l - 1] * bf[j - 1] * Math.pow(rt, mu);
            }
        }
        for (int i = 0; i < local; i++) {
            phi[i] = -phi[i] * width * interestRate * strikePrice;
        }
        // This is the local derivative matrix for the forcing term
        for (int i = 0; i < local; i++) {
            for (int j = 0; j < local; j++) {
                dphi[i][j] = zero;
            }
        }
        for (int j = 1; j <= local; j++) {
            for (int i = 1; i <= local; i++) {
                for (int l = 1; l <= ndeg; l++) {
                    bf[0] = u[0][l - 1];
                    bf[1] = u[1][l - 1];
                    bf[2] = u[2][l - 1];
                    bf[3] = u[6][l - 1];
                    bf[4] = u[7][l - 1];
                    bf[5] = u[8][l - 1];
                    rt = 0.0;
                    for (int k = 0; k < local; k++) {
                        rt += yl[k] * bf[k];
                    }
                    rt = one / (rt + value
                            - (strikePrice - xlocal[l - 1]));
                    dphi[j - 1][i - 1] += qw[l - 1] * bf[i - 1]
                            * bf[j - 1] * Math.pow(rt, mu + 1.0);
                }
            }
        }
        for (int i = 0; i < local; i++) {
            for (int j = 0; j < local; j++) {
                dphi[i][j] = mu * dphi[i][j] * width
                        * Math.pow(value, mu) * interestRate
                        * strikePrice;
            }
        }
    }
};

FeynmanKac european = new FeynmanKac(pdeCoefficients);
FeynmanKac american = new FeynmanKac(pdeCoefficients);

american.setForcingTerm(forceTerm);

european.computeCoefficients(nlbcd, nrbcd, boundaries, xGrid, tGrid);
american.computeCoefficients(nlbcd, nrbcd, boundaries, xGrid, tGrid);

// Evaluate solutions at vector of points evaluationPoints, at each
// time value prior to expiration.
```

```
        double[][] europeanCoefficients = european.getSplineCoefficients();
        double[][] americanCoefficients = american.getSplineCoefficients();

        double[][] splineValuesEuropean = new double[nt][];
        double[][] splineValuesAmerican = new double[nt][];

        for (int i = 0; i < nt; i++) {
            splineValuesEuropean[i]
                    = european.getSplineValue(evaluationPoints,
                            europeanCoefficients[i + 1], 0);
            splineValuesAmerican[i]
                    = american.getSplineValue(evaluationPoints,
                            americanCoefficients[i + 1], 0);
        }

        System.out.printf("%nAmerican Option Premium for Vanilla Put, "
                + "3 and 6 Months Prior to Expiry%n");
        System.out.printf("%7sNumber of equally spaced spline knots:"
                + "%4d%n", " ", nxGrid);
        System.out.printf("%7sNumber of unknowns:%4d%n", " ", n);
        System.out.printf(Locale.ENGLISH, "%7sStrike=%6.2f, sigma=%5.2f,"
                + " Interest Rate=%5.2f%n%n", " ", KS, sigma, r);
        System.out.printf("%7s%10s%20s%20s%n", " ", "Underlying",
                "European", "American");
        for (int i = 0; i < nv; i++) {
            System.out.printf(Locale.ENGLISH, "%7s%10.4f%10.4f%10.4f"
                    + "%10.4f%10.4f%n", " ",
                    evaluationPoints[i],
                    splineValuesEuropean[0][i],
                    splineValuesEuropean[1][i],
                    splineValuesAmerican[0][i],
                    splineValuesAmerican[1][i]);
        }
    }
}
```

Output

```
American Option Premium for Vanilla Put, 3 and 6 Months Prior to Expiry
       Number of equally spaced spline knots:  61
       Number of unknowns: 183
       Strike= 10.00, sigma= 0.40, Interest Rate= 0.10

       Underlying            European            American
           0.0000    9.7531    9.5123   10.0000   10.0000
           2.0000    7.7531    7.5123    8.0000    8.0000
           4.0000    5.7531    5.5128    6.0000    6.0000
           6.0000    3.7569    3.5583    4.0000    4.0000
           8.0000    1.9024    1.9181    2.0202    2.0954
          10.0000    0.6694    0.8703    0.6923    0.9218
          12.0000    0.1675    0.3477    0.1712    0.3624
          14.0000    0.0326    0.1279    0.0332    0.1321
          16.0000    0.0054    0.0448    0.0055    0.0461
```

Example 2: A diffusion model for Call Options

In Beckers (1980) there is a model for a Stochastic Differential Equation of option pricing. The idea is a "constant elasticity of variance diffusion (or CEV) class"

$$dS = \mu S dt + \sigma S^{\alpha/2} dW,\ 0 \le \alpha < 2.$$

The Black-Scholes model is the limiting case $\alpha \to 2$. A numerical solution of this diffusion model yields the price of a call option. Various values of the strike price *K*, time values, σ and power coefficient α are used to evaluate the option price at values of the underlying price. The sets of parameters in the computation are:

1. power $\alpha = 2.0, 1.0, 0.0$
2. strike price $K = 15.0, 20.0, 25.0$
3. volatility $\sigma = 0.2, 0.3, 0.4$
4. times until expiration = {1/12, 4 /12, 7 /12}
5. underlying prices = {19.0, 20.0, 21.0}
6. interest rate $r = 0.05$
7. $x_{\min} = 0, x_{\max} = 60$
8. $nx = 121, n = 3 \times nx = 363$

With this model the Feynman-Kac differential equation is defined by identifying:

- $x : S$
- $\sigma(x,t) :\ \sigma x^{\alpha/2};\ \frac{\partial \sigma}{\partial x} = \frac{\alpha\sigma}{2} x^{\alpha/2-1}$
- $\mu(x,t) :\ rx$
- $\kappa(x,t) :\ r$
- $\phi(f,x,t) \equiv 0$

The payoff function is the "vanilla option", $p(x) = \max(x - K, 0)$.

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

public class FeynmanKacEx2 {

    public static void main(String args[]) throws Exception {
        // Compute Constant Elasticity of Variance Model for Vanilla Call
        // The set of strike prices
        double[] strikePrices = {15.0, 20.0, 25.0};
        // The set of sigma values
        double[] sigma = {0.2, 0.3, 0.4};
```

```
// The set of model diffusion powers
double[] alpha = {2.0, 1.0, 0.0};
// Time values for the options
int nt = 3;
double[] tGrid = {1.0 / 12.0, 4.0 / 12.0, 7.0 / 12.0};
// Values of the underlying where evaluations are made
double[] evaluationPoints = {19.0, 20.0, 21.0};
// Value of the interest rate and continuous dividend
double r = 0.05, dividend = 0.0;
// Values of the min and max underlying values modeled
double xMin = 0.0, xMax = 60.0;
// Define parameters for the integration step. */
int nxGrid = 121;
int ntGrid = 3;
int nv = 3;
int nint = nxGrid - 1, n = 3 * nxGrid;
double[] xGrid = new double[nxGrid];
double dx;
// Number of left/right boundary conditions
int nlbcd = 3, nrbcd = 3;
// Time dependency
boolean[] timeDependence = new boolean[7];
double[] userData = new double[6];
int j;
// Define equally-spaced grid of points for the underlying price
dx = (xMax - xMin) / ((double) nint);
xGrid[0] = xMin;
xGrid[nxGrid - 1] = xMax;
for (int i = 2; i <= nxGrid - 1; i++) {
    xGrid[i - 1] = xGrid[i - 2] + dx;
}

System.out.printf("%2sConstant Elasticity of Variance Model"
        + " for Vanilla Call%n", " ");
System.out.printf(Locale.ENGLISH,
        "%7sInterest Rate:%7.3f   Continuous Dividend:%7.3f%n",
        " ", r, dividend);
System.out.printf(Locale.ENGLISH,
        "%7sMinimum and Maximum Prices of Underlying:%7.2f%7.2f%n",
        " ", xMin, xMax);
System.out.printf("%7sNumber of equally spaced spline knots:%4d%n",
        " ", nxGrid - 1);
System.out.printf("%7sNumber of unknowns:%4d%n%n", " ", n);
System.out.printf(Locale.ENGLISH,
        "%7sTime in Years Prior to Expiration: %7.4f%7.4f%7.4f%n",
        " ", tGrid[0], tGrid[1], tGrid[2]);
System.out.printf(Locale.ENGLISH,
        "%7sOption valued at Underlying Prices:%7.2f%7.2f%7.2f%n%n",
        " ", evaluationPoints[0], evaluationPoints[1],
        evaluationPoints[2]);

for (int i1 = 1; i1 <= 3; i1++) /* Loop over power */ {
    for (int i2 = 1; i2 <= 3; i2++) /* Loop over volatility */ {
        for (int i3 = 1; i3 <= 3; i3++) /* Loop over strike price */ {
            // Pass data through into evaluation methods.
            userData[0] = strikePrices[i3 - 1];
```

```
                userData[1] = xMax;
                userData[2] = sigma[i2 - 1];
                userData[3] = alpha[i1 - 1];
                userData[4] = r;
                userData[5] = dividend;

                FeynmanKac callOption = new FeynmanKac(
                        new MyCoefficients(userData));

                // Right boundary condition is time-dependent
                timeDependence[5] = true;
                callOption.setTimeDependence(timeDependence);
                callOption.computeCoefficients(nlbcd, nrbcd,
                        new MyBoundaries(userData), xGrid, tGrid);
                double[][] optionCoefficients
                        = callOption.getSplineCoefficients();
                double[][] splineValuesOption = new double[ntGrid][];

                // Evaluate solution at vector evaluationPoints, at each time
                // value prior to expiration.
                for (int i = 0; i < ntGrid; i++) {
                    splineValuesOption[i] = callOption.getSplineValue(
                            evaluationPoints, optionCoefficients[i + 1], 0);
                }

                System.out.printf(Locale.ENGLISH, "%2sStrike=%5.2f, Sigma="
                        + "%5.2f, Alpha=%5.2f%n", " ", strikePrices[i3 - 1],
                        sigma[i2 - 1], alpha[i1 - 1]);
                for (int i = 0; i < nv; i++) {
                    System.out.printf("%23sCall Option Values%2s",
                            " ", " ");
                    for (j = 0; j < nt; j++) {
                        System.out.printf(Locale.ENGLISH, "%7.4f ",
                                splineValuesOption[j][i]);
                    }
                    System.out.printf("%n");
                }
                System.out.printf("%n");
            }
        }
    }
}

static class MyCoefficients implements FeynmanKac.PdeCoefficients {

    final double zero = 0.0, half = 0.5;
    private double sigma, strikePrice, interestRate;
    private double alpha, dividend;

    public MyCoefficients(double[] myData) {
        this.strikePrice = myData[0];
        this.sigma = myData[2];
        this.alpha = myData[3];
        this.interestRate = myData[4];
        this.dividend = myData[5];
    }
```

```
    // The coefficient sigma(x)
    public double sigma(double x, double t) {
        return (sigma * Math.pow(x, alpha * half));
    }

    // The coefficient derivative d(sigma) / dx
    public double sigmaPrime(double x, double t) {
        return (half * alpha * sigma * Math.pow(x, alpha * half - 1.0));
    }

    // The coefficient mu(x)
    public double mu(double x, double t) {
        return ((interestRate - dividend) * x);
    }

    // The coefficient kappa(x)
    public double kappa(double x, double t) {
        return interestRate;
    }
}

static class MyBoundaries implements FeynmanKac.Boundaries {

    private double xMax, df, interestRate, strikePrice;

    public MyBoundaries(double[] myData) {
        this.strikePrice = myData[0];
        this.xMax = myData[1];
        this.interestRate = myData[4];
    }

    public void leftBoundaries(double time, double[][] bndCoeffs) {
        bndCoeffs[0][0] = 1.0;
        bndCoeffs[0][1] = 0.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = 0.0;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
        bndCoeffs[1][3] = 0.0;
        bndCoeffs[2][0] = 0.0;
        bndCoeffs[2][1] = 0.0;
        bndCoeffs[2][2] = 1.0;
        bndCoeffs[2][3] = 0.0;
    }

    public void rightBoundaries(double time, double[][] bndCoeffs) {
        df = Math.exp(interestRate * time);
        bndCoeffs[0][0] = 1.0;
        bndCoeffs[0][1] = 0.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = xMax - df * strikePrice;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
```

```
            bndCoeffs[1][3] = 1.0;
            bndCoeffs[2][0] = 0.0;
            bndCoeffs[2][1] = 0.0;
            bndCoeffs[2][2] = 1.0;
            bndCoeffs[2][3] = 0.0;
        }

        public double terminal(double x) {
            double zero = 0.0;
            // The payoff function
            double value = Math.max(x - strikePrice, zero);
            return value;
        }
    }
}
```

Output

```
  Constant Elasticity of Variance Model for Vanilla Call
       Interest Rate:  0.050   Continuous Dividend:  0.000
       Minimum and Maximum Prices of Underlying:   0.00  60.00
       Number of equally spaced spline knots: 120
       Number of unknowns: 363

       Time in Years Prior to Expiration:  0.0833 0.3333 0.5833
       Option valued at Underlying Prices:  19.00  20.00  21.00

  Strike=15.00, Sigma= 0.20, Alpha= 2.00
                       Call Option Values   4.0624  4.2576  4.4734
                       Call Option Values   5.0624  5.2505  5.4492
                       Call Option Values   6.0624  6.2486  6.4386

  Strike=20.00, Sigma= 0.20, Alpha= 2.00
                       Call Option Values   0.1312  0.5951  0.9693
                       Call Option Values   0.5024  1.0880  1.5093
                       Call Option Values   1.1980  1.7478  2.1745

  Strike=25.00, Sigma= 0.20, Alpha= 2.00
                       Call Option Values   0.0000  0.0111  0.0751
                       Call Option Values   0.0000  0.0376  0.1630
                       Call Option Values   0.0006  0.1036  0.3150

  Strike=15.00, Sigma= 0.30, Alpha= 2.00
                       Call Option Values   4.0636  4.3405  4.6627
                       Call Option Values   5.0625  5.2951  5.5794
                       Call Option Values   6.0624  6.2712  6.5248

  Strike=20.00, Sigma= 0.30, Alpha= 2.00
                       Call Option Values   0.3107  1.0261  1.5479
                       Call Option Values   0.7317  1.5404  2.0999
                       Call Option Values   1.3762  2.1674  2.7362

  Strike=25.00, Sigma= 0.30, Alpha= 2.00
                       Call Option Values   0.0005  0.1124  0.3564
                       Call Option Values   0.0035  0.2184  0.5565
```

```
                    Call Option Values   0.0184  0.3869  0.8230

Strike=15.00, Sigma= 0.40, Alpha= 2.00
                    Call Option Values   4.0755  4.5143  4.9673
                    Call Option Values   5.0660  5.4210  5.8328
                    Call Option Values   6.0633  6.3588  6.7306

Strike=20.00, Sigma= 0.40, Alpha= 2.00
                    Call Option Values   0.5109  1.4625  2.1260
                    Call Option Values   0.9611  1.9934  2.6915
                    Call Option Values   1.5807  2.6088  3.3202

Strike=25.00, Sigma= 0.40, Alpha= 2.00
                    Call Option Values   0.0081  0.3302  0.7795
                    Call Option Values   0.0287  0.5178  1.0656
                    Call Option Values   0.0820  0.7690  1.4097

Strike=15.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values   4.0624  4.2479  4.4312
                    Call Option Values   5.0624  5.2479  5.4312
                    Call Option Values   6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values   0.0000  0.0219  0.1051
                    Call Option Values   0.1497  0.4107  0.6484
                    Call Option Values   1.0832  1.3314  1.5773

Strike=25.00, Sigma= 0.20, Alpha= 1.00
                    Call Option Values  -0.0000 -0.0000  0.0000
                    Call Option Values  -0.0000 -0.0000  0.0000
                    Call Option Values  -0.0000 -0.0000  0.0000

Strike=15.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values   4.0624  4.2479  4.4312
                    Call Option Values   5.0624  5.2479  5.4312
                    Call Option Values   6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values   0.0010  0.0786  0.2208
                    Call Option Values   0.1993  0.4997  0.7539
                    Call Option Values   1.0835  1.3444  1.6022

Strike=25.00, Sigma= 0.30, Alpha= 1.00
                    Call Option Values  -0.0000  0.0000  0.0000
                    Call Option Values  -0.0000  0.0000  0.0000
                    Call Option Values  -0.0000  0.0000  0.0004

Strike=15.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values   4.0624  4.2479  4.4312
                    Call Option Values   5.0624  5.2479  5.4312
                    Call Option Values   6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.40, Alpha= 1.00
                    Call Option Values   0.0072  0.1540  0.3446
                    Call Option Values   0.2498  0.5950  0.8728
                    Call Option Values   1.0868  1.3795  1.6586
```

```
Strike=25.00, Sigma= 0.40, Alpha= 1.00
                     Call Option Values   0.0000  0.0000  0.0000
                     Call Option Values   0.0000  0.0000  0.0005
                     Call Option Values   0.0000  0.0002  0.0057

Strike=15.00, Sigma= 0.20, Alpha= 0.00
                     Call Option Values   4.0624  4.2479  4.4311
                     Call Option Values   5.0624  5.2479  5.4312
                     Call Option Values   6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.20, Alpha= 0.00
                     Call Option Values   0.0001  0.0001  0.0002
                     Call Option Values   0.0816  0.3316  0.5748
                     Call Option Values   1.0817  1.3308  1.5748

Strike=25.00, Sigma= 0.20, Alpha= 0.00
                     Call Option Values   0.0000 -0.0000 -0.0000
                     Call Option Values   0.0000 -0.0000 -0.0000
                     Call Option Values  -0.0000  0.0000 -0.0000

Strike=15.00, Sigma= 0.30, Alpha= 0.00
                     Call Option Values   4.0624  4.2479  4.4312
                     Call Option Values   5.0624  5.2479  5.4312
                     Call Option Values   6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.30, Alpha= 0.00
                     Call Option Values   0.0000 -0.0000  0.0026
                     Call Option Values   0.0894  0.3326  0.5753
                     Call Option Values   1.0826  1.3306  1.5749

Strike=25.00, Sigma= 0.30, Alpha= 0.00
                     Call Option Values   0.0000 -0.0000  0.0000
                     Call Option Values   0.0000 -0.0000  0.0000
                     Call Option Values   0.0000 -0.0000 -0.0000

Strike=15.00, Sigma= 0.40, Alpha= 0.00
                     Call Option Values   4.0624  4.2479  4.4312
                     Call Option Values   5.0624  5.2479  5.4312
                     Call Option Values   6.0624  6.2479  6.4312

Strike=20.00, Sigma= 0.40, Alpha= 0.00
                     Call Option Values  -0.0000  0.0001  0.0108
                     Call Option Values   0.0985  0.3383  0.5781
                     Call Option Values   1.0830  1.3306  1.5749

Strike=25.00, Sigma= 0.40, Alpha= 0.00
                     Call Option Values   0.0000  0.0000  0.0000
                     Call Option Values   0.0000 -0.0000  0.0000
                     Call Option Values   0.0000 -0.0000  0.0000
```

Example 3: European Option with two payoff strategies

This example evaluates the price of a European Option using two payoff strategies: Cash-or-Nothing and Vertical Spread. In the first case the payoff function is

$$p(x) = \begin{cases} 0, & x \le K \\ B, & x > K \end{cases} .$$

The value B is regarded as the bet on the asset price, see Wilmott et al. (1995, p. 39-40). The second case has the payoff function

$$p(x) = \max(x - K_1) - \max(x - K_2), K_2 > K_1.$$

Both problems use the same boundary conditions. Each case requires a separate integration of the Black-Scholes differential equation, but only the payoff function evaluation differs in each case. The sets of parameters in the computation are:

1. Strike and bet prices $K_1 = 10.0,\ K_2 = 15.0, andB = 2.0.$
2. Volatility $\sigma = 0.4$
3. Times until expiration = $\{1/4,\ 1/2\}$
4. Interest rate $r = 0.1$
5. $x_{\min} = 0.0, x_{\max} = 30.0$
6. $nx = 61, n = 3 \times nx = 183$

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

public class FeynmanKacEx3 {

    public static void main(String args[]) throws Exception {
        int i;
        int nxGrid = 61;
        int ntGrid = 2;
        int nv = 12;
        // The strike price
        double strikePrice = 10.0;
        // The spread value
        double spreadValue = 15.0;
        // The Bet for the Cash-or-Nothing Call
        double bet = 2.0;
        // The sigma value
        double sigma = 0.4;
        // Time values for the options
        int nt = 2;
        double[] tGrid = {0.25, 0.5};
        // Values of the underlying where evaluations are made
        double[] evaluationPoints = new double[nv];
        // Value of the interest rate and continuous dividend
```

```
double r = 0.1, dividend = 0.0;
// Values of the min and max underlying values modeled
double xMin = 0.0, xMax = 30.0;
// Define parameters for the integration step.
int nint = nxGrid - 1, n = 3 * nxGrid;
double[] xGrid = new double[nxGrid];
double dx;
// Number of left/right boundary conditions.
int nlbcd = 3, nrbcd = 3;
// Structure for the evaluation methods.
int iData;
double[] rData = new double[7];
boolean[] timeDependence = new boolean[7];

// Define an equally-spaced grid of points for the
// underlying price
dx = (xMax - xMin) / ((double) (nint));
xGrid[0] = xMin;
xGrid[nxGrid - 1] = xMax;
for (i = 2; i <= nxGrid - 1; i++) {
    xGrid[i - 1] = xGrid[i - 2] + dx;
}
for (i = 1; i <= nv; i++) {
    evaluationPoints[i - 1] = 2.0 + (i - 1) * 2.0;
}
rData[0] = strikePrice;
rData[1] = bet;
rData[2] = spreadValue;
rData[3] = xMax;
rData[4] = sigma;
rData[5] = r;
rData[6] = dividend;
// Flag the difference in payoff functions
// 1 for the Bet, 2 for the Vertical Spread
// In both cases, no time dependencies for
// the coefficients and the left boundary
// conditions
timeDependence[5] = true;
iData = 1;
FeynmanKac betOption = new FeynmanKac(new MyCoefficients(rData));
betOption.setTimeDependence(timeDependence);
betOption.computeCoefficients(nlbcd, nrbcd,
        new MyBoundaries(rData, iData), xGrid, tGrid);
double[][] betOptionCoefficients = betOption.getSplineCoefficients();
double[][] splineValuesBetOption = new double[ntGrid][];

iData = 2;
FeynmanKac spreadOption = new FeynmanKac(new MyCoefficients(rData));

spreadOption.setTimeDependence(timeDependence);
spreadOption.computeCoefficients(nlbcd, nrbcd,
        new MyBoundaries(rData, iData), xGrid, tGrid);
double[][] spreadOptionCoefficients
        = spreadOption.getSplineCoefficients();
double[][] splineValuesSpreadOption = new double[ntGrid][];
```

```
        // Evaluate solutions at vector evaluationPoints, at each time value
        // prior to expiration.
        for (i = 0; i < ntGrid; i++) {
            splineValuesBetOption[i]
                    = betOption.getSplineValue(evaluationPoints,
                            betOptionCoefficients[i + 1], 0);
            splineValuesSpreadOption[i] = spreadOption.getSplineValue(
                    evaluationPoints, spreadOptionCoefficients[i + 1], 0);
        }

        System.out.printf("%2sEuropean Option Value for A Bet%n", " ");
        System.out.printf("%3sand a Vertical Spread, 3 and 6 Months "
                + "Prior to Expiry%n", " ");
        System.out.printf("%5sNumber of equally spaced spline knots:%4d%n",
                " ", nxGrid);
        System.out.printf("%5sNumber of unknowns:%4d\n", " ", n);
        System.out.printf(Locale.ENGLISH,
                "%5sStrike=%5.2f, Sigma=%5.2f, Interest Rate=" + "%5.2f%n", " ",
                strikePrice, sigma, r);
        System.out.printf(Locale.ENGLISH,
                "%5sBet=%5.2f, Spread Value=%5.2f%n%n",
                " ", bet, spreadValue);
        System.out.printf("%17s%18s%18s%n", "Underlying", "A Bet",
                "Vertical Spread");
        for (i = 0; i < nv; i++) {
            System.out.printf(Locale.ENGLISH,
                    "%8s%9.4f%9.4f%9.4f%9.4f%9.4f%n", " ",
                    evaluationPoints[i],
                    splineValuesBetOption[0][i],
                    splineValuesBetOption[1][i],
                    splineValuesSpreadOption[0][i],
                    splineValuesSpreadOption[1][i]);
        }
    }

    // These classes define the coefficients, payoff, boundary conditions
    // and forcing term for American and European Options.
    static class MyCoefficients implements FeynmanKac.PdeCoefficients {

        final double zero = 0.0;
        double sigma, strikePrice, interestRate;
        double spread, bet, dividend;
        int dataInt;
        double value = 0.0;

        public MyCoefficients(double[] rData) {
            this.strikePrice = rData[0];
            this.bet = rData[1];
            this.spread = rData[2];
            this.sigma = rData[4];
            this.interestRate = rData[5];
            this.dividend = rData[6];
        }

        // The coefficient sigma(x)
        public double sigma(double x, double t) {
```

```
        return (sigma * x);
    }

    // The coefficient derivative d(sigma) / dx
    public double sigmaPrime(double x, double t) {
        return sigma;
    }

    // The coefficient mu(x)
    public double mu(double x, double t) {
        return ((interestRate - dividend) * x);
    }

    // The coefficient kappa(x)
    public double kappa(double x, double t) {
        return interestRate;
    }
}

static class MyBoundaries implements FeynmanKac.Boundaries {

    private double strikePrice, spread, bet, interestRate, df;
    private int dataInt;

    public MyBoundaries(double[] rData, int iData) {
        this.strikePrice = rData[0];
        this.bet = rData[1];
        this.spread = rData[2];
        this.interestRate = rData[5];
        this.dataInt = iData;
    }

    public void leftBoundaries(double time, double[][] bndCoeffs) {
        bndCoeffs[0][0] = 1.0;
        bndCoeffs[0][1] = 0.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = 0.0;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
        bndCoeffs[1][3] = 0.0;
        bndCoeffs[2][0] = 0.0;
        bndCoeffs[2][1] = 0.0;
        bndCoeffs[2][2] = 1.0;
        bndCoeffs[2][3] = 0.0;
    }

    public void rightBoundaries(double time, double[][] bndCoeffs) {
        // This is the discount factor using the risk-free
        // interest rate
        df = Math.exp(interestRate * time);
        // Use flag passed to decide on boundary condition
        switch (dataInt) {
            case 1:
                bndCoeffs[0][0] = 1.0;
                bndCoeffs[0][1] = 0.0;
```

```
                bndCoeffs[0][2] = 0.0;
                bndCoeffs[0][3] = bet * df;
                break;
            case 2:
                bndCoeffs[0][0] = 1.0;
                bndCoeffs[0][1] = 0.0;
                bndCoeffs[0][2] = 0.0;
                bndCoeffs[0][3] = (spread - strikePrice) * df;
                break;
        }
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
        bndCoeffs[1][3] = 0.0;
        bndCoeffs[2][0] = 0.0;
        bndCoeffs[2][1] = 0.0;
        bndCoeffs[2][2] = 1.0;
        bndCoeffs[2][3] = 0.0;
    }

    public double terminal(double x) {
        final double zero = 0.0;
        double value = 0.0;
        switch (dataInt) {
            // The payoff function - Use flag passed to decide which
            case 1:
                // After reaching the strike price the payoff jumps
                // from zero to the bet value.
                value = zero;
                if (x > strikePrice) {
                    value = bet;
                }
                break;
            case 2:
                /* Function is zero up to strike price.
                 Then linear between strike price and spread.
                 Then has constant value Spread-Strike Price after
                 the value Spread. */
                value = Math.max(x - strikePrice, zero)
                        - Math.max(x - spread, zero);
                break;
        }
        return value;
    }
  }
}
```

Output

```
 European Option Value for A Bet
  and a Vertical Spread, 3 and 6 Months Prior to Expiry
    Number of equally spaced spline knots:  61
    Number of unknowns: 183
    Strike=10.00, Sigma= 0.40, Interest Rate= 0.10
    Bet= 2.00, Spread Value=15.00
```

```
Underlying             A Bet   Vertical Spread
    2.0000  -0.0000   0.0000  -0.0000   0.0000
    4.0000   0.0000   0.0013   0.0000   0.0005
    6.0000   0.0112   0.0729   0.0038   0.0452
    8.0000   0.2686   0.4291   0.1486   0.3833
   10.0000   0.9948   0.9781   0.8898   1.1907
   12.0000   1.6103   1.4301   2.1904   2.2267
   14.0000   1.8650   1.6926   3.4267   3.1567
   16.0000   1.9335   1.8171   4.2274   3.8282
   18.0000   1.9477   1.8696   4.6261   4.2499
   20.0000   1.9501   1.8902   4.7903   4.4918
   22.0000   1.9505   1.8979   4.8493   4.6222
   24.0000   1.9506   1.9008   4.8685   4.6901
```

Example 4: Convertible bonds

This example evaluates the price of a convertible bond. Here, convertibility means that the bond may, at any time of the holder's choosing, be converted to a multiple of the specified asset. Thus a convertible bond with price x returns an amount K at time T *unless* the owner has converted the bond to νx, $\nu \geq 1$ units of the asset at some time prior to T. This definition, the differential equation and boundary conditions are given in Chapter 18 of Wilmott et al. (1996). Using a constant interest rate and volatility factor, the parameters and boundary conditions are:

1. Bond face value $K = 1$, conversion factor $\nu = 1.125$
2. Volatility $\sigma = 0.25$
3. Times until expiration = $\{1/2, 1\}$
4. Interest rate $r = 0.1$, dividend $D = 0.02$
5. $x_{\min} = 0, x_{\max} = 4$
6. $nx = 61, n = 3 \times nx = 183$
7. Boundary conditions $f(0,t) = K \exp(r - (T - t))$, $f(x_{\max}, t) = \nu x_{\max}$
8. Terminal data $f(x,T) = \max(K, \nu x)$
9. Constraint for bond holder $f(x,t) \geq \nu x$

Note that the error tolerance is set to a pure absolute error of value 10^{-3}. The free boundary constraint $f(x,t) \geq \nu x$ is achieved by use of a non-linear forcing term in interface `ForcingTerm`. The coefficient values of the Hermite quintic spline representing the approximate solution of the differential equation at the initial time point are provided with the interface `InitialData`.

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

public class FeynmanKacEx4 {
```

```
public static void main(String args[]) throws Exception {
    int i;
    int nxGrid = 61;
    int ntGrid = 2;
    int nv = 13;

    // Compute value of a Convertible Bond
    // The face value
    double KS = 1.0;
    // The sigma or volatility value
    double sigma = 0.25;
    // Time values for the options
    double[] tGrid = {0.5, 1.0};
    // Values of the underlying where evaluation are made
    double[] evaluationPoints = new double[nv];
    // Value of the interest rate, continuous dividend and factor
    double r = 0.1, dividend = 0.02, factor = 1.125;
    // Values of the min and max underlying values modeled
    double xMin = 0.0, xMax = 4.0;
    // Define parameters for the integration step.
    int nint = nxGrid - 1, n = 3 * nxGrid;
    double[] xGrid = new double[nxGrid];
    // Array for user-defined data
    double[] rData = new double[8];
    double dx, atol;
    // Number of left/right boundary conditions.
    int nlbcd = 3, nrbcd = 3;
    boolean[] timeDependence = new boolean[7];

    /*
     * Define an equally-spaced grid of points for the
     * underlying price
     */
    dx = (xMax - xMin) / ((double) nint);
    xGrid[0] = xMin;
    xGrid[nxGrid - 1] = xMax;
    for (i = 2; i <= nxGrid - 1; i++) {
        xGrid[i - 1] = xGrid[i - 2] + dx;
    }
    for (i = 1; i <= nv; i++) {
        evaluationPoints[i - 1] = (i - 1) * 0.25;
    }
    // Pass the data for evaluation
    rData[0] = KS;
    rData[1] = xMax;
    rData[2] = sigma;
    rData[3] = r;
    rData[4] = dividend;
    rData[5] = factor;
    // Use a pure absolute error tolerance for the integration
    atol = 1.0e-3;
    rData[6] = atol;
    // Compute value of convertible bond
    FeynmanKac convertibleBond = new FeynmanKac(new MyCoefficients(rData));
```

```
        MyForcingTerm forceTerm = new MyForcingTerm(rData);
        MyInitialData initialData = new MyInitialData(rData);

        convertibleBond.setForcingTerm(forceTerm);
        convertibleBond.setInitialData(initialData);
        //convertibleBond.setErrorTolerances(1.0e-3,0.0);
        convertibleBond.setAbsoluteErrorTolerances(1.0e-3);
        convertibleBond.setRelativeErrorTolerances(0.0);

        // Only the left boundary conditions are time dependent
        timeDependence[4] = true;
        convertibleBond.setTimeDependence(timeDependence);

        convertibleBond.computeCoefficients(nlbcd, nrbcd,
                new MyBoundaries(rData), xGrid, tGrid);
        double[][] bondCoefficients = convertibleBond.getSplineCoefficients();

        double[][] bondSplineValues = new double[ntGrid + 1][];

        /*
         * Evaluate and display solutions at vector of points XS(:),
         * at each time value prior to expiration.
         */
        for (i = 0; i <= ntGrid; i++) {
            bondSplineValues[i]
                    = convertibleBond.getSplineValue(evaluationPoints,
                            bondCoefficients[i], 0);
        }

        System.out.printf("%2sConvertible Bond Value, 0+, 6 and 12 Months "
                + "Prior to Expiry%n", " ");
        System.out.printf("%5sNumber of equally spaced spline knots:%4d%n",
                " ", nxGrid);
        System.out.printf("%5sNumber of unknowns:%4d%n", " ", n);
        System.out.printf(Locale.ENGLISH, "%5sStrike=%5.2f, Sigma=%5.2f%n",
                " ", KS, sigma);
        System.out.printf(Locale.ENGLISH, "%5sInterest Rate=%5.2f, "
                + "Dividend=%5.2f, Factor=%6.3f%n%n",
                " ", r, dividend, factor);
        System.out.printf("%15s%18s%n", "Underlying", "Bond Value");
        for (i = 0; i < nv; i++) {
            System.out.printf(Locale.ENGLISH, "%7s%8.4f%8.4f%8.4f%8.4f%n",
                    " ", evaluationPoints[i],
                    bondSplineValues[0][i],
                    bondSplineValues[1][i],
                    bondSplineValues[2][i]);
        }
    }

    /*
     * These classes define the coefficients, payoff, boundary conditions
     * and forcing term.
     */
    static class MyCoefficients implements FeynmanKac.PdeCoefficients {

        private double sigma, strikePrice, interestRate;
```

```
    private double dividend, factor, zero = 0.0;
    private double value = 0.0;

    public MyCoefficients(double[] rData) {
        this.strikePrice = rData[0];
        this.sigma = rData[2];
        this.interestRate = rData[3];
        this.dividend = rData[4];
        this.factor = rData[5];
    }

    // The coefficient sigma(x)
    public double sigma(double x, double t) {
        return (sigma * x);
    }

    // The coefficient derivative d(sigma) / dx
    public double sigmaPrime(double x, double t) {
        return sigma;
    }

    // The coefficient mu(x)
    public double mu(double x, double t) {
        return ((interestRate - dividend) * x);
    }

    // The coefficient kappa(x)
    public double kappa(double x, double t) {
        return interestRate;
    }
}

static class MyBoundaries implements FeynmanKac.Boundaries {

    private double interestRate, strikePrice, dp, factor, xMax;

    public MyBoundaries(double[] rData) {
        this.strikePrice = rData[0];
        this.xMax = rData[1];
        this.interestRate = rData[3];
        this.factor = rData[5];
    }

    public void leftBoundaries(double time, double[][] bndCoeffs) {
        dp = strikePrice * Math.exp(time * interestRate);
        bndCoeffs[0][0] = 1.0;
        bndCoeffs[0][1] = 0.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = dp;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
        bndCoeffs[1][3] = 0.0;
        bndCoeffs[2][0] = 0.0;
        bndCoeffs[2][1] = 0.0;
        bndCoeffs[2][2] = 1.0;
```

```
        bndCoeffs[2][3] = 0.0;
    }

    public void rightBoundaries(double time, double[][] bndCoeffs) {
        bndCoeffs[0][0] = 1.0;
        bndCoeffs[0][1] = 0.0;
        bndCoeffs[0][2] = 0.0;
        bndCoeffs[0][3] = factor * xMax;
        bndCoeffs[1][0] = 0.0;
        bndCoeffs[1][1] = 1.0;
        bndCoeffs[1][2] = 0.0;
        bndCoeffs[1][3] = factor;
        bndCoeffs[2][0] = 0.0;
        bndCoeffs[2][1] = 0.0;
        bndCoeffs[2][2] = 1.0;
        bndCoeffs[2][3] = 0.0;
    }

    public double terminal(double x) {
        // The payoff function
        double value = Math.max(factor * x, strikePrice);
        return value;
    }
}

static class MyForcingTerm implements FeynmanKac.ForcingTerm {

    private final double zero = 0.0, one = 1.0;
    private double value, strikePrice, interestRate;
    private double rt, mu, factor;

    public MyForcingTerm(double[] rData) {
        this.value = rData[6];
        this.strikePrice = rData[0];
        this.interestRate = rData[3];
        this.factor = rData[5];
    }

    public void force(int interval, double[] y, double time, double width,
            double[] xlocal, double[] qw, double[][] u,
            double[] phi, double[][] dphi) {
        final int local = 6;
        int ndeg = xlocal.length;
        double[] yl = new double[6];
        double[] bf = new double[6];
        for (int i = 0; i < local; i++) {
            yl[i] = y[3 * interval - 3 + i];
            phi[i] = zero;
        }
        for (int i = 0; i < local; i++) {
            for (int j = 0; j < local; j++) {
                dphi[i][j] = zero;
            }
        }

        mu = 2.0;
```

```
        /*
         * This is the local definition of the forcing term -
         * It "forces" the constraint f >= factor*x.
         */
        for (int j = 1; j <= local; j++) {
            for (int l = 1; l <= ndeg; l++) {
                bf[0] = u[0][l - 1];
                bf[1] = u[1][l - 1];
                bf[2] = u[2][l - 1];
                bf[3] = u[6][l - 1];
                bf[4] = u[7][l - 1];
                bf[5] = u[8][l - 1];
                rt = 0.0;
                for (int k = 0; k < local; k++) {
                    rt += yl[k] * bf[k];
                }
                rt = value / (rt + value - factor * xlocal[l - 1]);
                phi[j - 1] += qw[l - 1] * bf[j - 1] * Math.pow(rt, mu);
            }
        }
        for (int i = 0; i < local; i++) {
            phi[i] = -phi[i] * width * factor * strikePrice;
        }
        /*
         * This is the local derivative matrix for the forcing term
         */
        for (int j = 1; j <= local; j++) {
            for (int i = 1; i <= local; i++) {
                for (int l = 1; l <= ndeg; l++) {
                    bf[0] = u[0][l - 1];
                    bf[1] = u[1][l - 1];
                    bf[2] = u[2][l - 1];
                    bf[3] = u[6][l - 1];
                    bf[4] = u[7][l - 1];
                    bf[5] = u[8][l - 1];
                    rt = 0.0;
                    for (int k = 0; k < local; k++) {
                        rt += yl[k] * bf[k];
                    }
                    rt = one / (rt + value - factor * xlocal[l - 1]);
                    dphi[j - 1][i - 1] += qw[l - 1] * bf[i - 1]
                            * bf[j - 1] * Math.pow(value * rt, mu) * rt;
                }
            }
        }
        for (int i = 0; i < local; i++) {
            for (int j = 0; j < local; j++) {
                dphi[i][j] = -mu * dphi[i][j] * width
                        * factor * strikePrice;
            }
        }
    }
}

static class MyInitialData implements FeynmanKac.InitialData {
```

```
        private double[] data;

        public MyInitialData(double[] rData) {
            data = new double[rData.length];
            for (int i = 0; i < rData.length; i++) {
                data[i] = rData[i];
            }
        }

        public void init(double[] xGrid, double[] tGrid, double tp,
                double[] yprime, double[] y, double[] atol, double[] rtol) {
            int nxGrid = xGrid.length;
            if (tp == 0.0) { // Set initial data precisely.
                for (int i = 1; i <= nxGrid; i++) {
                    if (xGrid[i - 1] * data[5] < data[0]) {
                        y[3 * i - 3] = data[0];
                        y[3 * i - 2] = 0.0;
                        y[3 * i - 1] = 0.0;
                    } else {
                        y[3 * i - 3] = xGrid[i - 1] * data[5];
                        y[3 * i - 2] = data[5];
                        y[3 * i - 1] = 0.0;
                    }
                }
            }
        }
    }
}
```

Output

```
  Convertible Bond Value, 0+, 6 and 12 Months Prior to Expiry
     Number of equally spaced spline knots:  61
     Number of unknowns: 183
     Strike= 1.00, Sigma= 0.25
     Interest Rate= 0.10, Dividend= 0.02, Factor= 1.125

     Underlying        Bond Value
         0.0000  1.0000  0.9512  0.9048
         0.2500  1.0000  0.9512  0.9049
         0.5000  1.0000  0.9513  0.9065
         0.7500  1.0000  0.9737  0.9605
         1.0000  1.1250  1.1416  1.1464
         1.2500  1.4063  1.4117  1.4121
         1.5000  1.6875  1.6922  1.6922
         1.7500  1.9688  1.9731  1.9731
         2.0000  2.2500  2.2540  2.2540
         2.2500  2.5313  2.5349  2.5349
         2.5000  2.8125  2.8160  2.8160
         2.7500  3.0938  3.0970  3.0970
         3.0000  3.3750  3.3781  3.3781
```

Example 5: Calculating the Greeks using the Feynman-Kac Class

In this example, the Feynman-Kac (FK) class is used to solve for the Greeks, i.e. various derivatives of FK solutions applicable to the pricing of options and related financial derivatives. In order to illustrate and verify these calculations, the Greeks are calculated by two methods. The first method involves the FK solution to the diffusion model for call options given in Example 2 for the Black-Scholes (BS) case, i.e. $\alpha = 2$. The second method calculates the Greeks using the closed-form BS evaluations which can be found at http://en.wikipedia.org/wiki/The_Greeks.

Example 5 calculates FK and BS solutions $V(S,t)$ to the BS problem and the following Greeks:

- $Delta = \frac{\partial V}{\partial S}$ is the first derivative of the *Value*, $V(S,t)$, of a portfolio of derivative security derived from underlying instrument with respect to the underlying instrument's price *S*;
- $Gamma = \frac{\partial^2 V}{\partial S^2}$;
- $Theta = -\frac{\partial V}{\partial t}$ is the negative first derivative of *V* with respect to time *t*;
- $Charm = \frac{\partial^2 V}{\partial S \partial t}$;
- $Color = \frac{\partial^3 V}{\partial S^2 \partial t}$;
- $Rho = -\frac{\partial V}{\partial r}$ is the first derivative of *V* with respect to risk free rate *r*;
- $Vega = \frac{\partial V}{\partial \sigma}$ measures sensitivity to volatility parameter σ of the underlying *S*;
- $Volga = \frac{\partial^2 V}{\partial \sigma^2}$;
- $Vanna = \frac{\partial^2 V}{\partial S \partial \sigma}$;
- $Speed = \frac{\partial^3 V}{\partial S^3}$.

Intrinsic Greeks include derivatives involving only *S* and *t*, the intrinsic FK arguments. In the above list, *Value, Delta, Gamma, Theta, Charm, Color,* and *Speed* are all intrinsic Greeks. As is discussed in Hanson, R. (2008) *Integrating Feynman-Kac equations Using Hermite Quintic Finite Elements*, the expansion of the FK solution function $V(S,t)$ in terms of quintic polynomial functions defined on *S*-grid subintervals and subject to continuity constraints in derivatives 0, 1, and 2 across the boundaries of these subintervals allows *Value, Delta, Gamma, Theta, Charm,* and *Color* to be calculated directly by the FK class methods `getSplineCoefficients`, `getSplineCoefficientsPrime`, and `getSplineValue`.

Non-intrinsic Greeks are derivatives of *V* involving FK parameters other than the intrinsic arguments *S* and *t*, such as *r* and σ. Non-intrinsic Greeks in the above list include *Rho, Vega, Volga,* and *Vanna*. In order to calculate non-intrinsic Greek (parameter) derivatives or intrinsic Greek *S*- derivatives beyond the second (such as *Speed*) or *t*- derivatives beyond the first, the entire FK solution must be calculated 3 times (for a parabolic fit) or five times (for a quartic fit), at the point where the derivative is to be evaluated and at nearby points located symmetrically on either side.

Using a Taylor series expansion of $f(\sigma+\varepsilon)$ truncated to *m*+1 terms (to allow an *m*-degree polynomial fit of *m*+1 data points):

$$f(\sigma+\varepsilon) \;=\; \sum_{n=0}^{m} \frac{f^{(n)}(\sigma)}{n!}\varepsilon^n$$

we are able to derive the following parabolic (3 point) estimation of first and second derivatives $f^{(1)}(\sigma)$ and $f^{(2)}(\sigma)$ in terms of the three values: $f(\sigma-\varepsilon)$, $f(\sigma)$, and $f(\sigma+\varepsilon)$, where $\varepsilon = \varepsilon_{frac}\sigma$ and $0 < \varepsilon_{frac} << 1$:

$$f^{(1)}(\sigma) \;\equiv\; \frac{\partial f(\sigma)}{\partial \sigma} \;\approx\; f^{[1]}(\sigma,\varepsilon) \;\equiv\; \frac{f(\sigma+\varepsilon)-f(\sigma-\varepsilon)}{2\varepsilon}$$

$$f^{(2)}(\sigma) \;\equiv\; \frac{\partial^2 f(\sigma)}{\partial \sigma^2} \;\approx\; f^{[2]}(\sigma,\varepsilon) \;\equiv\; \frac{f(\sigma+\varepsilon)+f(\sigma-\varepsilon)-2f(\sigma)}{\varepsilon^2}$$

Similarly, the quartic (5 point) estimation of $f^{(1)}(\sigma)$ and $f^{(2)}(\sigma)$ in terms of $f(\sigma-2\varepsilon)$, $f(\sigma-\varepsilon)$, $f(\sigma)$, $f(\sigma+\varepsilon)$, and $f(\sigma+2\varepsilon)$ is:

$$f^{(1)}(\sigma) \;\approx\; \frac{4}{3}f^{[1]}(\sigma,\varepsilon) \;-\; \frac{1}{3}f^{[1]}(\sigma,2\varepsilon)$$

$$f^{(2)}(\sigma) \;\approx\; \frac{4}{3}f^{[2]}(\sigma,\varepsilon) \;-\; \frac{1}{3}f^{[2]}(\sigma,2\varepsilon)$$

For our example, the quartic estimate does not appear to be significantly better than the parabolic estimate, so we have produced only parabolic estimates by setting variable `iquart` to 0. The user may try the example with the quartic estimate simply by setting `iquart` to 1.

As is pointed out in *Integrating Feynman-Kac equations Using Hermite Quintic Finite Elements*, the quintic polynomial expansion function used by Feynman-Kac only allows for continuous derivatives through the second derivative. While up to fifth derivatives can be calculated from the quintic expansion (indeed class `FeynmanKac` method `getSplineValue` will allow the third derivative to be calculated by setting parameter `ideriv` to `3`, as is done in Example 5), the accuracy is compromised by the inherent lack of continuity across grid points (i.e. subinterval boundaries).

The accurate second derivatives in *S* returned by FK method `getSplineValue` can be leveraged into a third derivative estimate by calculating three FK second derivative solutions, the first solution for grid and evaluation point sets $\{S,\ f^{(2)}(S)\}$ and the second and third solutions for solution grid and evaluation point sets $\{S+\varepsilon,\ f^{(2)}(S+\varepsilon)\}$ and $\{S-\varepsilon,\ f^{(2)}(S-\varepsilon)\}$, where the solution grid and evaluation point sets are shifted up and down by ε. In Example 5, ε is set to $\varepsilon_{frac}\bar{S}$, where $\bar{S}$ is the average value of *S* over the range of grid values and $0 < \varepsilon_{frac} << 1$. The third derivative solution can then be obtained using the parabolic estimate:

$$f^{(3)}(S) \;=\; \frac{\partial f^{(2)}(S)}{\partial S} \;\approx\; \frac{f^{(2)}(S+\varepsilon)-f^{(2)}(S-\varepsilon)}{2\varepsilon}$$

This procedure is implemented in Example 5 to calculate the Greek *Speed*. (For comparison purposes, *Speed* is also calculated *directly* in Example 5 by setting the `getSplineValue` input S derivative parameter `iSDeriv` to 3. The output from this direct calculation is called "*Speed2*".)

The average and maximum relative errors (defined as the absolute value of the difference between the BS and FK values divided by the BS value) for each of the Greeks is given at the end of the output. (These relative error statistics are given for nine combinations of Strike Price and Volatility, but only one of the nine combinations is actually printed in the output.) Both intrinsic and non-intrinsic Greeks have good accuracy (average relative error is in the range 0.01 – 0.0001) except for *Volga*, which has an average relative error of about 0.05. This is probably a result of the fact that *Volga* involves differences of differences, which will more quickly erode accuracy than calculations using only one difference to approximate a derivative. Possible ways to improve upon the 2 to 4 significant digits of accuracy achieved in Example 5 include increasing FK integration accuracy by reducing the initial step size (using method `setInitialStepsize`), by choosing more closely spaced S and t grid points (by adjusting method `computeCoefficients` input parameter arrays `xGrid` and `tGrid`), and by adjusting ε_{frac} so that the central differences used to calculate the derivatives are not too small to compromise accuracy.

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

public class FeynmanKacEx5 {

    private static double[] strikePrices = {15.0, 20.0, 25.0};
    // The set of sigma values
    private static double[] sigma = {0.2, 0.3, 0.4};
    // The set of model diffusion powers:  alpha = 2.0 <==> Black Scholes
    private static double[] alpha = {2.0, 1.0, 0.0};
    // Time values for the options
    private static double[] tGrid = {1.0 / 12.0, 4.0 / 12.0, 7.0 / 12.0};
    // Values of the min and max underlying values modeled
    private static double xMin = 0.0, xMax = 60.0;
    // Value of the interest rate and continuous dividend
    private static double r = 0.05, dividend = 0.0;
    // Define parameters for the integration step.
    private static int nXGgrid = 121, nTGrid = 3, nv = 3;
    private static int nint = nXGgrid - 1, n = 3 * nXGgrid;
    private static double[] xGrid = new double[nXGgrid];
    private static double dx;
    // Time dependency
    private static boolean[] timeDependence = new boolean[7];
    // Number of left/right boundary conditions
    private static int nlbcd = 3, nrbcd = 3;
    // Values of the underlying price where evaluations are made
    private static double[] evaluationPoints = {19.0, 20.0, 21.0};
    private static double epsfrac = .001;  //used to calc derivatives
    private static double dx2 = epsfrac * 0.5 * (xMin + xMax);
    private static double sqrt2pi = Math.sqrt(2. * Math.PI);
    private static String greekName[] = {
        " Value", " Delta", " Gamma", " Theta", " Charm",
        " Color", "  Vega", " Volga", " Vanna", "   Rho", " Speed", "Speed2"};

    // Time values for the options
```

```
private static int nt = 3;
private static double[] rex = new double[greekName.length];
private static double[] reavg = new double[greekName.length];
private static int[] irect = new int[greekName.length];

// Compute Constant Elasticity of Variance Model for Vanilla Call
public static void main(String args[]) throws Exception {
    // Define equally-spaced grid of points for the underlying price
    dx = (xMax - xMin) / ((double) nint);
    xGrid[0] = xMin;
    xGrid[nXGgrid - 1] = xMax;

    for (int i = 2; i <= nXGgrid - 1; i++) {
        xGrid[i - 1] = xGrid[i - 2] + dx;
    }
    System.out.printf("  Constant Elasticity of Variance Model"
            + " for Vanilla Call Option%n");
    System.out.printf(Locale.ENGLISH,
            "%7sInterest Rate:%7.3f   Continuous Dividend:%7.3f%n",
            " ", r, dividend);
    System.out.printf(Locale.ENGLISH,
            "%7sMinimum and Maximum Prices of Underlying:%7.2f%7.2f%n",
            " ", xMin, xMax);
    System.out.printf("%7sNumber of equally spaced spline knots:%4d%n",
            " ", nXGgrid - 1);
    System.out.printf("%7sNumber of unknowns:%4d%n%n", " ", n);
    System.out.printf(Locale.ENGLISH,
            "%7sTime in Years Prior to Expiration: %7.4f%7.4f%7.4f%n",
            " ", tGrid[0], tGrid[1], tGrid[2]); // tGrid[] = tau == T - t
    System.out.printf(Locale.ENGLISH,
            "%7sOption valued at Underlying Prices:%7.2f%7.2f%7.2f%n%n",
            " ", evaluationPoints[0], evaluationPoints[1],
            evaluationPoints[2]);

    /*
     * iquart = 0 : derivatives estimated with 3-point fitted parabola
     * iquart = 1 : derivatives estimated with 5-point fitted quartic
     *              polynomial
     */
    int iquart = 0;
    if (iquart == 0) {
        System.out.printf(Locale.ENGLISH,
                "       3 point (parabolic) estimate of "
                + "parameter derivatives;%n");
    } else {
        System.out.printf(Locale.ENGLISH,
                "       5 point (quartic) estimate of "
                + "parameter derivatives%n");
    }
    System.out.printf(Locale.ENGLISH, "       epsfrac = %11.8f%n", epsfrac);
    //alpha:  Black-Scholes
    for (int i2 = 1; i2 <= 3; i2++) /* Loop over volatility */ {
        for (int i3 = 1; i3 <= 3; i3++) /* Loop over strike price */ {
            calculateGreeks(i2, i3, iquart);
        }
    }
```

```
        System.out.println();
        for (int ig = 0; ig < 12; ig++) {
            reavg[ig] /= irect[ig];
            System.out.printf(Locale.ENGLISH, "%n Greek: " + greekName[ig]
                    + ";  avg rel err: %15.12f" + ";  max rel err: %15.12f",
                    reavg[ig], rex[ig]);
        }
        System.out.println();
    } // end main

    private static void calculateGreeks(int volatility, int strikePrice,
            int iquart) throws Exception {
        int i1 = 1;
        int nt = 3;
        if ((volatility == 1) && (strikePrice == 1)) {
            System.out.printf(Locale.ENGLISH, "%n"
                    + "                    Strike=%5.2f, Sigma="
                    + "%5.2f, Alpha=%5.2f:",
                    strikePrices[strikePrice - 1], sigma[volatility - 1],
                    alpha[i1 - 1]);
            System.out.printf(Locale.ENGLISH, "%n"
                    + "                years to expiration: "
                    + "   %7.4f       %7.4f       %7.4f"
                    + "%n",
                    tGrid[0], tGrid[1], tGrid[2]);
        }
        /* Loop over t derivative index */
        for (int iTDeriv = 0; iTDeriv < 2; iTDeriv++) {
            int iSDerivMax = 4 - iTDeriv;
            /* Loop over S derivative index */
            for (int iSDeriv = 0; iSDeriv < iSDerivMax; iSDeriv++) {
                int iSigDerivMax = 1;
                if (iTDeriv == 0) {
                    if (iSDeriv == 0) {
                        iSigDerivMax = 3;
                    }
                    if (iSDeriv == 1) {
                        iSigDerivMax = 2;
                    }
                }
                //Loop over sigma deriv index
                for (int iSigDeriv = 0; iSigDeriv < iSigDerivMax; iSigDeriv++) {
                    int iRDerivMax = 1;
                    if (iTDeriv == 0 && iSDeriv == 0 && iSigDeriv == 0) {
                        iRDerivMax = 2;
                    }
                    // Loop over r derivative index
                    for (int iRDeriv = 0; iRDeriv < iRDerivMax; iRDeriv++) {
                        int ispeedmin = 0;
                        int ispeedmax = 1;
                        if (iTDeriv == 0 && iSDeriv == 2) {
                            ispeedmax = 2;
                        }
                        if (iTDeriv == 0 && iSDeriv == 3) {
                            ispeedmin = 2;
                            ispeedmax = 3;
```

```
}
// Loop over speed index
for (int ispeed = ispeedmin; ispeed < ispeedmax;
        ispeed++) {
    // Pass data through into evaluation methods.
    double[] userData = new double[6];
    userData[0] = strikePrices[strikePrice - 1];
    userData[1] = xMax;
    userData[2] = sigma[volatility - 1];
    userData[3] = alpha[i1 - 1];
    userData[4] = r;
    userData[5] = dividend;
    double[][] splineValuesOption
            = new double[nTGrid][nv];
    double[][] splineValuesOptionP
            = new double[nTGrid][];
    double[][] splineValuesOptionM
            = new double[nTGrid][];
    double[][] splineValuesOptionPP
            = new double[nTGrid][];
    double[][] splineValuesOptionMM
            = new double[nTGrid][];
    double[] xGridP = new double[nXGgrid];
    double[] xGridM = new double[nXGgrid];
    double[] evaluationPointsP = new double[nv];
    double[] evaluationPointsM = new double[nv];
    double[] xGridPP = new double[nXGgrid];
    double[] xGridMM = new double[nXGgrid];
    double[] evaluationPointsPP = new double[nv];
    double[] evaluationPointsMM = new double[nv];
    double xMaxP = xMax;
    double xMaxM = xMax;
    double xMaxPP = xMax, xMaxMM = xMax;

    // Evaluate FK solution at vector evaluationPoints,
    // at  each time value prior to expiration.
    if ((iSigDeriv != 1 || iSDeriv == 1 || iRDeriv != 1)
            && (ispeed == 0 || ispeed == 2)) {

        FeynmanKac callOption = new FeynmanKac(
                new MyCoefficients(userData));
        //Right boundary condition time-dependent
        timeDependence[5] = true;
        callOption.setTimeDependence(timeDependence);
        callOption.computeCoefficients(nlbcd, nrbcd,
                new MyBoundaries(userData), xGrid,
                tGrid);
        double[][] optionCoefficients
                = new double[tGrid.length + 1][3
                * xGrid.length];
        if (iTDeriv == 0) {
            optionCoefficients
                    = callOption.
                    getSplineCoefficients();
        } else {
            optionCoefficients
```

```
                = callOption.
                getSplineCoefficientsPrime();
    }
    for (int i = 0; i < nTGrid; i++) {
        // FK option values for tau = tGrid[i]:
        splineValuesOption[i]
                = callOption.getSplineValue(
                        evaluationPoints,
                        optionCoefficients[i + 1],
                        iSDeriv);
    }
}
if (iSigDeriv > 0 || iRDeriv > 0 || ispeed == 1) {
    System.arraycopy(xGrid, 0, xGridP, 0, nXGgrid);
    System.arraycopy(xGrid, 0, xGridM, 0, nXGgrid);
    System.arraycopy(evaluationPoints, 0,
            evaluationPointsP, 0, nv);
    System.arraycopy(evaluationPoints, 0,
            evaluationPointsM, 0, nv);
    if (ispeed == 1) {
        for (int i = 0; i < nXGgrid; i++) {
            xGridP[i] = xGrid[i] + dx2;
            xGridM[i] = xGrid[i] - dx2;
        }
        for (int i = 0; i < nv; i++) {
            evaluationPointsP[i]
                    = evaluationPoints[i] + dx2;
            evaluationPointsM[i]
                    = evaluationPoints[i] - dx2;
        }
        xMaxP = xMax + dx2;
        xMaxM = xMax - dx2;
    }

    userData[1] = xMaxP;
    // calculate spline values for
    // sigmaP = sigma[i2-1]*(1. + epsfrac)
    // or rP = r*(1. + epsfrac)
    if (iSigDeriv > 0) {
        userData[2] = sigma[volatility - 1]
                * (1. + epsfrac);
    } else if (iRDeriv > 0) {
        userData[4] = r * (1. + epsfrac);
    }
    FeynmanKac callOptionP
            = new FeynmanKac(
                    new MyCoefficients(userData));
    //Right boundary condition time-dependent
    timeDependence[5] = true;
    callOptionP.setTimeDependence(timeDependence);
    callOptionP.computeCoefficients(nlbcd, nrbcd,
            new MyBoundaries(userData), xGridP,
            tGrid);
    double[][] optionCoefficientsP
            = new double[tGrid.length + 1][3
            * xGrid.length];
```

```
if (iTDeriv == 0) {
    optionCoefficientsP
            = callOptionP.
            getSplineCoefficients();
} else {
    optionCoefficientsP
            = callOptionP.
            getSplineCoefficientsPrime();
}
for (int i = 0; i < nTGrid; i++) {
    // FK option values for tau = tGrid[i]:
    splineValuesOptionP[i]
            = callOptionP.getSplineValue(
                    evaluationPointsP,
                    optionCoefficientsP[i + 1],
                    iSDeriv);
}

userData[1] = xMaxM;
// calculate spline values for
// sigmaM = sigma[i2-1]*(1. - epsfrac)   or
// rM = r*(1. - epsfrac):
if (iSigDeriv > 0) {
    userData[2] = sigma[volatility - 1]
            * (1. - epsfrac);
} else {
    userData[4] = r * (1. - epsfrac);
}
FeynmanKac callOptionM
        = new FeynmanKac(
                new MyCoefficients(userData));
//Right boundary condition time-dependent
timeDependence[5] = true;
callOptionM.setTimeDependence(timeDependence);
callOptionM.computeCoefficients(nlbcd, nrbcd,
        new MyBoundaries(userData), xGridM,
        tGrid);
double[][] optionCoefficientsM
        = new double[tGrid.length + 1][3
        * xGrid.length];
if (iTDeriv == 0) {
    optionCoefficientsM
            = callOptionM.
            getSplineCoefficients();
} else {
    optionCoefficientsM
            = callOptionM.
            getSplineCoefficientsPrime();
}
for (int i = 0; i < nTGrid; i++) {
    // FK option values for tau = tGrid[i]:
    splineValuesOptionM[i]
            = callOptionM.
            getSplineValue(
                    evaluationPointsM,
                    optionCoefficientsM[i + 1],
```

```
                    iSDeriv);
}
if (iquart == 1) {
    System.arraycopy(xGrid, 0, xGridPP, 0,
            nXGgrid);
    System.arraycopy(xGrid, 0, xGridMM, 0,
            nXGgrid);
    System.arraycopy(evaluationPoints, 0,
            evaluationPointsPP, 0, nv);
    System.arraycopy(evaluationPoints, 0,
            evaluationPointsMM, 0, nv);
    if (ispeed == 1) {
        for (int i = 0; i < nXGgrid; i++) {
            xGridPP[i] = xGrid[i] + 2. * dx2;
            xGridMM[i] = xGrid[i] - 2. * dx2;
        }
        for (int i = 0; i < nv; i++) {
            evaluationPointsPP[i]
                    = evaluationPoints[i]
                    + 2. * dx2;
            evaluationPointsMM[i]
                    = evaluationPoints[i]
                    - 2. * dx2;
        }
        xMaxPP = xMax + 2. * dx2;
        xMaxMM = xMax - 2. * dx2;
    }

    userData[1] = xMaxPP;
    if (iSigDeriv > 0) {
        // calculate spline values for
        // sigmaPP = sigma[i2-1]
        //          *(1. + 2.*epsfrac):
        userData[2]
                = sigma[volatility - 1]
                * (1. + 2. * epsfrac);
    } else if (iRDeriv > 0) {
        // calculate spline values for
        // rPP = r*(1. + 2.*epsfrac):
        userData[4] = r * (1. + 2. * epsfrac);
    }
    FeynmanKac callOptionPP
            = new FeynmanKac(
                    new MyCoefficients(userData));
    //Right boundary condition time-dependent
    timeDependence[5] = true;
    callOptionPP.
            setTimeDependence(timeDependence);
    callOptionPP.
            computeCoefficients(nlbcd, nrbcd,
                    new MyBoundaries(userData),
                    xGridPP, tGrid);
    double[][] optionCoefficientsPP
            = new double[tGrid.length + 1][3
            * xGrid.length];
    if (iTDeriv == 0) {
```

```
    optionCoefficientsPP
            = callOptionPP.
            getSplineCoefficients();
} else {
    optionCoefficientsPP
            = callOptionPP.
            getSplineCoefficientsPrime();
}
for (int i = 0; i < nTGrid; i++) {
    // FK option values for tau = tGrid[i]:
    splineValuesOptionPP[i]
            = callOptionPP.getSplineValue(
                    evaluationPointsPP,
                    optionCoefficientsPP[i + 1],
                    iSDeriv);
}

userData[1] = xMaxMM;
// calculate spline values for
// sigmaMM = sigma[i2-1]*(1. - 2.*epsfrac)
// or rMM = r*(1. - 2.*epsfrac)
if (iSigDeriv > 0) {
    userData[2]
            = sigma[volatility - 1]
            * (1. - 2. * epsfrac);
} else if (iRDeriv > 0) {
    userData[4] = r * (1. - 2. * epsfrac);
}
FeynmanKac callOptionMM
        = new FeynmanKac(
                new MyCoefficients(userData)
        );
//Right boundary condition time-dependent
timeDependence[5] = true;
callOptionMM.
        setTimeDependence(timeDependence);
callOptionMM.
        computeCoefficients(nlbcd, nrbcd,
                new MyBoundaries(userData),
                xGridMM, tGrid);
double[][] optionCoefficientsMM
        = new double[tGrid.length + 1][3
        * xGrid.length];
if (iTDeriv == 0) {
    optionCoefficientsMM
            = callOptionMM.
            getSplineCoefficients();
} else {
    optionCoefficientsMM
            = callOptionMM.
            getSplineCoefficientsPrime();
}
for (int i = 0; i < nTGrid; i++) {
    // FK option values for tau = tGrid[i]:
    splineValuesOptionMM[i]
            = callOptionMM.getSplineValue(
```

```
                                evaluationPointsMM,
                                optionCoefficientsMM[i + 1],
                                iSDeriv);
            }
        }
    }

    if (iSigDeriv == 1 || iRDeriv == 1 || ispeed == 1) {
        double eps = 0., f11 = 0., f12 = 0.;
        if (iSigDeriv == 1) {
            eps = sigma[volatility - 1] * epsfrac;
        }
        if (iRDeriv == 1) {
            eps = r * epsfrac;
        }
        if (ispeed == 1) {
            eps = dx2;
        }
        for (int i = 0; i < nTGrid; i++) {
            for (int j = 0; j < nv; j++) {
                f11 = (splineValuesOptionP[i][j]
                        - splineValuesOptionM[i][j])
                        / (2. * eps);
                if (iquart == 0) {
                    splineValuesOption[i][j] = f11;
                } else {
                    f12 = (splineValuesOptionPP[i][j]
                            - splineValuesOptionMM[i][j])
                            / (4. * eps);
                    splineValuesOption[i][j] = (4.
                            * f11 - f12) / 3.;
                }
            }
        }
    }

    if (iSigDeriv == 2) {
        double eps = sigma[volatility - 1] * epsfrac;
        double f21 = 0.;
        double f22 = 0.;
        for (int i = 0; i < nTGrid; i++) {
            for (int j = 0; j < nv; j++) {
                f21 = (splineValuesOptionP[i][j]
                        + splineValuesOptionM[i][j] - 2.
                        * splineValuesOption[i][j])
                        / (eps * eps);
                if (iquart == 0) {
                    splineValuesOption[i][j] = f21;
                } else {
                    f22 = (splineValuesOptionPP[i][j]
                            + splineValuesOptionMM[i][j]
                            - 2.
                            * splineValuesOption[i][j])
                            / (4. * eps * eps);
                    splineValuesOption[i][j]
                            = (4. * f21 - f22) / 3.;
```

```
            }
        }
    }
}

// Evaluate BS solution at vector evaluationPoints,
// at each time value prior to expiration.
double[][] BSValuesOption = new double[nTGrid][nv];
for (int i = 0; i < nTGrid; i++) {
    /*
     *  Black-Scholes (BS) European call option
     *  value = ValBSEC(S,t) =
     *              exp(-q*tau)*S*N01CDF(d1) -
     *              exp(-r*tau)*K*N01CDF(d2),
     *  where:
     *    tau = time to maturity = T - t;
     *    q = annual dividend yield;
     *    r = risk free rate;
     *    K = strike price;
     *    S = stock price;
     *    N01CDF(x) = N(0,1)_CDF(x);
     *    d1 = ( log( S/K ) +
     *         ( r - q + 0.5*sigma**2 )*tau ) /
     *         ( sigma*sqrt(tau) );
     *    d2 = d1 - sigma*sqrt(tau)
     */
    // BS option values for tau = tGrid[i]:
    double tau = tGrid[i];
    double sigsqtau
            = Math.pow(sigma[volatility - 1], 2)
            * tau;
    double sqrt_sigsqtau = Math.sqrt(sigsqtau);
    double sigsq = sigma[volatility - 1]
            * sigma[volatility - 1];
    for (int j = 0; j < nv; j++) {
        // Values of the underlying price where
        // evaluations are made:
        double S = evaluationPoints[j];
        double d1 = (Math.log(S
                / strikePrices[strikePrice - 1])
                + (r - dividend) * tau + 0.5
                * sigsqtau) / sqrt_sigsqtau;
        double n01pdf_d1 = Math.exp(-0.5 * d1 * d1)
                / sqrt2pi;
        double nu = Math.exp(-dividend * tau) * S
                * n01pdf_d1 * Math.sqrt(tau);
        if (iTDeriv == 0) {
            if (iSDeriv == 0) {
                double d2 = d1 - sqrt_sigsqtau;
                if (iSigDeriv == 0) {
                    if (iRDeriv == 0) {
                        BSValuesOption[i][j]
                                = Math.exp(-dividend
                                        * tau) * S
                                * Cdf.normal(d1)
                                - Math.exp(-r * tau)
```

```
                        * strikePrices[strikePrice - 1]
                        * Cdf.normal(d2);
            } else {
                BSValuesOption[i][j]
                        = Math.exp(-r * tau)
                        * strikePrices[strikePrice - 1]
                        * tau
                        * Cdf.normal(d2);
            }
        } else if (iSigDeriv == 1) {
            //greek = Vega
            BSValuesOption[i][j] = nu;
        } else if (iSigDeriv == 2) {
            //greek = Volga
            BSValuesOption[i][j] = nu * d1
                    * d2
                    / sigma[volatility - 1];
        }
    } else if (iSDeriv == 1) {
        //greek = delta
        if (iSigDeriv == 0) {
            BSValuesOption[i][j]
                    = Math.exp(-dividend
                            * tau)
                    * Cdf.normal(d1);
        } else if (iSigDeriv == 1) {
            //greek = Vanna
            BSValuesOption[i][j]
                    = (nu / S) * (1. - d1
                    / sqrt_sigsqtau);
        }
    } else if (iSDeriv == 2) {
        if (ispeed == 0) {  //greek = gamma
            BSValuesOption[i][j]
                    = Math.exp(-dividend
                            * tau)
                    * n01pdf_d1
                    / (S * sqrt_sigsqtau);
        } else { //greek = speed
            BSValuesOption[i][j]
                    = -Math.exp(-dividend
                            * tau)
                    * n01pdf_d1 * (1. + d1
                    / sqrt_sigsqtau)
                    / (S * S
                    * sqrt_sigsqtau);
        }
    } else if (iSDeriv == 3
            && ispeed == 2) {
        //greek = speed
        BSValuesOption[i][j]
                = -Math.exp(-dividend * tau)
                * n01pdf_d1
                * (1. + d1 / sqrt_sigsqtau)
                / (S * S * sqrt_sigsqtau);
    }
```

```
            } else if (iTDeriv == 1) {
                double d2 = d1 - sqrt_sigsqtau;
                if (iSDeriv == 0) { //greek = theta
                    BSValuesOption[i][j]
                            = Math.exp(-dividend * tau)
                            * S
                            * (dividend * Cdf.normal(d1)
                            - 0.5 * sigsq
                            * n01pdf_d1 / sqrt_sigsqtau)
                            - r * Math.exp(-r * tau)
                            * strikePrices[strikePrice
                            - 1]
                            * Cdf.normal(d2);
                } else if (iSDeriv == 1) {
                    //greek = charm
                    BSValuesOption[i][j]
                            = Math.exp(-dividend * tau)
                            * (-dividend * Cdf.normal(d1)
                            + n01pdf_d1 * ((r
                            - dividend) * tau - 0.5
                            * d2 * sqrt_sigsqtau)
                            / (tau * sqrt_sigsqtau));
                } else if (iSDeriv == 2) {
                    //greek = color
                    BSValuesOption[i][j]
                            = -Math.exp(-dividend * tau)
                            * n01pdf_d1
                            * (2. * dividend * tau + 1.
                            + d1 * (2. * (r - dividend)
                            * tau - d2 * sqrt_sigsqtau)
                            / sqrt_sigsqtau)
                            / (2. * S * tau
                            * sqrt_sigsqtau);
                }
            }
        }
    }

    double relerrmax = 0.;
    int gNi = 3 * iTDeriv + iSDeriv;
    if (iSigDeriv == 1 && iSDeriv == 0) {
        gNi = 6; //vega
    }
    if (iSigDeriv == 2 && iSDeriv == 0) {
        gNi = 7; //volga
    }
    if (iSigDeriv == 1 && iSDeriv == 1) {
        gNi = 8; //vanna
    }
    if (iRDeriv == 1) {
        gNi = 9; //rho
    }
    if (ispeed >= 1) {
        gNi = 9 + ispeed; //speed
    }
    for (int i = 0; i < nv; i++) {
```

```
                                for (int j = 0; j < nt; j++) {
                                    double sVo = splineValuesOption[j][i], BSVo
                                            = BSValuesOption[j][i];
                                    //greeks(itd=1) ~ d/dtau = -d/dt
                                    //for iSDeriv > 0:
                                    if (iTDeriv == 1 && iSDeriv > 0) {
                                        sVo = -sVo;
                                    }
                                    double relerr
                                            = Math.abs((sVo - BSVo) / BSVo);
                                    if (relerr > relerrmax) {
                                        relerrmax = relerr;
                                    }
                                    reavg[gNi] += relerr;
                                    irect[gNi] += 1;
                                }
                            }
                            if (relerrmax > rex[gNi]) {
                                rex[gNi] = relerrmax;
                            }

                            if ((volatility == 1) && (strikePrice == 1)) {
                                for (int i = 0; i < nv; i++) {
                                    System.out.printf(" underlying price:"
                                            + " %4.1f;  FK " + greekName[gNi]
                                            + ": ", evaluationPoints[i]);
                                    for (int j = 0; j < nt; j++) {
                                        double sVo = splineValuesOption[j][i];
                                        //greeks(itd=1) ~ d/dtau = -d/dt
                                        //for isd > 0:
                                        if (iTDeriv == 1 && iSDeriv > 0) {
                                            sVo = -sVo;
                                        }
                                        System.out.printf(Locale.ENGLISH,
                                                "%13.10f ", sVo);
                                    }
                                    System.out.println();
                                    System.out.printf("             "
                                            + "            BS "
                                            + greekName[gNi] + ": ");
                                    for (int j = 0; j < nt; j++) {
                                        System.out.printf(Locale.ENGLISH,
                                                "%13.10f ",
                                                BSValuesOption[j][i]);
                                    }
                                    System.out.println();
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    static class MyCoefficients implements FeynmanKac.PdeCoefficients {
```

```
        final double zero = 0.0, half = 0.5;
        private double sigma, strikePrice, interestRate;
        private double alpha, dividend;

        public MyCoefficients(double[] myData) {
            this.strikePrice = myData[0];
            this.sigma = myData[2];
            this.alpha = myData[3];
            this.interestRate = myData[4];
            this.dividend = myData[5];
        }

        // The coefficient sigma(x)
        public double sigma(double x, double t) {
            return (sigma * Math.pow(x, alpha * half));
        }

        // The coefficient derivative d(sigma) / dx
        public double sigmaPrime(double x, double t) {
            return (half * alpha * sigma * Math.pow(x, alpha * half - 1.0));
        }

        // The coefficient mu(x)
        public double mu(double x, double t) {
            return ((interestRate - dividend) * x);
        }

        // The coefficient kappa(x)
        public double kappa(double x, double t) {
            return interestRate;
        }
    }

    static class MyBoundaries implements FeynmanKac.Boundaries {

        private double xMax, df, interestRate, strikePrice;

        public MyBoundaries(double[] myData) {
            this.strikePrice = myData[0];
            this.xMax = myData[1];
            this.interestRate = myData[4];
        }

        public void leftBoundaries(double time, double[][] bndCoeffs) {
            bndCoeffs[0][0] = 1.0;
            bndCoeffs[0][1] = 0.0;
            bndCoeffs[0][2] = 0.0;
            bndCoeffs[0][3] = 0.0;
            bndCoeffs[1][0] = 0.0;
            bndCoeffs[1][1] = 1.0;
            bndCoeffs[1][2] = 0.0;
            bndCoeffs[1][3] = 0.0;
            bndCoeffs[2][0] = 0.0;
            bndCoeffs[2][1] = 0.0;
            bndCoeffs[2][2] = 1.0;
```

```
            bndCoeffs[2][3] = 0.0;
        }

        public void rightBoundaries(double time, double[][] bndCoeffs) {
            df = Math.exp(interestRate * time);
            bndCoeffs[0][0] = 1.0;
            bndCoeffs[0][1] = 0.0;
            bndCoeffs[0][2] = 0.0;
            bndCoeffs[0][3] = xMax - df * strikePrice;
            bndCoeffs[1][0] = 0.0;
            bndCoeffs[1][1] = 1.0;
            bndCoeffs[1][2] = 0.0;
            bndCoeffs[1][3] = 1.0;
            bndCoeffs[2][0] = 0.0;
            bndCoeffs[2][1] = 0.0;
            bndCoeffs[2][2] = 1.0;
            bndCoeffs[2][3] = 0.0;
        }

        public double terminal(double x) {
            double zero = 0.0;
            // The payoff function
            double value = Math.max(x - strikePrice, zero);
            return value;
        }
    }
}
```

Output

```
  Constant Elasticity of Variance Model for Vanilla Call Option
       Interest Rate:  0.050   Continuous Dividend:  0.000
       Minimum and Maximum Prices of Underlying:   0.00  60.00
       Number of equally spaced spline knots: 120
       Number of unknowns: 363

       Time in Years Prior to Expiration:  0.0833 0.3333 0.5833
       Option valued at Underlying Prices:  19.00  20.00  21.00

       3 point (parabolic) estimate of parameter derivatives;
       epsfrac =  0.00100000

                  Strike=15.00, Sigma= 0.20, Alpha= 2.00:
                years to expiration:     0.0833       0.3333       0.5833
 underlying price: 19.0;  FK  Value:  4.0623732450  4.2575924184  4.4733805278
                          BS  Value:  4.0623732509  4.2575929678  4.4733814062
 underlying price: 20.0;  FK  Value:  5.0623700127  5.2505145764  5.4492418798
                          BS  Value:  5.0623700120  5.2505143129  5.4492428547
 underlying price: 21.0;  FK  Value:  6.0623699727  6.2485587059  6.4385718831
                          BS  Value:  6.0623699726  6.2485585270  6.4385720688
 underlying price: 19.0;  FK    Rho:  1.2447807022  4.8365676562  8.0884594648
                          BS    Rho:  1.2447806658  4.8365650322  8.0884502627
 underlying price: 20.0;  FK    Rho:  1.2448021850  4.8929216545  8.3041708173
                          BS    Rho:  1.2448021908  4.8929245641  8.3041638392
 underlying price: 21.0;  FK    Rho:  1.2448024992  4.9107294561  8.4114197621
```

```
                         BS    Rho:  1.2448024996  4.9107310444  8.4114199038
underlying price: 19.0;  FK   Vega:  0.0003289870  0.3487168323  1.1153520921
                         BS   Vega:  0.0003295819  0.3487535501  1.1153536190
underlying price: 20.0;  FK   Vega:  0.0000056652  0.1224632724  0.6032458218
                         BS   Vega:  0.0000056246  0.1224675413  0.6033084039
underlying price: 21.0;  FK   Vega:  0.0000000623  0.0376974472  0.3028275297
                         BS   Vega:  0.0000000563  0.0376857196  0.3028629419
underlying price: 19.0;  FK  Volga:  0.0286253243  8.3705172571 16.7944557372
                         BS  Volga:  0.0286064650  8.3691191978 16.8219823169
underlying price: 20.0;  FK  Volga:  0.0007137846  4.2505027498 12.9315444575
                         BS  Volga:  0.0007186004  4.2519372748 12.9612638820
underlying price: 21.0;  FK  Volga:  0.0000100364  1.7613084768  8.6626165796
                         BS  Volga:  0.0000097963  1.7617504949  8.6676581034
underlying price: 19.0;  FK  Delta:  0.9999864098  0.9877532309  0.9652249945
                         BS  Delta:  0.9999863811  0.9877520034  0.9652261127
underlying price: 20.0;  FK  Delta:  0.9999998142  0.9964646548  0.9842482622
                         BS  Delta:  0.9999998151  0.9964644003  0.9842476147
underlying price: 21.0;  FK  Delta:  0.9999999983  0.9990831687  0.9932459040
                         BS  Delta:  0.9999999985  0.9990834124  0.9932451927
underlying price: 19.0;  FK  Vanna: -0.0012418872 -0.3391850563 -0.6388552010
                         BS  Vanna: -0.0012431594 -0.3391932673 -0.6387423326
underlying price: 20.0;  FK  Vanna: -0.0000244490 -0.1366771953 -0.3945466660
                         BS  Vanna: -0.0000244825 -0.1367114682 -0.3945405194
underlying price: 21.0;  FK  Vanna: -0.0000002905 -0.0466333335 -0.2187406645
                         BS  Vanna: -0.0000002726 -0.0466323413 -0.2187858632
underlying price: 19.0;  FK  Gamma:  0.0000543456  0.0144908955  0.0264849216
                         BS  Gamma:  0.0000547782  0.0144911447  0.0264824761
underlying price: 20.0;  FK  Gamma:  0.0000008315  0.0045912854  0.0129288434
                         BS  Gamma:  0.0000008437  0.0045925328  0.0129280372
underlying price: 21.0;  FK  Gamma:  0.0000000080  0.0012817012  0.0058860348
                         BS  Gamma:  0.0000000077  0.0012818272  0.0058865489
underlying price: 19.0;  FK  Speed: -0.0002127758 -0.0157070513 -0.0181086989
                         BS  Speed: -0.0002123854 -0.0156192867 -0.0179536520
underlying price: 20.0;  FK  Speed: -0.0000037305 -0.0056184183 -0.0098403706
                         BS  Speed: -0.0000037568 -0.0055859333 -0.0097472434
underlying price: 21.0;  FK  Speed: -0.0000000385 -0.0017185470 -0.0048615664
                         BS  Speed: -0.0000000378 -0.0017082128 -0.0048130214
underlying price: 19.0;  FK Speed2: -0.0002310655 -0.0156276977 -0.0179516855
                         BS Speed2: -0.0002123854 -0.0156192867 -0.0179536520
underlying price: 20.0;  FK Speed2: -0.0000043215 -0.0055923924 -0.0097502997
                         BS Speed2: -0.0000037568 -0.0055859333 -0.0097472434
underlying price: 21.0;  FK Speed2: -0.0000000475 -0.0017117661 -0.0048153106
                         BS Speed2: -0.0000000378 -0.0017082128 -0.0048130214
underlying price: 19.0;  FK  Theta: -0.7472631891 -0.8301000450 -0.8845209253
                         BS  Theta: -0.7472638978 -0.8301108199 -0.8844992143
underlying price: 20.0;  FK  Theta: -0.7468881086 -0.7706770630 -0.8152217385
                         BS  Theta: -0.7468880640 -0.7706789470 -0.8152097697
underlying price: 21.0;  FK  Theta: -0.7468815742 -0.7479185416 -0.7728950748
                         BS  Theta: -0.7468815673 -0.7479153725 -0.7728982104
underlying price: 19.0;  FK  Charm: -0.0014382828 -0.0879903285 -0.0843323992
                         BS  Charm: -0.0014397520 -0.0879913927 -0.0843403333
underlying price: 20.0;  FK  Charm: -0.0000284881 -0.0364107814 -0.0547260337
                         BS  Charm: -0.0000285354 -0.0364209077 -0.0547074804
underlying price: 21.0;  FK  Charm: -0.0000003396 -0.0126436426 -0.0313343015
                         BS  Charm: -0.0000003190 -0.0126437838 -0.0313252716
underlying price: 19.0;  FK  Color:  0.0051622176  0.0685064195  0.0299871130
```

```
                          BS  Color:  0.0051777484  0.0684737183  0.0300398444
underlying price: 20.0;  FK  Color:  0.0001188761  0.0355826975  0.0274292189
                          BS  Color:  0.0001205713  0.0355891884  0.0274307898
underlying price: 21.0;  FK  Color:  0.0000015431  0.0143174419  0.0190897160
                          BS  Color:  0.0000015141  0.0143247729  0.0190752019


Greek:  Value;  avg rel err:  0.000146170676;  max rel err:  0.009030693732
Greek:  Delta;  avg rel err:  0.000035817236;  max rel err:  0.001158483024
Greek:  Gamma;  avg rel err:  0.001088461868;  max rel err:  0.044851604910
Greek:  Theta;  avg rel err:  0.000054196357;  max rel err:  0.001412847201
Greek:  Charm;  avg rel err:  0.001213382638;  max rel err:  0.064579338827
Greek:  Color;  avg rel err:  0.003323753784;  max rel err:  0.136355681844
Greek:   Vega;  avg rel err:  0.001513788865;  max rel err:  0.106176227011
Greek:  Volga;  avg rel err:  0.058530479593;  max rel err:  1.639567620281
Greek:  Vanna;  avg rel err:  0.001062052173;  max rel err:  0.065672253096
Greek:    Rho;  avg rel err:  0.000146868218;  max rel err:  0.009438786906
Greek:  Speed;  avg rel err:  0.007251538925;  max rel err:  0.123630425554
Greek: Speed2;  avg rel err:  0.008429577505;  max rel err:  0.255722275986
```

FeynmanKac.PdeCoefficients interface

`public interface com.imsl.math.FeynmanKac.PdeCoefficients`

Public interface for user supplied PDE coefficients in the Feynman-Kac PDE.

Methods

kappa

`public double kappa(double x, double t)`

Description

Returns the value of the κ coefficient at the given point.

Time dependency of κ can be controlled via method `setTimeDependence`. Use of this method will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each `t` value.

Parameters

`x` – a `double`, the point in space at which κ is to be evaluated.

`t` – a `double`, the time point at which κ is to be evaluated.

Returns

a `double`, the value of κ at `(x,t)`.

mu

```
public double mu(double x, double t)
```

Description

Returns the value of the μ coefficient at the given point.

Time dependency of μ can be controlled via method `setTimeDependence`. Use of this method will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each `t` value.

Parameters

`x` – a `double`, the point in space at which μ is to be evaluated.

`t` – a `double`, the time point at which μ is to be evaluated.

Returns

a `double`, the value of μ at `(x,t)`.

sigma

```
public double sigma(double x, double t)
```

Description

Returns the value of the σ coefficient at the given point.

Time dependency of σ can be controlled via method `setTimeDependence`. Use of this method will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each `t` value.

Parameters

`x` – a `double`, the point in space at which σ is to be evaluated.

`t` – a `double`, the time point at which σ is to be evaluated.

Returns

a `double`, the value of σ at `(x,t)`.

sigmaPrime

```
public double sigmaPrime(double x, double t)
```

Description

Returns the value of $\sigma' = \frac{\partial \sigma(x,t)}{\partial x}$ at the given point.

Time dependency of σ' can be controlled via method `setTimeDependence`. Use of this method will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each `t` value.

Parameters

`x` – a `double`, the point in space at which σ' is to be evaluated.

`t` – a `double`, the time point at which σ' is to be evaluated.

Returns

a `double`, the value of σ' at `(x,t)`.

FeynmanKac.Boundaries interface

`public interface com.imsl.math.FeynmanKac.Boundaries`

Public interface for user supplied boundary coefficients and terminal condition the PDE must satisfy.

Methods

leftBoundaries

`public void leftBoundaries(double time, double[][] coefficients)`

Description

Returns the coefficient values of the left boundary conditions. There are `numLeftBounds` conditions specified at the left end, $x_{\min}$. The left boundary conditions are

$$a_i(x,t)f + b_i(x,t)f_x + c_i(x,t)f_{xx} = d_i(x,t),\ x = x_{\min},\ 1 \le i \le \text{numLeftBounds}.$$

Time dependency of the boundary coefficients can be controlled via method `setTimeDependence`. Use of this method will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each `t` value.

Parameters

`time` – a `double`, the time point at which the boundary coefficients are to be evaluated.

`coefficients` – an output `double` array of dimension `numLeftBounds` by 4 containing the computed boundary coefficient values. The coefficients are stored row-wise according to the matrix scheme

$$(a_i(x_{\min},t), b_i(x_{\min},t), c_i(x_{\min},t), d_i(x_{\min},t))_{1\le i\le \text{numLeftBounds}}.$$

rightBoundaries

`public void rightBoundaries(double time, double[][] coefficients)`

Description

Returns the coefficient values of the right boundary conditions. There are `numRightBounds` conditions specified at the right end, $x_{\max}$. The right boundary conditions are

$$a_i(x,t)f + b_i(x,t)f_x + c_i(x,t)f_{xx} = d_i(x,t),\ x = x_{\max},\ 1 \le i \le \text{numRightBounds}.$$

Time dependency of the boundary coefficients can be controlled via method `setTimeDependence`. Use of this method will usually yield a more efficient algorithm because some finite element matrices do not have to be reassembled for each `t` value.

Parameters

`time` – a `double`, the time point at which the boundary coefficients are to be evaluated.

`coefficients` – an output `double` array of dimension `numRightBounds` by 4, containing the computed boundary coefficient values. The coefficients are stored row-wise according to the matrix scheme

$$(a_i(x_{\max},t), b_i(x_{\max},t), c_i(x_{\max},t), d_i(x_{\max},t))_{1\le i\le \text{numRightBounds}}.$$

terminal

```
public double terminal(double z)
```

Description

Returns the terminal condition value.

Parameter

`z` – a `double` scalar, the point in `x-` direction, where the terminal condition is evaluated.

Returns

a `double` scalar, the value of the terminal condition $p(x)$ at point `z`.

FeynmanKac.InitialData interface

```
public interface com.imsl.math.FeynmanKac.InitialData
```

Public interface for adjustment of initial data or as an opportunity for output during the integration steps.

Method

init

```
public void init(double[] xGrid, double[] tGrid, double time, double[] yprime,
double[] y, double[] absoluteErrorTolerance, double[] relativeErrorTolerance)
```

Description

Method that allows for adjustment of initial data or as an opportunity for output during the integration steps.

Parameters

`xGrid` – a `double` array containing the grid points in the x-direction.

`tGrid` – a `double` array containing the grid points in the t-direction.

`time` – a `double`, containing the time point for the evaluation.

yprime – a double array of length 3*xGrid.length containing the derivatives of the Hermite quintic spline coefficients at time point time.

y – a double array of length 3*xGrid.length containing the coefficients of the Hermite quintic spline at time point time.

absoluteErrorTolerance – a double array of length 3*xGrid.length containing absolute error tolerances.

relativeErrorTolerance – a double array of length 3*xGrid.length containing relative error tolerances.

FeynmanKac.ForcingTerm interface

```
public interface com.imsl.math.FeynmanKac.ForcingTerm
```

Public interface for non-zero forcing term in the Feynman-Kac equation.

Method

force

```
public void force(int interval, double[] y, double time, double width, double[]
xlocal, double[] qw, double[][] u, double[] phi, double[][] dphi)
```

Description

Computes approximations to the forcing term $\phi(f,x,t)$ and its derivative $\partial\phi/\partial y$.

Parameters

interval – an int, the index related to the integration interval [xGrid[interval-1], xGrid[interval]].

y – an input double array of length 3*xGrid.length containing the coefficients of the Hermite quintic spline representing the solution of the Feynman-Kac equation at time point time. For each

$$x \in [x_i, x_{i+1}],\, h_i = x_{i+1} - x_i, z_i = (x - x_i)/h_i,\, i = 1, \ldots, \text{xGrid.length} - 1$$

the approximate solution is locally defined by

$$f(x,t) = f_i b_0(z) + f_{i+1} b_0(1-z) + h_i f_i' b_1(z) - h_i f_{i+1}' b_1(1-z) + h_i^2 f_i'' b_2(z) + h_i^2 f_{i+1}'' b_2(1-z).$$

The values $y_i = f_i$, $y_{i+1} = f_i'$, $y_{i+2} = f_i''$, $i = 1, 4, 7, \ldots, 3 \cdot \text{xGrid.length} - 2$, are stored as successive triplets in y.

time – a double, the time point.

width – a double, the width of the integration interval, width=xGrid[interval]-xGrid[interval-1].

`xlocal` – an input `double` array containing the Gauss-Legendre points translated and normalized to the interval `[xGrid[interval-1], xGrid[interval]]`.

`qw` – an input `double` array containing the Gauss-Legendre weights.

`u` – an input `double` array of dimension `12 by xlocal.length` containing the basis function values that define $\tilde{\beta}(x)$ at the Gauss-Legendre points `xlocal`. Setting

$$u_{k,i} := \text{u[k][i]} \quad \text{and} \quad x_i := \text{xlocal[i]},$$

vector $\tilde{\beta}(x_i)$ is defined as

$$\tilde{\beta}(x_i) := (\beta_{3*(\text{interval}-1)}(x_i), \ldots, \beta_{3*\text{interval}+2}(x_i))^T = (u_{0,i}, u_{1,i}, u_{2,i}, u_{6,i}, u_{7,i}, u_{8,i})^T.$$

`phi` – an output `double` array of length 6 containing Gauss-Legendre approximations for the local contributions

$$\phi_t := \int_{\text{xgrid[interval-1]}}^{\text{xgrid[interval]}} \phi(f,x,t)\tilde{\beta}(x)dx,$$

where `t=time` and $\tilde{\beta}(x) := (\beta_{3*(\text{interval}-1)}(x), \ldots, \beta_{3*\text{interval}+2}(x))^T$. Denoting by `degree` the number of Gauss-Legendre points (`xlocal.length`) and setting $x_j :=$ xlocal[j], vector `phi` contains elements

$$\text{phi[i]} = \text{width} * \sum_{j=0}^{\text{degree-1}} \text{qw}[j]\, \tilde{\beta}_i(x_j)\, \phi(f, x_j, t)$$

for `i=0,...,5`.

`dphi` – an output `double` array of dimension `6 by 6` containing a Gauss-Legendre approximation for the Jacobian of the local contributions ϕ_t at time point `t=time`,

$$\frac{\partial \phi_t}{\partial y} = \int_{\text{xgrid[interval-1]}}^{\text{xgrid[interval]}} \frac{\partial \phi(f,x,t)}{\partial f}\, \tilde{\beta}(x)\, \tilde{\beta}^T(x)dx.$$

The approximation to this symmetric matrix is stored row-wise, i.e.

$$\text{dphi[i][j]} = \text{width} * \sum_{k=0}^{\text{degree-1}} \text{qw}[k]\, \tilde{\beta}_i(x_k)\, \tilde{\beta}_j(x_k)\, \frac{\partial \phi}{\partial f}\Big|_{x=\text{xlocal}[k],\, t=\text{time}}$$

for `i,j=0,...,5`.

FeynmanKac.ToleranceTooSmallException class

```
static public class com.imsl.math.FeynmanKac.ToleranceTooSmallException extends
com.imsl.IMSLException
```

Tolerance is too small.

Constructor

FeynmanKac.ToleranceTooSmallException

```
public FeynmanKac.ToleranceTooSmallException(String key, Object[] arguments)
```

Description

Tolerance is too small.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.TooManyIterationsException class

```
static public class com.imsl.math.FeynmanKac.TooManyIterationsException extends
com.imsl.IMSLException
```

Too many iterations required by the DAE solver.

Constructor

FeynmanKac.TooManyIterationsException

```
public FeynmanKac.TooManyIterationsException(String key, Object[] arguments)
```

Description

Too many iterations required by the DAE solver.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.ErrorTestException class

```
static public class com.imsl.math.FeynmanKac.ErrorTestException extends
com.imsl.IMSLException
```

Error test failure detected.

Constructor

FeynmanKac.ErrorTestException

```
public FeynmanKac.ErrorTestException(String key, Object[] arguments)
```

Description

Error test failure detected.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.CorrectorConvergenceException class

```
static public class com.imsl.math.FeynmanKac.CorrectorConvergenceException
extends com.imsl.IMSLException
```

Corrector failed to converge.

Constructor

FeynmanKac.CorrectorConvergenceException

```
public FeynmanKac.CorrectorConvergenceException(String key, Object[] arguments)
```

Description

Corrector failed to converge.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.IterationMatrixSingularException class

```
static public class com.imsl.math.FeynmanKac.IterationMatrixSingularException
extends com.imsl.IMSLException
```

Iteration matrix is singular.

Constructor

FeynmanKac.IterationMatrixSingularException

```
public FeynmanKac.IterationMatrixSingularException(String key, Object[]
arguments)
```

Description

Iteration matrix is singular.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.TimeIntervalTooSmallException class

```
static public class com.imsl.math.FeynmanKac.TimeIntervalTooSmallException
extends com.imsl.IMSLException
```

Distance between starting time point and end point for the integration is too small.

Constructor

FeynmanKac.TimeIntervalTooSmallException

```
public FeynmanKac.TimeIntervalTooSmallException(String key, Object[] arguments)
```

Description

Distance between starting time point and end point for the integration is too small.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.TcurrentTstopInconsistentException class

```
static public class com.imsl.math.FeynmanKac.TcurrentTstopInconsistentException
extends com.imsl.IMSLException
```

The end value for the integration in time, *tout*, is not consistent with the current time value, *t*.

Constructor

FeynmanKac.TcurrentTstopInconsistentException

```
public FeynmanKac.TcurrentTstopInconsistentException(String key, Object[]
arguments)
```

Description

The end value for the integration in time, *tout*, is not consistent with the current time value, *t*.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.TEqualsToutException class

```
static public class com.imsl.math.FeynmanKac.TEqualsToutException extends
com.imsl.IMSLException
```

The current integration point in time and the end point are equal.

Constructor

FeynmanKac.TEqualsToutException

```
public FeynmanKac.TEqualsToutException(String key, Object[] arguments)
```

Description

The current integration point in time and the end point are equal.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.InitialConstraintsException class

```
static public class com.imsl.math.FeynmanKac.InitialConstraintsException
extends com.imsl.IMSLException
```

The constraints at the initial point are inconsistent.

Constructor

FeynmanKac.InitialConstraintsException

```
public FeynmanKac.InitialConstraintsException(String key, Object[] arguments)
```

Description

The constraints at the initial point are inconsistent.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.ConstraintsInconsistentException class

```
static public class com.imsl.math.FeynmanKac.ConstraintsInconsistentException
extends com.imsl.IMSLException
```

The constraints are inconsistent.

Constructor

FeynmanKac.ConstraintsInconsistentException

```
public FeynmanKac.ConstraintsInconsistentException(String key, Object[]
arguments)
```

Description

The constraints are inconsistent.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

FeynmanKac.BoundaryInconsistentException class

```
static public class com.imsl.math.FeynmanKac.BoundaryInconsistentException
extends com.imsl.IMSLException
```

The boundary conditions are inconsistent.

Constructor

FeynmanKac.BoundaryInconsistentException

```
public FeynmanKac.BoundaryInconsistentException(String key, Object[] arguments)
```

Description

The boundary conditions are inconsistent.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

Chapter 7: Transforms

Types

class FFT 318
class ComplexFFT 322

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.

FFT class

```
public class com.imsl.math.FFT implements Serializable, Cloneable
```

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 δ seconds starting at time t_0. That is, we have

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

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 $f(t_0) = f(t_0 + n\Delta)$. Hence, the period of the function is assumed to be $T = n\Delta$. We can invert the above transform for p as follows:

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

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

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

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

then we have

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

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

$$P_0 := |q_0|$$

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

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

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

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

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

If n is odd,

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

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.

Constructor

FFT

```
public FFT(int n)
```

Description

Constructs an FFT object.

Parameter

`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.

Parameter

`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.

Parameter

`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
```

ComplexFFT class

`public class com.imsl.math.ComplexFFT implements Serializable, Cloneable`

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:

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

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

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

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:

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

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.

Constructor

ComplexFFT

```
public ComplexFFT(int n)
```

Description

Constructs a complex FFT object.

Parameter

`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.

Parameter

`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.

Parameter

`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 8: Nonlinear Equations

Types

class ZeroPolynomial 328
class ZerosFunction 333
class ZeroSystem 339

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:

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

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

ZeroPolynomial class

```
public class com.imsl.math.ZeroPolynomial implements Serializable, Cloneable
```

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.

All the software contained in this library is protected by copyright Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN NO EVENT, NEITHER THE AUTHORS, NOR THE PUBLISHER, NOR ANY MEMBER OF THE EDITORIAL BOARD OF THE JOURNAL "NUMERICAL ALGORITHMS", NOR ITS EDITOR-IN-CHIEF, BE LIABLE FOR ANY ERROR IN THE SOFTWARE, ANY MISUSE OF IT OR ANY DAMAGE ARISING OUT OF ITS USE. THE ENTIRE RISK OF USING THE SOFTWARE LIES WITH THE PARTY DOING SO. ANY USE OF THE SOFTWARE CONSTITUTES ACCEPTANCE OF THE TERMS OF THE ABOVE STATEMENT.

Field

EPSILON_SMALL

```
static final public double EPSILON_SMALL
```

The smallest relative spacing for doubles.

Constructor

ZeroPolynomial

```
public ZeroPolynomial()
```

Description

Creates an instance of the solver.

Methods

computeRoots

```
public Complex[] computeRoots(Complex[] coef) throws
ZeroPolynomial.DidNotConvergeException
```

Description

Computes the roots of the polynomial with Complex coefficients.

$$p(x) = \text{coef}[n] \times x^n + \text{coef}[n-1] \times x^{n-1} + \ldots + \text{coef}[0]$$

Parameter

`coef` – a `Complex` array containing the polynomial coefficients.

Returns

a `Complex` array containing the roots of the polynomial.

computeRoots

```
public Complex[] computeRoots(double[] coef) throws
ZeroPolynomial.DidNotConvergeException
```

Description

Computes the roots of the polynomial with real coefficients.

$$p(x) = \text{coef}[n] \times x^n + \text{coef}[n-1] \times x^{n-1} + \ldots + \text{coef}[0]$$

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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 void setMaxIterations(int maxIterations)`

Description

Sets the maximum number of iterations allowed. The default value is 30.

Parameter

`maxIterations` – an `int` which specifies the maximum number of iterations allowed

Exception

`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
```

ZeroPolynomial.DidNotConvergeException class

`static public class com.imsl.math.ZeroPolynomial.DidNotConvergeException`
`extends com.imsl.IMSLException`

The iteration did not converge

Constructors

ZeroPolynomial.DidNotConvergeException

`public ZeroPolynomial.DidNotConvergeException(String message)`

ZeroPolynomial.DidNotConvergeException

`public ZeroPolynomial.DidNotConvergeException(String key, Object[] arguments)`

ZerosFunction class

`public class com.imsl.math.ZerosFunction implements Serializable, Cloneable`

Finds the real zeros of a real, continuous, univariate function, *f(x)*.

`ZerosFunction` computes *n* real zeros of a real, continuous, univariate function *f*. The search for the zeros of the function can be limited to a specified interval, or extended over the entire real line. The algorithm is generally more efficient if an interval is specified. The user supplied function, *f(x)*, must return valid results for all values in the specified interval. If no interval is given, the user-supplied function must return valid results for all real numbers.

The function has two convergence criteria. The first criterion accepts a root, x, if

$$|f(x)| \leq \tau$$

where τ = `error`, see method `setError`.

The second criterion accepts a root if it is known to be inside of an interval of length at most `errorAbsolute`, see method `setAbsoluteError`.

A root is accepted if it satisfies either criteria and is not within `minSeparation` of another accepted root, see method `setMinimumSeparation`.

If initial guesses for the roots are given, Müller's method (Müller 1956) is used for each of these guesses. For each guess, the Müller iteration is stopped if the next step would be outside of the bound, if given. The iteration is also stopped if the algorithm cannot make further progress in finding a root.

If no guesses for the zeros were given, or if Müller's method with the guesses did not find the requested number of roots, a meta-algorithm, combining Müller's and Brent's methods, is used. Müller's method is used primarily to find the roots of functions, such as $f(x) = x^2$, where the function does not cross the y=0 line. Brent's method is used to find other types of roots.

The meta-algorithm successively refines the interval using a one-dimensional Faure low-discrepancy sequence. The Faure sequence may be scaled by setting the bound interval [a,b] using the `setBounds` method. The Faure sequence will be scaled from (0,1) to (a,b).

If no bound on the function's domain is given, the entire real line must be searched for roots. In this case the Faure sequence is scaled from (0, 1) to $(-\infty, +\infty)$ using the mapping

$$h(u) = \text{xScale} \cdot \tan(\pi(u - 1/2))$$

where `xScale` is set by the `setXScale` method.

At each step of the iteration, the next point in the Faure sequence is added to the list of breakpoints defining the subintervals. Call the points $x_0 = a, x_1 = b, x_2, x_3, \ldots$. The new point, x_s splits an existing subinterval, $[x_p, x_q]$.

The function is evaluated at x_s. If its value is small enough, specifically if

$$|f(x_s)| < \text{mullerTolerance}$$

then Müller's method is used with x_p, x_q and x_s as starting values. If a root is found, it is added to the list of roots. If more roots are required, the new Faure point is used.

If Müller's method did not find a root using the new point, the function value at the point is compared with the function values at the endpoints of the subinterval it divides. If $f(x_p)f(x_s) < 0$ and no root has previously been found in $[x_p, x_s]$, then Brent's method is used to find a root in this interval. Similarly, if the function changes sign over the interval $[x_s, x_q]$, and a root has not already been found in the subinterval, Brent's method is used.

Constructor

ZerosFunction

`public ZerosFunction()`

Description

Creates an instance of the solver.

Methods

allConverged

`public boolean allConverged()`

Description

Returns true if the iterations for all of the roots have converged.

computeZeros

`public double[] computeZeros(ZerosFunction.Function objectF)`

Description

Returns the zeros of a univariate function.

Parameter

`objectF` – contains the function for which the zeros will be found

Returns

a `double` array containing the zerso of the univariate function.

getMaxEvaluations

`public int getMaxEvaluations()`

Description

Returns the maximum number of function evaluations allowed.

Returns

an `int` containing the maximum number of function evaluations allowed

---

getNumberOfEvaluations

```
public int getNumberOfEvaluations()
```

Description

Returns the actual number of function evaluations performed.

Returns

an `int` containing the actual number of function evaluations performed

---

getNumberOfRoots

```
public int getNumberOfRoots()
```

Description

Returns the requested number of roots to be found.

Returns

an `int` containing the requested number of roots to be found

---

getNumberOfRootsFound

```
public int getNumberOfRootsFound()
```

Description

Returns the number of zeros found.

Returns

an `int` containing the number of roots found

---

setAbsoluteError

```
public void setAbsoluteError(double errorAbsolute)
```

Description

Sets the second convergence criterion.

The second criterion accepts a root if the root is known to be inside an interval of length at most `errorAbsolute`.

Parameter

> `errorAbsolute` – a `double` value specifying the second convergence criterion. A root is accepted if the absolute value of the function at the point is less than or equal to `errorAbsolute`. `errorAbsolute` must be greater than or equal to 0.0. Default: `errorAbsolute` = 2.22e-14

---

setBounds

```
public void setBounds(double lowerBound, double upperBound)
```

Description

Sets the closed interval in which to search for the roots. The function must be defined for all values in this interval.

Parameters

`lowerBound` – a `double` containing the lower interval bound. `lowerBound` cannot be greater than or equal to `upperBound`.

`upperBound` – a `double` containing the upper interval bound.

By default the search for the roots is not bounded.

setError

`public void setError(double error)`

Description

Sets the first convergence criterion.

A root is accepted if it is bracketed within an interval of length `error`.

$$|f(x)| \leq \tau$$

where τ = `error`.

Parameter

`error` – a `double` containing the first convergence criterion. `error` must be greater than or equal to 0.0. By default, `error` = 2.0e-8/`xScale`.

setGuess

`public void setGuess(double[] guess)`

Description

Sets the initial guess for the zeros.

Parameter

`guess` – a `double` array containing the initial guesses for the number of zeros to be found. If a bound on the zeros is also given, the guesses must satisfy the bound condition.

setMaxEvaluations

`public void setMaxEvaluations(int maxEvaluations)`

Description

Sets the maximum number of function evaluations allowed.

Methods `allConverged` and `getNumberOfRootsFound` can be used to confirm whether or not the number of roots requested were found within the maximum evaluations specified. `maxEvaluations` must be greater than or equal to 0.0.

Parameter

`maxEvaluations` – an `int` containing the maximum number of function evaluations allowed. Once this limit is reached, the roots found are returned.

setMinimumSeparation

`public void setMinimumSeparation(double minSeparation)`

Description

Sets the minimum separation between accepted roots.

Parameter

`minSeparation` – a `double` containing the minimum separation between accepted roots. If two points satisfy the convergence criteria, but are within `minSeparation` of each other, only one of the roots is accepted. `minSeparation` must be greater than or equal to 0.0.

By default, `minSeparation` = 1.0e-8/`xScale`.

setMullerTolerance

```
public void setMullerTolerance(double mullerTolerance)
```

Description

Sets the tolerance used during refinement to determine if Müllers method is started.

Müller's method is started if, during refinement, a point is found for which the absolute value of the function is less than `mullerTolerance` and the point is not near an already discovered root. If `mullerTolerance` is less than or equal to zero, Müller's method is never used.

Parameter

`mullerTolerance` – a `double` containing the tolerance used during refinement to determine when the Müller's method is used. By default, `mullerTolerance` = 1.0e-8/`errorAbsolute`

setNumberOfRoots

```
public void setNumberOfRoots(int numRoots)
```

Description

Sets the number of roots to be found.

Parameter

`numRoots` – an `int` containing the number of roots to be found. `numRoots` must be greater than or equal to zero. By default, `numRoots`=1.

setXScale

```
public void setXScale(double xScale)
```

Description

Sets the the scaling in the x-coordinate.

If no bound on the function's domain is given, the entire real line must be searched for roots. In this case the Faure sequence is scaled from (0, 1) to $(-\infty, \infty)$ using the mapping

$$h(u) = \text{xScale} \cdot \tan\left(\pi(u - 1/2)\right)$$

.

Parameter

`xScale` – a `double` containing the scaling in the x-coordinate. The absolute value of the roots divided by `xScale` should be about one. `xScale` must be greater than 0.0. By default, `xScale`=1.0.

Example: Zeros of a Univariate Function

In this example 3 zeros of the sin function are found.

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

public class ZerosFunctionEx1 {

    public static void main(String args[]) {

        ZerosFunction.Function fcn = new ZerosFunction.Function() {
            public double f(double x) {
                return Math.sin(x);
            }
        };

        ZerosFunction zf = new ZerosFunction();
        double guess[] = {5, 18, -6};
        zf.setGuess(guess);
        double zeros[] = zf.computeZeros(fcn);
        for (int k = 0; k < zeros.length; k++) {
            System.out.println(zeros[k] + " = " + (zeros[k] / Math.PI) + " pi");
        }
    }
}
```

Output

```
-2.953824405025999E-22 = -9.40231510170729E-23 pi
3.1415926535897936 = 1.0000000000000002 pi
6.283185307179586 = 2.0 pi
```

ZerosFunction.Function interface

`public interface com.imsl.math.ZerosFunction.Function`

Public interface for the user supplied function to `ZerosFunction`.

The user supplied function, *f(x)*, must return valid results for all values in the specified interval. If no interval is given, the user-supplied function must return valid results for all real numbers.

Method

f

`public double f(double x)`

Description

Returns the value of the function at the given point.

Parameter

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

Returns

a `double` containing the value of the function at x

ZeroSystem class

```
public class com.imsl.math.ZeroSystem implements Serializable, Cloneable
```

Solves a system of n nonlinear equations f(x) = 0 using a modified Powell hybrid algorithm.

`ZeroSystem` is based on the MINPACK subroutine `HYBRDJ` , 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`.

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	Iteration increments are printed.
FINE	Prints convergnce tests.
FINER	Intermediate solution values are provided.
FINEST	Tracks progress through internal methods.

Constructor

ZeroSystem

```
public ZeroSystem(int n)
```

Description

Creates an object to find the zeros of a system of n equations.

Parameter

`n` – an `int` indicating the number of equations to be solved and the number of unknowns.

Methods

getLogger

```
public Logger getLogger()
```

Description

Returns the logger object.

Returns

a `java.util.logging.Logger` object, if present, or `null`.

setGuess

```
public void setGuess(double[] xguess)
```

Description

Sets the initial estimate of the root. The default is to set `xguess` to all zeros.

Parameter

`xguess` – a `double` array containing the initial guess. The length of `xguess` must be equal to the number of equations.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Sets the maximum number of iterations allowed. The default value is 200.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed

Exception

`IllegalArgumentException` is thrown if maxIterations is less than or equal to zero.

setRelativeError

```
public 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.

Parameter

`errorRelative` – a `double` specifying the relative error tolerance

Exception

`IllegalArgumentException` is thrown if `errorRelative` is less than 0 or greater than 1.

solve

`public double[] solve(ZeroSystem.Function objectF) throws ZeroSystem.TooManyIterationsException, ZeroSystem.ToleranceTooSmallException, ZeroSystem.DidNotConvergeException`

Description

Solve a system of nonlinear equations using the modified Powell hybrid algorithm

Parameter

`objectF` – a `ZeroSystem.Function` that 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

Exceptions

`TooManyIterationsException` is thrown if the maximum number of iterations is exceeded

`ToleranceTooSmallException` is thrown if the error tolerance is too small

`DidNotConvergeException` is thrown if the algorithm does not converge

Example 1: 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
```

Example 2: Solve a System of Nonlinear Equations with Logging

A system of nonlinear equations is solved with reduced accuracy and logging enabled.

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

public class ZeroSystemEx2 {

    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] = 0.5 * x[0] + x[1] + 0.5 * x[2] - x[5] / x[6];
                f[1] = x[2] + x[3] + 2 * x[4] - 2.0 / x[6];
                f[2] = x[0] + x[1] + x[4] - 1 / x[6];
                f[3] = -28837 * x[0] - 139009 * x[1] - 78213 * x[2] + 18927
                        * x[3] + 8427 * x[4] + 13492 / x[6]
                        - 10690 * x[5] / x[6];
                f[4] = x[0] + x[1] + x[2] + x[3] + x[4] - 1;
                f[5] = 400 * x[0] * x[3] * x[3] * x[3] - 178370.0 * x[2] * x[4];
                f[6] = x[0] * x[2] - 2.6058 * x[1] * x[3];
            }
        };

        ZeroSystem zf = new ZeroSystem(7);
        zf.setRelativeError(1e-2);      // reduced accuracy
        double guess[] = {0.5, 0.0, 0.0, 0.5, 0.0, 0.5, 2.0};
        zf.setGuess(guess);

        Logger logger = zf.getLogger();
        ConsoleHandler ch = new ConsoleHandler();
        ch.setLevel(Level.ALL);       // default ConsoleHandler Level is INFO
        logger.setLevel(Level.FINER);
        logger.addHandler(ch);
        ch.setFormatter(new com.imsl.IMSLFormatter());

        new PrintMatrix("zeros").print(zf.solve(fcn));
    }
}
```

Output

```
Iteration 1
Current Solution

0  0.5
1  0
2  0
3  0.5
4  0
5  0.5
6  2


Convergence if 0.2143059349437466 <= 0.02179449471770337
Convergence if 881.8545798486278 == 0.0
Convergence if 0.4286118698874933 <= 0.023714058983732928
Convergence if 62.20236343524046 == 0.0
Iteration 2
    Current Solution

0   0.4519416338243868
1   0.00047570035964456
2   0.00811462449691682
3   0.5413220960737729
4  -0.00707517903874563
5   0.5209316770542504
6   2.20336087353378


Convergence if 0.21430593494374664 <= 0.023714058983732928
Convergence if 62.20236343524046 == 0.0
Convergence if 0.21430593494374664 <= 0.025668928384900824
Convergence if 48.631150262832776 == 0.0
Iteration 3
    Current Solution

0   0.4103556836647792
1   0.00345389046504245
2   0.0205011507011974
3   0.5643329875514621
4  -0.00193459468156145
5   0.5357476651118313
6   2.4113510334391686


Convergence if 0.10715296747187332 <= 0.025668928384900824
Convergence if 48.631150262832776 == 0.0
Convergence if 0.21430593494374667 <= 0.026655240629937994
Convergence if 24.594369376894957 == 0.0
Iteration 4
    Current Solution

0  0.38976301357761567
1  0.00475347250310963
2  0.02636818944534906
```

```
3  0.5751249696426907
4  0.00124115547252234
5  0.5418663859482151
6  2.5155508333834984


Convergence if 0.10715296747187333 <= 0.026655240629937994
Convergence if 24.594369376894957 == 0.0
Convergence if 0.2143059349437467 <= 0.027655165558489504
Convergence if 11.23333567341594 == 0.0
Iteration 5
    Current Solution

0  0.37141787401480864
1  0.00596373459917273
2  0.03176913111146456
3  0.5853405218780866
4  0.00345206419996186
5  0.5483732628042493
6  2.620255151899655


Convergence if 0.10715296747187335 <= 0.027655165558489504
Convergence if 11.23333567341594 == 0.0
Convergence if 0.21430593494374675 <= 0.028667286402919635
Convergence if 4.273723534011024 == 0.0
Iteration 6
    Current Solution

0  0.35522919183046353
1  0.00699109092408178
2  0.03660354056250206
3  0.5951581971633527
4  0.00460693872783803
5  0.5557202116250662
6  2.725343495169716


Convergence if 0.10715296747187338 <= 0.028667286402919635
Convergence if 4.273723534011024 == 0.0
Convergence if 0.05357648373593669 <= 0.028667286402919635
Convergence if 4.273723534011024 == 0.0
Iteration 6
    Current Solution

0  0.35522919183046353
1  0.00699109092408178
2  0.03660354056250206
3  0.5951581971633527
4  0.00460693872783803
5  0.5557202116250662
6  2.725343495169716


Convergence if 0.10715296747187338 <= 0.02917820821842807
Convergence if 1.3397448169540218 == 0.0
```

```
Iteration 7
    Current Solution

0  0.3478531575077773
1  0.00752847153174577
2  0.03866774311942223
3  0.6004623965959569
4  0.00437003290309475
5  0.5601127012361977
6  2.777917247138637


Convergence if 0.05357648373593669 <= 0.02917820821842807
Convergence if 1.3397448169540218 == 0.0
Convergence if 0.05357648373593669 <= 0.02969087049889735
Convergence if 1.2190150551287604 == 0.0
Iteration 8
    Current Solution

0  0.3410446926772013
1  0.0080070341885894
2  0.04065623471212016
3  0.6052985736893154
4  0.00418754444673453
5  0.564491500544133
6  2.830617394205158


Convergence if 0.026788241867968344 <= 0.02969087049889735
Convergence if 1.2190150551287604 == 0.0
  zeros
     0
0  0.341
1  0.008
2  0.041
3  0.605
4  0.004
5  0.564
6  2.831
```

ZeroSystem.DidNotConvergeException class

`static public class com.imsl.math.ZeroSystem.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge.

Constructors

ZeroSystem.DidNotConvergeException

```
public ZeroSystem.DidNotConvergeException(String message)
```

ZeroSystem.DidNotConvergeException

```
public ZeroSystem.DidNotConvergeException(String key, Object[] arguments)
```

ZeroSystem.Function interface

```
public interface com.imsl.math.ZeroSystem.Function
```

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

Method

f

```
public void f(double[] x, double[] f)
```

Description

Returns the value of the function at the given point.

Parameters

`x` – a `double` array of length `n` which contains the point at which the functions are to be evaluated. The contents of this array must not be altered by this function.

`f` – a `double` array of length `n` which contains the value of the function at `x` .

ZeroSystem.Jacobian interface

```
public interface com.imsl.math.ZeroSystem.Jacobian implements
com.imsl.math.ZeroSystem.Function
```

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

Method

jacobian

```
public void jacobian(double[] x, double[][] jac)
```

Description

Returns the value of the Jacobian at the given point.

Parameters

x – a double array of length n 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 n by n 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].

ZeroSystem.ToleranceTooSmallException class

```
static public class com.imsl.math.ZeroSystem.ToleranceTooSmallException extends
com.imsl.IMSLException
```

Tolerance too small

Constructor

ZeroSystem.ToleranceTooSmallException

```
public ZeroSystem.ToleranceTooSmallException(String key, Object[] arguments)
```

ZeroSystem.TooManyIterationsException class

```
static public class com.imsl.math.ZeroSystem.TooManyIterationsException extends
com.imsl.IMSLException
```

Too many iterations.

Constructors

ZeroSystem.TooManyIterationsException

```
public ZeroSystem.TooManyIterationsException()
```

ZeroSystem.TooManyIterationsException

```
public ZeroSystem.TooManyIterationsException(Object[] arguments)
```

ZeroSystem.TooManyIterationsException

```
public ZeroSystem.TooManyIterationsException(String key, Object[] arguments)
```

Chapter 9: Optimization

Types

class MinUncon 351
class MinUnconMultiVar 357
class NonlinLeastSquares 369
class SparseLP 380
class DenseLP 406
class QuadraticProgramming 421
class MinConGenLin 429
class BoundedLeastSquares 441
class BoundedVariableLeastSquares 451
class NonNegativeLeastSquares 456
class MinConNLP 461
class NumericalDerivatives 490

Usage Notes

Unconstrained Minimization

The unconstrained minimization problem can be stated as follows:

$$\min_{x \in R^n} f(x)$$

where $f : \mathbf{R}^n \rightarrow \mathbf{R}$ 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 `NumericalDerivatives` class can also be used in computing the gradient for `MinUnconMultiVar` as well as other classes. For an example, see `NumericalDerivatives` Example 6.

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 `SparseLP` uses an infeasible primal-dual interior-point method to solve sparse linear programming problems of all sizes. The constraint matrix is stored in sparse coordinate storage format.

The class `DenseLP` uses an active set strategy 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 $f : \mathbf{R}^n \rightarrow \mathbf{R}$ and $g_i : \mathbf{R}^n \rightarrow \mathbf{R}$, 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.

Return Values from User-Supplied Functions

All values returned by user-supplied functions must be valid real numbers. It is the user's responsibility to check that the values returned by a user-supplied function do not contain NaN, infinity, or negative infinity values.

Example: Minimum of a smooth function

The minimum of $e^x - 5x$ is found using function evaluations only.

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

public class OptimizationIntroEx1 {
    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) {
                double y = Math.exp(x) - 5.*x;
                if(!Double.isNaN(y)) {
                       return y;
                } else {
                       return 0.0;
                }
            }
        };
        System.out.println("Minimum is " + zf.computeMin(fcn));
    }
}
```

MinUncon class

`public class com.imsl.math.MinUncon implements Serializable, Cloneable`

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 ε, 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 ε 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 ε 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 ε 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 δ 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 $f_n \ge f_c$ 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 $\varepsilon_c = \max\{1.0, |x_c|\}\varepsilon$, ε is a relative error tolerance and ε_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.

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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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 1.6094175999200164
```

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
```

MinUncon.Function interface

`public interface com.imsl.math.MinUncon.Function`

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

Method

f

`public double f(double x)`

Description

Public interface for the smooth function of a single variable to be minimized.

Parameter

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

Returns

a `double`, the value of the function at x

MinUncon.Derivative interface

```
public interface com.imsl.math.MinUncon.Derivative implements
com.imsl.math.MinUncon.Function
```

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

Method

g

```
public double g(double x)
```

Description

Public interface for the smooth function of a single variable to be minimized.

Parameter

`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

MinUnconMultiVar class

```
public class com.imsl.math.MinUnconMultiVar implements Serializable, Cloneable
```

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

$$f(x_n) \le f(x_c) + \alpha g^T d, \;\; \alpha \in (0, 0.5)$$

Finally, the optimality condition $||g(x)|| \le \varepsilon$ where ε is a gradient tolerance, is sought.

When optimality is not achieved, B is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

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 the gradient for a more accurate gradient evaluation (`setGradient`).

The user may also choose to supply the Hessian. If, during intermediate calculations, the user supplied Hessian becomes non-symmetric positive definite, then the algorithm will resort to using the default approximate Hessian.

Constructor

MinUnconMultiVar

`public MinUnconMultiVar(int n)`

Description

Unconstrained minimum constructor for a function of n variables of type `double`.

Parameter

`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
MinUnconMultiVar.FalseConvergenceException,
MinUnconMultiVar.MaxIterationsException,
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.

Parameter

`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. If F implements `Hessian` then the user-supplied gradient and Hessian are used. Otherwise, an approximate Hessian is used during computation.

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 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.

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the number of `java.lang.Thread` instances used for parallel processing.

Returns

an `int` containing the number of `java.lang.Thread` instances used for parallel processing.

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.

Parameter

`fdigit` – a `double` scalar value specifying the number of good digits in the user supplied function

Exception

`IllegalArgumentException` is thrown if `fdigit` 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.

Parameter

`fscale` – a `double` scalar specifying the function scaling value for scaling the gradient

Exception

`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.

Parameter

`gradientTolerance` – a `double` specifying the gradient tolerance used to compute the gradient

Exception

`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..

Parameter

`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

Parameter

`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.

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.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed

Exception

`IllegalArgumentException` is thrown if maxIterations 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.

Parameter

`maximumStepsize` – a nonnegative `double` value specifying the maximum allowable stepsize

Exception

`IllegalArgumentException` is thrown if maximumStepsize is less than or equal to 0

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing. If `numberOfThreads` is greater than 1, then interface `Function.f` is evaluated in parallel and `Function.f` must be thread-safe. Otherwise, unexpected behavior can occur.

Default: `numberOfThreads` = 1.

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.

Parameter

`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.

Exception

`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..

Parameter

`xscale` – a `double` array specifying the diagonal scaling matrix for the variables

Exception

`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)
```

Example 3: 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 Hessian.

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

public class MinUnconMultiVarEx3 {

    static class MyFunction implements MinUnconMultiVar.Hessian {

        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 void hessian(double[] x, double[][] hess) {
            hess[0][0] = -4.e2 * x[1] + 1.2e3 * x[0] * x[0] + 2.e0;
            hess[1][0] = -4.e2 * x[0];
            hess[0][1] = hess[1][0];
            hess[1][1] = 2.e2;
        }
    }

    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 (1.0000000139452756, 1.0000000143547352)
```

MinUnconMultiVar.Function interface

```
public interface com.imsl.math.MinUnconMultiVar.Function
```

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

Method

f

```
public double f(double[] x)
```

Description

Public interface for the multivariate function to be minimized.

Parameter

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

Returns

a `double`, the value of the function at x

MinUnconMultiVar.Gradient interface

```
public interface com.imsl.math.MinUnconMultiVar.Gradient implements
com.imsl.math.MinUnconMultiVar.Function
```

Public interface for the user supplied gradient to the `MinUnconMultiVar` object.

Method

gradient

```
public 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

MinUnconMultiVar.Hessian interface

```
public interface com.imsl.math.MinUnconMultiVar.Hessian implements
com.imsl.math.MinUnconMultiVar.Gradient
```

Public interface for the user supplied Hessian to the `MinUnconMultiVar` object.

Method

hessian

```
public void hessian(double[] x, double[][] hess)
```

Description

Public interface for the Hessian of the multivariate function to be minimized.

Parameters

`x` – a `double` array, the point at which the Hessian of the function is to be evaluated

`hess` – a `double` matrix, the value of the Hessian of the function at x

MinUnconMultiVar.ApproximateMinimumException class

```
static public class com.imsl.math.MinUnconMultiVar.ApproximateMinimumException
extends com.imsl.IMSLException
```

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.

Constructors

MinUnconMultiVar.ApproximateMinimumException

`public MinUnconMultiVar.ApproximateMinimumException(String message)`

Description

Constructs a `ApproximateMinimumException` object.

Parameter

`message` – a `String` containing the error message

MinUnconMultiVar.ApproximateMinimumException

`public MinUnconMultiVar.ApproximateMinimumException(String key, Object[] arguments)`

Description

Constructs a `ApproximateMinimumException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinUnconMultiVar.FalseConvergenceException class

```
static public class com.imsl.math.MinUnconMultiVar.FalseConvergenceException
extends com.imsl.IMSLException
```

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.

Constructors

MinUnconMultiVar.FalseConvergenceException

`public MinUnconMultiVar.FalseConvergenceException(String message)`

Description

Constructs a `FalseConvergenceException` object.

Parameter

`message` – a `String` containing the error message

MinUnconMultiVar.FalseConvergenceException

```
public MinUnconMultiVar.FalseConvergenceException(String key, Object[]
arguments)
```

Description

Constructs a `FalseConvergenceException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinUnconMultiVar.MaxIterationsException class

```
static public class com.imsl.math.MinUnconMultiVar.MaxIterationsException
extends com.imsl.IMSLException
```

Maximum number of iterations exceeded.

Constructors

MinUnconMultiVar.MaxIterationsException

```
public MinUnconMultiVar.MaxIterationsException(String message)
```

Description

Constructs a `MaxIterationsException` object.

Parameter

`message` – a `String` containing the error message

MinUnconMultiVar.MaxIterationsException

```
public MinUnconMultiVar.MaxIterationsException(String key, Object[] arguments)
```

Description

Constructs a `MaxIterationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinUnconMultiVar.UnboundedBelowException class

`static public class com.imsl.math.MinUnconMultiVar.UnboundedBelowException extends com.imsl.IMSLException`

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.

Constructors

MinUnconMultiVar.UnboundedBelowException

`public MinUnconMultiVar.UnboundedBelowException(String message)`

Description

Constructs a `UnboundedBelowException` object.

Parameter

`message` – a `String` containing the error message

MinUnconMultiVar.UnboundedBelowException

`public MinUnconMultiVar.UnboundedBelowException(String key, Object[] arguments)`

Description

Constructs a `UnboundedBelowException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

NonlinLeastSquares class

`public class com.imsl.math.NonlinLeastSquares implements Serializable, Cloneable`

Nonlinear least squares.

`NonlinLeastSquares` is based on the MINPACK routine `LMDIF` by Moré 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. The first stopping criteria occurs when the norm of the function is less than the value set in method `setAbsoluteTolerance`. The second stopping criteria occurs when the norm of the scaled gradient is less than the value set in method `setGradientTolerance`. The third stopping criteria occurs when the scaled distance between the last two steps is less than the value set by method `setStepTolerance`. 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`.

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	*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 maximum stepsize is too small.

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the number of `java.lang.Thread` instances used for parallel processing.

Returns

an `int` containing the number of `java.lang.Thread` instances used for parallel processing.

setAbsoluteTolerance

```
public void setAbsoluteTolerance(double absoluteTolerance)
```

Description

Set the absolute function tolerance.

Parameter

`absoluteTolerance` – a `double` scalar value specifying the absolute function tolerance. When the norm of the function is less than `absoluteTolerance` the procedure stops.

Default: `absoluteTolerance` = 1.0e-32

Exception

`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.

Parameter

`ngood` – an `int` specifying the number of good digits in the user supplied function which defines the least-squares problem

Exception

`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.

Parameter

`falseConvergenceTolerance` – a double scalar value specifying the false convergence tolerance

Exception

`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.

Parameter

`fscale` – a `double` array specifying the diagonal scaling matrix for the functions. The *i*-th component of `fscale` is a positive scalar specifying the reciprocal magnitude of the *i*-th component function of the problem. By default, the identity is used.

Exception

`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 scaled gradient tolerance stopping critierion.

Parameter

`gradientTolerance` – a `double` specifying the scaled gradient tolerance used in stopping criteria. The *i*-th component of the scaled gradient at `x` is calculated as

$$\frac{|g_i| * \max(|x_i|, \frac{1}{s_i})}{\frac{1}{2}||F(x)||_2^2}$$

where

$$g_i = \left(J(x)^T F(x)\right)_i * (f_s)_i^2 ,$$

J(x) is the Jacobian, s = `xscale`, f_s= `fscale`, and

$$||F(x)||_2^2 = \sum_{i=1}^{m} f_i(x)^2 .$$

When the norm of the scaled gradient is less than `gradientTolerance` the procedure stops. By default, `gradientTolerance` = $\sqrt[3]{\varepsilon}$ where ε is machine precision.

Exception

`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.

Parameter

`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.

Parameter

`initialTrustRegion` – a `double` scalar value specifying the initial trust region radius

Exception

`IllegalArgumentException` is thrown if initialTrustRegion 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.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed

Exception

`IllegalArgumentException` is thrown if maxIterations 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.

Parameter

`maximumStepsize` – a nonnegative `double` value specifying the maximum allowable stepsize.

Exception

`IllegalArgumentException` is thrown if maximumStepsize is less than or equal to 0

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing. If `numberOfThreads` is greater than 1, then interface `Function.f` is evaluated in parallel and `Function.f` must be thread-safe. Otherwise, unexpected behavior can occur.

Default: `numberOfThreads` = 1.

setRelativeTolerance

```
public void setRelativeTolerance(double relativeTolerance)
```

Description

Set the relative function tolerance.

Parameter

`relativeTolerance` – a `double` scalar value specifying the relative function tolerance
Default: `relativeTolerance` = 1.0e-20

Exception

`IllegalArgumentException` is thrown if `relativeTolerance` is less than or equal to 0

setStepTolerance

```
public void setStepTolerance(double stepTolerance)
```

Description

Set the scaled step tolerance.

Parameter

`stepTolerance` – a `double` scalar value specifying the scaled step tolerance used in stopping criteria. The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, \frac{1}{s_i})}$$

where s = `xscale`. When the scaled distance between the last two steps is less than `stepTolerance` the procedure stops.
By default, `stepTolerance` = $\sqrt[3]{\varepsilon}$ where ε is machine precision.

Exception

`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.

Parameter

`xscale` – a `double` array specifying the diagonal scaling matrix for the variables. `xscale` is used in scaling the gradient and the distance between two points. See methods `setGradientTolerance` and `setStepTolerance` for more detail. By default, the identity is used.

Exception

`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
NonlinLeastSquares.TooManyIterationsException
```

Description

Solve a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm and a Jacobian.

Parameter

`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

Exception

`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.};

        NonlinLeastSquares zs = new NonlinLeastSquares(m, n);
        zs.setGuess(xguess);
        zs.setXscale(xscale);
        zs.setFscale(fscale);
        double[] 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.};

        NonlinLeastSquares zs = new NonlinLeastSquares(m, n);
        zs.setGuess(xguess);
        zs.setXscale(xscale);
        zs.setFscale(fscale);
        double[] 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
```

NonlinLeastSquares.TooManyIterationsException class

`static public class com.imsl.math.NonlinLeastSquares.TooManyIterationsException extends com.imsl.IMSLException`

Too many iterations.

Constructors

NonlinLeastSquares.TooManyIterationsException

`public NonlinLeastSquares.TooManyIterationsException()`

Description

Constructs a `TooManyIterationsException` object.

NonlinLeastSquares.TooManyIterationsException

`public NonlinLeastSquares.TooManyIterationsException(Object[] arguments)`

Description

Constructs a `TooManyIterationsException` object.

Parameter

`arguments` – an `Object` array containing arguments used within the error message string

NonlinLeastSquares.TooManyIterationsException

`public NonlinLeastSquares.TooManyIterationsException(String key, Object[] arguments)`

Description

Constructs a `TooManyIterationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

NonlinLeastSquares.Function interface

`public interface com.imsl.math.NonlinLeastSquares.Function`

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

Method

f

```
public 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.

NonlinLeastSquares.Jacobian interface

```
public interface com.imsl.math.NonlinLeastSquares.Jacobian implements
com.imsl.math.NonlinLeastSquares.Function
```

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

Method

jacobian

```
public 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

SparseLP class

`public class com.imsl.math.SparseLP implements Serializable, Cloneable`

Solves a sparse linear programming problem by an infeasible primal-dual interior-point method.

Class `SparseLP` uses an infeasible primal-dual interior-point method to solve linear programming problems, i.e., problems of the form

$$\min_{x \in R^n} c^T x \quad \text{subject to} \quad \begin{array}{l} b_l \le Ax \le b_u \\ x_l \le x \le x_u \end{array}$$

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.

Internally, `SparseLP` transforms the problem given by the user into a simpler form that is computationally more tractable. After redefining the notation, the new form reads

$$\min c^T x \quad \text{subject to} \quad \begin{array}{rlll} Ax & = b & & \\ x_i + s_i & = u_i, & x_i, s_i \ge 0, & i \in I_u \\ x_j & \ge 0, & & j \in I_s \end{array}$$

Here, $I_u \cup I_s = \{1, \ldots, n\}$ is a partition of the index set $\{1, \ldots, n\}$ into upper bounded and standard variables.

In order to simplify the description it is assumed in the following that the problem above contains only variables with upper bounds, i.e. is of the form

$$(P) \quad \min c^T x \quad \text{subject to} \quad \begin{array}{c} Ax = b \\ x + s = u, \\ x, s \ge 0 \end{array}$$

The corresponding dual problem is

$$(D) \quad \max b^T y - u^T w \quad \text{subject to} \quad \begin{array}{r} A^T y + z - w = c, \\ z, w \ge 0 \end{array}$$

The Karush-Kuhn-Tucker (KKT) optimality conditions for *(P)* and *(D)* are

$$
\begin{aligned}
Ax &= b, & (1.1)\\
x+s &= u, & (1.2)\\
A^T y + z - w &= c, & (1.3)\\
XZe &= 0, & (1.4)\\
SWe &= 0, & (1.5)\\
x,z,s,w &\geq 0, & (1.6)
\end{aligned}
$$

where $X = diag(x), Z = diag(z), S = diag(s), W = diag(w)$ are diagonal matrices and $e = (1,\ldots,1)^T$ is a vector of ones.

Class SparseLP, like all infeasible interior-point methods, generates a sequence

$$(x_k, s_k, y_k, z_k, w_k), \quad k = 0,1,\ldots$$

of iterates that satisfy $(x_k, s_k, y_k, z_k, w_k) > 0$ for all k, but are in general not feasible, i.e. the linear constraints (1.1)-(1.3) are only satisfied in the limiting case $k \to \infty$.

The barrier parameter μ, defined by

$$\mu = \frac{x^T z + s^T w}{2n}$$

measures how good the complementarity conditions (1.4), (1.5) are satisfied.

Mehrotra's predictor-corrector algorithm is a variant of Newton's method applied to the KKT conditions (1.1)-(1.5). Class SparseLP uses a modified version of this algorithm to compute the iterates $(x_k, s_k, y_k, z_k, w_k)$. In every step of the algorithm, the search direction vector

$$\Delta := (\Delta x, \Delta s, \Delta y, \Delta z, \Delta w)$$

is decomposed into two parts, $\Delta = \Delta_a + \Delta_c^{\omega}$, where Δ_a and Δ_c^{ω} denote the affine-scaling and the weighted centering components, respectively. Here,

$$\Delta_c^{\omega} := (\omega_P \Delta x_c, \omega_P \Delta s_c, \omega_D \Delta y_c, \omega_D \Delta z_c, \omega_D \Delta w_c)$$

where ω_P and ω_D denote the primal and dual corrector weights, respectively.

The vectors Δ_a and $\Delta_c := (\Delta x_c, \Delta s_c, \Delta y_c, \Delta z_c, \Delta w_c)$ are determined by solving the linear system

$$
\begin{bmatrix}
A & 0 & 0 & 0 & 0\\
I & 0 & I & 0 & 0\\
0 & A^T & 0 & I & -I\\
Z & 0 & 0 & X & 0\\
0 & 0 & W & 0 & S
\end{bmatrix}
\begin{bmatrix}
\Delta x\\ \Delta y\\ \Delta s\\ \Delta z\\ \Delta w
\end{bmatrix}
=
\begin{bmatrix}
r_b\\ r_u\\ r_c\\ r_{xz}\\ r_{ws}
\end{bmatrix}
\qquad (2)
$$

for two different right-hand sides.

For Δ_a, the right-hand side is defined as

$$(r_b, r_u, r_c, r_{xz}, r_{ws}) = (b - Ax, u - x - s, c - A^T y - z + w, -XZe, -WSe).$$

Here, r_b and r_u are the violations of the primal constraints and r_c defines the violations of the dual constraints.

The resulting direction Δ_a is the pure Newton step applied to the system (1.1)-(1.5).

In order to obtain the corrector direction Δ_c, the maximum stepsizes α_{Pa} in the primal and α_{Da} in the dual space preserving nonnegativity of (x,s) and (z,w) respectively, are determined, and the predicted complementarity gap

$$g_a = (x + \alpha_{Pa}\Delta x_a)^T (z + \alpha_{Da}\Delta z_a) + (s + \alpha_{Pa}\Delta s_a)^T (w + \alpha_{Da}\Delta w_a)$$

is computed. It is then used to determine the barrier parameter

$$\hat{\mu} = \left(\frac{g_a}{g}\right)^2 \frac{g_a}{2n},$$

where $g = x^T z + s^T w$ denotes the current complementarity gap.

The direction Δ_c is then computed by choosing

$$(r_b, r_u, r_c, r_{xz}, r_{ws}) = (0, 0, 0, \hat{\mu}e - \Delta X_a \Delta Z_a e, \hat{\mu}e - \Delta W_a \Delta S_a e)$$

as the right-hand side in the linear system (2).

Class `SparseLP` now uses a line search to find the optimal weight $\hat{\omega} = (\hat{\omega}_P, \hat{\omega}_D)$ that maximizes the stepsizes (α_P, α_D) in the primal and dual directions of $\Delta = \Delta_a + \Delta_c^{\omega}$, respectively.

A new iterate is then computed using a step reduction factor $\alpha_0 = 0.99995$:

$$\begin{aligned}(x_{k+1}, s_{k+1}, y_{k+1}, z_{k+1}, w_{k+1}) \quad = \quad & (x_k, s_k, y_k, z_k, w_k) + \\ & \alpha_0 \left(\alpha_P \Delta x, \alpha_P \Delta s, \alpha_D \Delta y, \alpha_D \Delta z, \alpha_D \Delta w\right)\end{aligned}$$

In addition to the weighted Mehrotra predictor-corrector, `SparseLP` also uses multiple centrality correctors to enlarge the primal-dual stepsizes per iteration step and to reduce the overall number of iterations required to solve an LP problem. The maximum number of centrality corrections depends on the ratio of the factorization and solve efforts for system (2) and is therefore problem dependent. For a detailed description of multiple centrality correctors, refer to Gondzio (1994).

The linear system (2) can be reduced to more compact forms, the augmented system (AS)

$$\begin{bmatrix} -\Theta^{-1} & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r \\ h \end{bmatrix} \qquad (3)$$

or further by elimination of Δx to the normal equations (NE) system

$$A\Theta A^T \Delta y = A\Theta r + h, \quad (4)$$

where

$$\Theta = \left(X^{-1}Z + S^{-1}W\right)^{-1}, r = r_c - X^{-1}r_{xz} + S^{-1}r_{ws} - S^{-1}Wr_u, h = r_b.$$

The matrix on the left-hand side of (3), which is symmetric indefinite, can be transformed into a symmetric quasidefinite matrix by regularization. Since these types of matrices allow for a Cholesky-like factorization, the resulting linear system can be solved easily for $(\Delta x, \Delta y)$ by triangular substitutions. For more information on the regularization technique, see Altman and Gondzio (1998). For the NE system, matrix $A\Theta A^T$ is positive definite, and therefore a sparse Cholesky algorithm can be used to factor $A\Theta A^T$ and solve the system for Δy by triangular substitutions with the Cholesky factor L.

In class `SparseLP`, both approaches are implemented. The AS approach is chosen if A contains dense columns, if there are a considerable number of columns in A that are much denser than the remaining ones, or if there are many more rows than columns in the structural part of A. Otherwise, the NE approach is selected.

Class `SparseLP` stops with optimal termination status if the current iterate satisfies the following three conditions:

$$\frac{\mu}{1 + 0.5\left(|c^T x| + |b^T y - u^T w|\right)} \le \text{optimalityTolerance}$$

$$\frac{\| (b - Ax, x + s - u) \|}{1 + \| (b, u) \|} \le \text{primalTolerance}$$

and

$$\frac{\| c - A^T y - z + w \|}{1 + \| c \|} \le \text{dualTolerance}$$

where `primalTolerance`, `dualTolerance` and `optimalityTolerance` are primal infeasibility, dual infeasibility and optimality tolerances, respectively. The default value is 1.0e-10 for `optimalityTolerance` and 1.0e-8 for the other two tolerances.

Class `SparseLP` is based on the code HOPDM developed by Jacek Gondzio et al., see the HOPDM User's Guide (1995).

Constructors

SparseLP

```
public SparseLP(MPSReader mps)
```

Description

Constructs a `SparseLP` object using an `MPSReader` object.

Parameter

`mps` – an `MPSReader` object specifying the Linear Programming problem

SparseLP

```
public SparseLP(SparseMatrix a, double[] b, double[] c)
```

Description

Constructs a `SparseLP` object.

Parameters

`a` – a `SparseMatrix` object containing the location and value of each nonzero coefficient in the constraint matrix *A*. If there is no constraint matrix, set `a = null`.

`b` – a `double` array of length *m*, the number of constraints, containing the right-hand side of the constraints. If there are limits on both sides of the constraints, then `b` contains the lower limit of the constraints.

`c` – a `double` array of length *n*, the number of variables, containing the coefficients of the objective function

SparseLP

```
public SparseLP(int[] colPtr, int[] rowInd, double[] values, double[] b,
double[] c)
```

Description

Constructs a `SparseLP` object using Compressed Sparse Column (CSC), or Harwell-Boeing format. See Compressed Sparse Column (CSC) Format, Chapter 1.

Parameters

`colPtr` – an `int` array containing the location in `values` in which to place the first nonzero value of each succeeding column of the constraint matrix *A*. `colPtr`, `rowInd` and `values` specify the location and value of each nonzero coefficient in the constraint matrix *A* in CSC format.

`rowInd` – an `int` array containing a list of the row indices of each column of the constraint matrix *A*. `colPtr`, `rowInd` and `values` specify the location and value of each nonzero coefficient in the constraint matrix *A* in CSC format.

`values` – a `double` array containing the value of each nonzero coefficient in the constraint matrix *A*. `colPtr`, `rowInd` and `values` specify the location and value of each nonzero coefficient in the constraint matrix *A* in CSC format.

`b` – a `double` array of length *m*, number of constraints, containing the right-hand side of the constraints; if there are limits on both sides of the constraints, then `b` contains the lower limit of the constraints

`c` – a `double` array of length *n*, the number of variables, containing the coefficients of the objective function

Methods

getConstant

```
public double getConstant()
```

Description

Returns the value of the constant term in the objective function.

Returns

a `double` scalar containing the value of the constant term in the objective function

getConstraintType

```
public int[] getConstraintType()
```

Description

Returns the types of general constraints in the matrix A. See `setConstraintType`.

Returns

an `int` array containing the types of general constraints in the matrix A

getDualInfeasibility

```
public double getDualInfeasibility()
```

Description

Returns the dual infeasibility of the solution.

Returns

a `double` scalar containing the dual infeasibility of the solution, $\| c - A^T y - z + w \|$

getDualInfeasibilityTolerance

```
public double getDualInfeasibilityTolerance()
```

Description

Returns the dual infeasibility tolerance.

Returns

a `double` scalar containing the dual infeasibility tolerance

getDualSolution

```
public double[] getDualSolution()
```

Description

Returns the dual solution.

Returns

a `double` array containing the dual solution

getIterationCount

```
public int getIterationCount()
```

Description

Returns the number of iterations used by the primal-dual solver.

Returns

an `int` scalar containing the number of iterations used by the primal-dual solver

getLargestCPRatio

```
public double getLargestCPRatio()
```

Description

Returns the ratio of the largest complementarity product to the average.

Returns

a `double` scalar containing the ratio of the largest complementarity product to the average

getLowerBound

```
public double[] getLowerBound()
```

Description

Returns the lower bound on the variables.

Returns

a `double` array containing the lower bound on the variables

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the maximum number of iterations allowed for the primal-dual solver.

Returns

an `int` scalar containing the maximum number of iterations allowed for the primal-dual solver

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

getPreordering

```
public int getPreordering()
```

Description

Returns the variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of the normal equations or augmented system matrix. See `setPreordering`.

Returns

an `int` scalar containing the variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of the normal equations or augmented system matrix

getPresolve

```
public int getPresolve()
```

Description

Returns the presolve option. See `setPresolve`.

Returns

an `int` scalar containing the presolve option

getPrimalInfeasibility

```
public double getPrimalInfeasibility()
```

Description

Returns the primal infeasibility of the solution.

Returns

a `double` scalar containing the primal infeasibility of the solution, $\| x + s - u \|$

getPrimalInfeasibilityTolerance

```
public double getPrimalInfeasibilityTolerance()
```

Description

Returns the primal infeasibility tolerance.

Returns

a `double` scalar containing the primal infeasibility tolerance

getPrintLevel

```
public int getPrintLevel()
```

Description

Returns the print level. See `setPrintLevel`.

Returns

an `int` scalar containing the print level

getRelativeOptimalityTolerance

```
public double getRelativeOptimalityTolerance()
```

Description

Returns the relative optimality tolerance.

Returns

a `double` scalar containing the relative optimality tolerance

getSmallestCPRatio

```
public double getSmallestCPRatio()
```

Description

Returns the ratio of the smallest complementarity product to the average.

Returns

a `double` scalar containing the ratio of the smallest complementarity product to the average

getSolution

```
public double[] getSolution()
```

Description

Returns the solution x of the linear programming problem.

Returns

a `double` array containing the solution x of the linear programming problem

getTerminationStatus

```
public int getTerminationStatus()
```

Description

Returns the termination status for the problem.

Returns

an `int` scalar containing the termination status for the problem

status	Description
0	Optimal solution found.
1	The problem is primal infeasible (or dual unbounded).
2	The problem is primal unbounded (or dual infeasible).
3	Suboptimal solution found (accuracy problems).
4	Iterations limit `maxIterations` exceeded.
5	An error outside of the solution phase of the algorithm, e.g. a user input or a memory allocation error.

getUpperBound

```
public double[] getUpperBound()
```

Description

Returns the upper bound on the variables.

Returns

a `double` array containing the upper bound on the variables

getUpperLimit

`public double[] getUpperLimit()`

Description

Returns the upper limit of the constraints that have both a lower and an upper bound.

Returns

a `double` array containing the upper limit of the constraints that have both a lower and an upper bound. Returns `null` if the upper limit has not been set.

getViolation

`public double getViolation()`

Description

Returns the violation of the variable bounds.

Returns

a `double` scalar containing the violation of the variable bounds, $\| b - Ax \|$

setConstant

`public void setConstant(double c0)`

Description

Sets the value of the constant term in the objective function.

Parameter

`c0` – a `double` scalar containing the value of the constant term in the objective function
Default: `c0 = 0`

setConstraintType

`public void setConstraintType(int[] constraintType)`

Description

Sets the types of general constraints in the matrix A.

Parameter

`constraintType` – an `int` array of length m containing the types of general constraints in the matrix A. Let $r_i = a_{i1}x_1 + \ldots + a_{in}x_n$. Then, the value of `constraintType[i]` signifies the following:

constraintType	Constraint
0	$\mathrm{r}_i = \mathrm{b}_i$
1	$\mathrm{r}_i \leq \mathrm{bu}_i$
2	$\mathrm{r}_i \geq \mathrm{b}_i$
3	$\mathrm{b}_i \leq \mathrm{r}_i \leq \mathrm{bu}_i$
4	Ignore this constraint

Note that `constraintType[i] = 3` should only be used for constraints `i` with both a finite lower and a finite upper bound. For one-sided constraints, use `constraintType[i] = 1` or `constraintType[i] = 2`. For free constraints, use `constraintType[i] = 4`.

Default: `constraintType[i] = 0`

setDualInfeasibilityTolerance

```
public void setDualInfeasibilityTolerance(double dualTolerance)
```

Description

Sets the dual infeasibility tolerance.

Parameter

`dualTolerance` – a `double` scalar containing the dual infeasibility tolerance

Default: `dualTolerance = 1.0e-8`

setLowerBound

```
public void setLowerBound(double[] lowerBound)
```

Description

Sets the lower bound on the variables.

Parameter

`lowerBound` – a `double` array of length n containing the lower bound x_l on the variables. If there is no lower bound on a variable, then -1.0e30 should be set as the lower bound.

Default: `lowerBound[i] = 0`

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Sets the maximum number of iterations allowed for the primal-dual solver.

Parameter

`maxIterations` – an `int` scalar containing the maximum number of iterations allowed for the primal-dual solver

Default: `maxIterations = 200`

setPreordering

```
public void setPreordering(int preorder)
```

Description

Sets the variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of the normal equations or augmented system matrix.

Parameter

`preorder` – an `int` scalar containing the variant of the Minimum Degree Ordering (MDO) algorithm used in the preordering of the normal equations or augmented system matrix

preorder	Method
0	A variant of the MDO algorithm using pivotal cliques.
1	A variant of George and Liu's Quotient Minimum Degree algorithm.

Default: `preorder = 0`

setPresolve

```
public void setPresolve(int presolve)
```

Description

Sets the presolve option.

Parameter

`presolve` – an `int` containing the the presolve option to resolve the LP problem in order to reduce the problem size or to detect infeasibility or unboundedness of the problem. Depending on the number of presolve techniques used, different presolve levels can be chosen:

presolve	Description
0	No presolving.
1	Eliminate singleton rows.
2	In addition to 1, eliminate redundant (and forcing) rows.
3	In addition to 1 and 2, eliminate dominated variables.
4	In addition to 1, 2, and 3, eliminate singleton columns.
5	In addition to 1, 2, 3, and 4, eliminate doubleton rows.
6	In addition to 1, 2, 3, 4, and 5, eliminate aggregate columns.

Default: `presolve = 0`

setPrimalInfeasibilityTolerance

```
public void setPrimalInfeasibilityTolerance(double primalTolerance)
```

Description

Sets the primal infeasibility tolerance.

Parameter

`primalTolerance` – a `double` scalar containing the primal infeasibility tolerance

Default: `primalTolerance = 1.0e-8`

setPrintLevel

```
public void setPrintLevel(int printLevel)
```

Description

Sets the print level.

Parameter

`printLevel` – an `int` containing the print level

printLevel	Action
0	No printing.
1	Prints statistics on the LP problem and the solution process.

Default: `printLevel = 0`

setRelativeOptimalityTolerance

```
public void setRelativeOptimalityTolerance(double optimalityTolerance)
```

Description

Sets the relative optimality tolerance.

Parameter

`optimalityTolerance` – a `double` scalar containing the relative optimality tolerance

Default: `optimalityTolerance = 1.0e-10`

setUpperBound

```
public void setUpperBound(double[] upperBound)
```

Description

Sets the upper bound on the variables.

Parameter

`upperBound` – a `double` array of length n containing the upper bound x_u on the variables. If there is no upper bound on a variable, then 1.0e30 should be set as the upper bound.

Default: None of the variables has an upper bound

setUpperLimit

```
public void setUpperLimit(double[] bu)
```

Description

Sets the upper limit of the constraints that have both a lower and an upper bound.

Parameter

`bu` – a `double` array of length m containing the upper limit b_u of the constraints that have both a lower and an upper bound. If such a constraint exists, then method `setConstraintType` must be used to define the type of the constraints. If `constraintType[i] != 3`, i.e. if constraint `i` is not two-sided, then the corresponding entry in `bu`, `bu[i]`, is ignored.

Default: None of the constraints has an upper limit

solve

```
public void solve() throws SparseLP.DiagonalWeightMatrixException,
SparseLP.CholeskyFactorizationAccuracyException,
SparseLP.PrimalUnboundedException, SparseLP.PrimalInfeasibleException,
SparseLP.DualInfeasibleException, SparseLP.InitialSolutionInfeasibleException,
SparseLP.TooManyIterationsException, SparseLP.ProblemUnboundedException,
SparseLP.ZeroColumnException, SparseLP.ZeroRowException,
SparseLP.IncorrectlyEliminatedException, SparseLP.IncorrectlyActiveException,
SparseLP.IllegalBoundsException
```

Description

Solves the sparse linear programming problem by an infeasible primal-dual interior-point method.

Exceptions

`DiagonalWeightMatrixException` is thrown if a diagonal element of the diagonal weight matrix is too small

`CholeskyFactorizationAccuracyException` is thrown if the Cholesky factorization failed because of accuracy problems

`PrimalUnboundedException` is thrown if the primal problem is unbounded

`PrimalInfeasibleException` is thrown if the primal problem is infeasible

`DualInfeasibleException` is thrown if the dual problem is infeasible

`InitialSolutionInfeasibleException` is thrown if the initial solution for the one-row linear program is infeasible

`TooManyIterationsException` is thrown if the maximum number of iterations has been exceeded

`ProblemUnboundedException` is thrown if the problem is unbounded

`ZeroColumnException` is thrown if a column of the constraint matrix has no entries

`ZeroRowException` is thrown if a row of the constraint matrix has no entries

`IncorrectlyEliminatedException` is thrown if one or more LP variables are falsely characterized by the internal presolver

`IncorrectlyActiveException` is thrown if one or more LP variables are falsely characterized by the internal presolver

`IllegalBoundsException` is thrown if the lower bound is greater than the upper bound

Example 1: Sparse Linear Programming

The linear programming problem

$$\min f(x) = 2x_1 - 8x_2 + 3x_3 \quad \text{subject to} \quad \begin{aligned} x_1 + 3x_2 &\le 3 \\ 2x_2 + 3x_3 &\le 6 \\ x_1 + x_2 + x_3 &\ge 2 \\ -1 \le x_1 &\le 5 \\ 0 \le x_2 &\le 7 \\ 0 \le x_3 &\le 9 \end{aligned}$$

is solved.

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

public class SparseLPEx1 {

    public static void main(String args[]) throws Exception {
        double[] b = {3.0, 6.0, 2.0};
        double[] c = {2.0, -8.0, 3.0};
        double[] xlb = {-1.0, 0.0, 0.0};
        double[] xub = {5.0, 7.0, 9.0};
        int[] irtype = {1, 1, 2};

        SparseMatrix a = new SparseMatrix(3, 3);
        a.set(0, 0, 1.0);
        a.set(0, 1, 3.0);
        a.set(1, 1, 2.0);
        a.set(1, 2, 3.0);
        a.set(2, 0, 1.0);
        a.set(2, 1, 1.0);
        a.set(2, 2, 1.0);

        // Solve problem.
        SparseLP lp = new SparseLP(a, b, c);
        lp.setLowerBound(xlb);
        lp.setUpperBound(xub);
        lp.setConstraintType(irtype);
        lp.solve();

        // Print solution.
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNoColumnLabels();
        new PrintMatrix("Solution").print(mf, lp.getSolution());

        System.out.printf("Objective:  %.2f\n", lp.getOptimalValue());
    }
}
```

Output

```
 Solution

0  -0.375
1   1.125
2   1.25

Objective:  -6.00
```

Example 2: Solve Problem Defined In an MPS File

This example demonstrates how the class `MPSReader` can be used with `SparseLP` to solve a linear programming problem defined in an MPS file. The MPS file used in this example is an uncompressed version of the file 'afiro', available from http://www.netlib.org/lp/data/.

```
import com.imsl.io.*;
import com.imsl.math.*;
import java.io.*;
import java.text.*;

public class SparseLPEx2 {

    public static void main(String args[]) throws Exception {
        // Read from MPS file.
        InputStream stream = SparseLPEx2.class.getResourceAsStream("afiro");
        Reader reader = new InputStreamReader(stream);
        MPSReader mps = new MPSReader();
        mps.read(reader);

        // Solve problem.
        SparseLP lp = new SparseLP(mps);
        lp.setPresolve(6);
        lp.solve();

        // Print solution.
        System.out.println("Problem Name: " + mps.getName());
        System.out.printf("Objective:  %.2f\n", lp.getOptimalValue());

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(1);
        nf.setMinimumFractionDigits(1);

        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNumberFormat(nf);
        mf.setNoColumnLabels();

        new PrintMatrix("Solution:").print(mf, lp.getSolution());

        // Close all resources.
        reader.close();
        stream.close();
    }
}
```

Output

```
Problem Name: AFIRO
Objective:  -464.75
 Solution:

 0   80.0
 1   25.5
 2   54.5
 3   84.8
 4   65.4
 5    0.0
 6    0.0
 7    0.0
 8    0.0
 9    0.0
```

```
10    0.0
11    0.0
12   18.2
13   47.2
14   69.4
15  500.0
16  475.9
17   24.1
18    0.0
19  215.0
20  141.7
21    0.0
22    0.0
23    0.0
24    0.0
25    0.0
26    0.0
27    0.0
28  339.9
29  242.3
30   60.9
31    0.0
```

SparseLP.DiagonalWeightMatrixException class

`static public class com.imsl.math.SparseLP.DiagonalWeightMatrixException`
`extends com.imsl.IMSLException`

A diagonal element of the diagonal weight matrix is too small.

Constructors

SparseLP.DiagonalWeightMatrixException

`public SparseLP.DiagonalWeightMatrixException(String message)`

Description

A diagonal element of the diagonal weight matrix is too small.

Parameter

`message` – a `String` containing the error message

SparseLP.DiagonalWeightMatrixException

`public SparseLP.DiagonalWeightMatrixException(String key, Object[] arguments)`

Description

A diagonal element of the diagonal weight matrix is too small.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.CholeskyFactorizationAccuracyException class

```
static public class
com.imsl.math.SparseLP.CholeskyFactorizationAccuracyException extends
com.imsl.IMSLException
```

The Cholesky factorization failed because of accuracy problems.

Constructors

SparseLP.CholeskyFactorizationAccuracyException

```
public SparseLP.CholeskyFactorizationAccuracyException(String message)
```

Description

The Cholesky factorization failed because of accuracy problems.

Parameter

`message` – a `String` containing the error message

SparseLP.CholeskyFactorizationAccuracyException

```
public SparseLP.CholeskyFactorizationAccuracyException(String key, Object[]
arguments)
```

Description

The Cholesky factorization failed because of accuracy problems.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.PrimalUnboundedException class

```
static public class com.imsl.math.SparseLP.PrimalUnboundedException extends
com.imsl.IMSLException
```

The primal problem is unbounded.

Constructors

SparseLP.PrimalUnboundedException

```
public SparseLP.PrimalUnboundedException(String message)
```

Description

The primal problem is unbounded.

Parameter

`message` – a `String` containing the error message

SparseLP.PrimalUnboundedException

```
public SparseLP.PrimalUnboundedException(String key, Object[] arguments)
```

Description

The primal problem is unbounded.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.PrimalInfeasibleException class

```
static public class com.imsl.math.SparseLP.PrimalInfeasibleException extends
com.imsl.IMSLException
```

The primal problem is infeasible.

Constructors

SparseLP.PrimalInfeasibleException

`public SparseLP.PrimalInfeasibleException(String message)`

Description

The primal problem is infeasible.

Parameter

`message` – a `String` containing the error message

SparseLP.PrimalInfeasibleException

`public SparseLP.PrimalInfeasibleException(String key, Object[] arguments)`

Description

The primal problem is infeasible.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.DualInfeasibleException class

`static public class com.imsl.math.SparseLP.DualInfeasibleException extends com.imsl.IMSLException`

The dual problem is infeasible.

Constructors

SparseLP.DualInfeasibleException

`public SparseLP.DualInfeasibleException(String message)`

Description

The dual problem is infeasible.

Parameter

`message` – a `String` containing the error message

SparseLP.DualInfeasibleException

`public SparseLP.DualInfeasibleException(String key, Object[] arguments)`

Description

The dual problem is infeasible.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.InitialSolutionInfeasibleException class

```
static public class com.imsl.math.SparseLP.InitialSolutionInfeasibleException
extends com.imsl.IMSLException
```

The initial solution for the one-row linear program is infeasible.

Constructors

SparseLP.InitialSolutionInfeasibleException

```
public SparseLP.InitialSolutionInfeasibleException(String message)
```

Description

The initial solution for the one-row linear program is infeasible.

Parameter

`message` – a `String` containing the error message

SparseLP.InitialSolutionInfeasibleException

```
public SparseLP.InitialSolutionInfeasibleException(String key, Object[]
arguments)
```

Description

The initial solution for the one-row linear program is infeasible.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.TooManyIterationsException class

`static public class com.imsl.math.SparseLP.TooManyIterationsException extends com.imsl.IMSLException`

The maximum number of iterations has been exceeded.

Constructors

SparseLP.TooManyIterationsException

`public SparseLP.TooManyIterationsException(String message)`

Description

The maximum number of iterations has been exceeded.

Parameter

`message` – a `String` containing the error message

SparseLP.TooManyIterationsException

`public SparseLP.TooManyIterationsException(String key, Object[] arguments)`

Description

The maximum number of iterations has been exceeded.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.ProblemUnboundedException class

`static public class com.imsl.math.SparseLP.ProblemUnboundedException extends com.imsl.IMSLException`

The problem is unbounded.

Constructors

SparseLP.ProblemUnboundedException

```
public SparseLP.ProblemUnboundedException(String message)
```

Description

The problem is unbounded.

Parameter

`message` – a `String` containing the error message

SparseLP.ProblemUnboundedException

```
public SparseLP.ProblemUnboundedException(String key, Object[] arguments)
```

Description

The problem is unbounded.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.ZeroColumnException class

```
static public class com.imsl.math.SparseLP.ZeroColumnException extends
com.imsl.IMSLException
```

A column of the constraint matrix has no entries.

Constructors

SparseLP.ZeroColumnException

```
public SparseLP.ZeroColumnException(String message)
```

Description

A column of the constraint matrix has no entries.

Parameter

`message` – a `String` containing the error message

SparseLP.ZeroColumnException

```
public SparseLP.ZeroColumnException(String key, Object[] arguments)
```

Description

A column of the constraint matrix has no entries.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.ZeroRowException class

```
static public class com.imsl.math.SparseLP.ZeroRowException extends
com.imsl.IMSLException
```

A row of the constraint matrix has no entries.

Constructors

SparseLP.ZeroRowException

```
public SparseLP.ZeroRowException(String message)
```

Description

A row of the constraint matrix has no entries.

Parameter

`message` – a `String` containing the error message

SparseLP.ZeroRowException

```
public SparseLP.ZeroRowException(String key, Object[] arguments)
```

Description

A row of the constraint matrix has no entries.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.IncorrectlyEliminatedException class

`static public class com.imsl.math.SparseLP.IncorrectlyEliminatedException extends com.imsl.IMSLException`

One or more LP variables are falsely characterized by the internal presolver.

Constructors

SparseLP.IncorrectlyEliminatedException

`public SparseLP.IncorrectlyEliminatedException(String message)`

Description

One or more LP variables are falsely characterized by the internal presolver.

Parameter

`message` – a `String` containing the error message

SparseLP.IncorrectlyEliminatedException

`public SparseLP.IncorrectlyEliminatedException(String key, Object[] arguments)`

Description

One or more LP variables are falsely characterized by the internal presolver.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.IncorrectlyActiveException class

`static public class com.imsl.math.SparseLP.IncorrectlyActiveException extends com.imsl.IMSLException`

One or more LP variables are falsely characterized by the internal presolver.

Constructors

SparseLP.IncorrectlyActiveException

`public SparseLP.IncorrectlyActiveException(String message)`

Description

One or more LP variables are falsely characterized by the internal presolver.

Parameter

`message` – a `String` containing the error message

SparseLP.IncorrectlyActiveException

`public SparseLP.IncorrectlyActiveException(String key, Object[] arguments)`

Description

One or more LP variables are falsely characterized by the internal presolver.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

SparseLP.IllegalBoundsException class

```
static public class com.imsl.math.SparseLP.IllegalBoundsException extends
com.imsl.IMSLException
```

The lower bound is greater than the upper bound.

Constructors

SparseLP.IllegalBoundsException

`public SparseLP.IllegalBoundsException(String message)`

Description

The lower bound is greater than the upper bound.

Parameter

`message` – a `String` containing the error message

SparseLP.IllegalBoundsException

`public SparseLP.IllegalBoundsException(String key, Object[] arguments)`

Description

The lower bound is greater than the upper bound.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

DenseLP class

```
public class com.imsl.math.DenseLP implements Serializable, Cloneable
```

Solves a linear programming problem using an active set strategy.

Class `DenseLP` uses an active set strategy 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.

If the linear constraints are infeasible an L_1 solution to the constraints are used as a replacement for the stated constraints. An exception is thrown but a generalized solution is computed and available using methods `getPrimalSolution` or `getDualSolution`. Similar comments hold for any of the three additional conditions: (1) There are multiple solutions; (2) some constraints are discarded, or (3) cycling in the algorithm is identified.

Refer to the following paper for further information:

Krogh, Fred, T. (2005), see *An Algorithm for Linear Programming*
http://mathalacarte.com/fkrogh/pub/lp.pdf

Constructors

DenseLP

`public DenseLP(MPSReader mps)`

Description

Constructor using an MPSReader object.

Parameter

`mps` – An `MPSReader` object specifying the Linear Programming problem.

Exception

`IllegalArgumentException` is thrown if the problem dimensions are not consistent.

DenseLP

`public DenseLP(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.

Exception

`IllegalArgumentException` is thrown if the dimensions of `a`, `b.length`, and `c.length` are not consistent.

Methods

clone

`public 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.

getIterationCount

```
public int getIterationCount()
```

Description

Returns the iteration count.

Returns

an `int` scalar containing the iteration count.

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. If a solution has not been computed, `Double.NaN` is returned.

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`.

Parameter

`constraintType` – an `int` array containing the types of general constraints. Let $r_i = a_{i1}x_1 + ... + a_{in}x_n$. Then the value of `constraintType[i]` signifies the following:

constraintType	Constraint
0	$\mathrm{r}_i = \mathrm{b}_i$
1	$\mathrm{r}_i \leq \mathrm{b}_i$ *or* $\mathrm{r}_i \leq \mathrm{b}_{\mathrm{u}_i}$ *if supplied*
2	$\mathrm{r}_i \geq \mathrm{b}_i$
3	$\mathrm{b}_i \leq \mathrm{r}_i \leq \mathrm{b}_{\mathrm{u}_i}$

Default=0.

setLowerBound

```
public void setLowerBound(double[] lowerBound)
```

Description

Sets the lower bound, x_l, on the variables. If there is no lower bound on a variable, then 1.0e30 should be set as the lower bound.

Parameter

`lowerBound` – a `double` array containing the lower bound on the variables. Default = 0.

setRefinementType

```
public void setRefinementType(int iRefinement)
```

Description

Set the type of refinement used.

Parameter

`iRefinement` – An `int` scalar value which defines the type of refinement to be used. The possible settings are:

iRefinement	Action
0	No refinement. Always compute dual. Default.
1	Iterative refinement.
2	Use extended refinement. Iterate until no more progress.

If refinement is used, the coefficient matrices and other data are saved at the beginning of the computation. When finished this data together with the solution obtained is checked for consistency. If the discrepancy is too large, the solution process is restarted using the problem data just after processing the equalities, but with the final x values and final active set.

setUpperBound

```
public void setUpperBound(double[] upperBound)
```

Description

Sets the upper bound, x_u, on the variables. If there is no upper bound on a variable, then -1.0e30 should be set as the upper bound.

Parameter

`upperBound` – a `double` array containing the upper bound on the variables. The default is no upper bound.

setUpperLimit

```
public void setUpperLimit(double[] upperLimit)
```

Description

Sets the upper limit of the constraints.

Parameter

`upperLimit` – a `double` array containing the upper limit, b_u, of the constraints.

solve

```
final public void solve() throws DenseLP.BoundsInconsistentException,
DenseLP.NoAcceptablePivotException, DenseLP.ProblemUnboundedException,
DenseLP.ProblemVacuousException, DenseLP.AllConstraintsNotSatisfiedException,
DenseLP.MultipleSolutionsException, DenseLP.SomeConstraintsDiscardedException,
DenseLP.CyclingOccurringException
```

Description

Solves the problem using an active set method. `solve` must be invoked prior to calling any of the "get" methods.

Exceptions

`BoundsInconsistentException` is thrown if the bounds are inconsistent.

`NoAcceptablePivotException` is thrown if an acceptable pivot could not be found.

`ProblemUnboundedException` is thrown if there is no finite solution to the problem.

`ProblemVacuousException` is thrown if the problem is vacuous.

`AllConstraintsNotSatisfiedException` is thrown if all constraints are not satisfied.

Example 1: Linear Programming

The linear programming problem in the standard form

$$\min f(x) = -x_1 - 3x_2$$

subject to:

$x_1 + x_2 + x_3 = 1.5$
$x_1 + x_2 - x_4 = 0.5$
$x_1 + x_5 = 1.0$
$x_2 + x_6 = 1.0$
$x_i \geq 0, \text{ for } i = 1, \ldots, 6$

is solved.

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

public class DenseLPEx1 {

    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};

        DenseLP zf = new DenseLP(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:

$0.5 \leq x_1 + x_2 \leq 1.5$
$0 \leq x_1 \leq 1.0$
$0 \leq x_2 \leq 1.0$

is solved.

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

public class DenseLPEx2 {

    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};

        DenseLP zf = new DenseLP(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
```

Example 3: Linear Programming Maximize Example

Maximize the linear programming problem

$$\max f(x) = 10x_1 + 15x_2 + 15x_3 + 13x_4 + 9x_5$$

subject to the following set of restrictions:

$100x_1 + 50x_2 + 50x_3 + 40x_4 + 120x_5 \leq 300$
$40x_1 + 50x_2 + 50x_3 + 15x_4 + 30x_5 \leq 40$
$0 \leq x_1 \leq 1.0; 0 \leq x_2 \leq 1.0; 0 \leq x_3 \leq 1.0; 0 \leq x_4 \leq 1.0; 0 \leq x_5 \leq 1.0$

Since DenseLP *minimizes*, the sign of the objective function must be changed to compute this solution. The signs of the dual solution components and the optimal value must also be changed. Because x_2 and x_3 are not uniquely determined within the bounds, this problem has a convex family of solutions. DenseLP issues an exception, `MultipleSolutionsException`. A particular solution is available and retrieved in the `finally` block.

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

public class DenseLPEx3 {

    public static void main(String[] args) throws Exception {

        int[] constraintType = {1, 1};  /* Ax <= b */

        double[] lowerVariableBound = {0.0, 0.0, 0.0, 0.0, 0.0};
        double[] upperVariableBound = {1.0, 1.0, 1.0, 1.0, 1.0};
```

```
        double[][] A = {
            {100.0, 50.0, 50.0, 40.0, 120.0},
            {40.0, 50.0, 50.0, 15.0, 30.0}
        };
        double[] b = {300.0, 40.0};  /* constraint type Ax <= b */

        double[] c = {10.0, 15.0, 15.0, 13.0, 9.0};

        /*  Since DenseLP minimizes, change signs of the
         objective coefficients. */
        double[] negC = new double[c.length];
        for (int i = 0; i < c.length; i++) {
            negC[i] = -c[i];
        }

        DenseLP zf = new DenseLP(A, b, negC);
        zf.setLowerBound(lowerVariableBound);
        zf.setConstraintType(constraintType);
        zf.setUpperBound(upperVariableBound);

        try {
            zf.solve();
        } catch (DenseLP.MultipleSolutionsException e) {
            /* x_2 and x_3 are not uniquely determined, expect multiple solutions.
             * Catch the exception, but continue to print result found. */
            System.out.println("Multiple solutions giving "
                    + "essentially the same minimum exist.");
        } finally {
            double[] dSolution = zf.getDualSolution();
            /* Change the sign of the dual solution and optimal value
             * since DenseLP minimizes. */
            for (int i = 0; i < dSolution.length; i++) {
                dSolution[i] = -dSolution[i];
            }
            double optimalValue = -zf.getOptimalValue();

            new PrintMatrix("Solution").print(zf.getPrimalSolution());
            new PrintMatrix("Dual Solution").print(dSolution);
            System.out.println("Optimal Value = " + optimalValue);
        }
    }
}
```

Output

```
Multiple solutions giving essentially the same minimum exist.
 Solution
    0
0  0
1  0.257
2  0.243
3  1
4  0

Dual Solution
```

```
    0
0  -0
1   0.3

Optimal Value = 20.5
```

DenseLP.WrongConstraintTypeException class

`static public class com.imsl.math.DenseLP.WrongConstraintTypeException extends com.imsl.IMSLException`

Constructors

DenseLP.WrongConstraintTypeException

`public DenseLP.WrongConstraintTypeException(String message)`

DenseLP.WrongConstraintTypeException

`public DenseLP.WrongConstraintTypeException(String key, Object[] arguments)`

DenseLP.BoundsInconsistentException class

`static public class com.imsl.math.DenseLP.BoundsInconsistentException extends com.imsl.IMSLException`

The bounds given are inconsistent.

Constructors

DenseLP.BoundsInconsistentException

`public DenseLP.BoundsInconsistentException(String message)`

Description

The bounds given are inconsistent.

Parameter

`message` – a `String` containing the error message

DenseLP.BoundsInconsistentException

`public DenseLP.BoundsInconsistentException(String key, Object[] arguments)`

Description

The bounds given are inconsistent.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.NoAcceptablePivotException class

`static public class com.imsl.math.DenseLP.NoAcceptablePivotException extends com.imsl.IMSLException`

No acceptable pivot could be found.

Constructors

DenseLP.NoAcceptablePivotException

`public DenseLP.NoAcceptablePivotException()`

Description

No acceptable pivot could be found.

DenseLP.NoAcceptablePivotException

`public DenseLP.NoAcceptablePivotException(String message)`

Description

No acceptable pivot could be found.

Parameter

`message` – a `String` containing the error message

DenseLP.NoAcceptablePivotException

`public DenseLP.NoAcceptablePivotException(String key, Object[] arguments)`

Description

No acceptable pivot could be found.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.ProblemUnboundedException class

```
static public class com.imsl.math.DenseLP.ProblemUnboundedException extends
com.imsl.IMSLException
```

The problem is unbounded.

Constructors

DenseLP.ProblemUnboundedException

```
public DenseLP.ProblemUnboundedException()
```

Description

The problem is unbounded.

DenseLP.ProblemUnboundedException

```
public DenseLP.ProblemUnboundedException(String message)
```

Description

The problem is unbounded.

Parameter

`message` – a `String` containing the error message

DenseLP.ProblemUnboundedException

```
public DenseLP.ProblemUnboundedException(String key, Object[] arguments)
```

Description

The problem is unbounded.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.ProblemVacuousException class

`static public class com.imsl.math.DenseLP.ProblemVacuousException extends com.imsl.IMSLException`

The problem is vaxuous.

Constructors

DenseLP.ProblemVacuousException

`public DenseLP.ProblemVacuousException()`

Description

The problem is vaxuous.

DenseLP.ProblemVacuousException

`public DenseLP.ProblemVacuousException(String message)`

Description

The problem is vaxuous.

Parameter

`message` – a `String` containing the error message

DenseLP.ProblemVacuousException

`public DenseLP.ProblemVacuousException(String key, Object[] arguments)`

Description

The problem is vaxuous.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.MultipleSolutionsException class

`static public class com.imsl.math.DenseLP.MultipleSolutionsException extends com.imsl.IMSLException`

The problem has multiple solutions giving essentially the same minimum.

Constructors

DenseLP.MultipleSolutionsException

`public DenseLP.MultipleSolutionsException()`

Description

The problem has multiple solutions giving essentially the same minimum.

DenseLP.MultipleSolutionsException

`public DenseLP.MultipleSolutionsException(String message)`

Description

The problem has multiple solutions giving essentially the same minimum.

Parameter

`message` – a `String` containing the error message

DenseLP.MultipleSolutionsException

`public DenseLP.MultipleSolutionsException(String key, Object[] arguments)`

Description

The problem has multiple solutions giving essentially the same minimum.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.AllConstraintsNotSatisfiedException class

`static public class com.imsl.math.DenseLP.AllConstraintsNotSatisfiedException extends com.imsl.IMSLException`

All constraints are not satisfied.

L1 minimization was applied to all constraints (including bounds and simple variables) but the equalities, to approximate violated non-equalities as well as possible. If a feasible solution is possible then try using refinement.

Constructors

DenseLP.AllConstraintsNotSatisfiedException

`public DenseLP.AllConstraintsNotSatisfiedException()`

Description

All constraints are not satisfied.

DenseLP.AllConstraintsNotSatisfiedException

`public DenseLP.AllConstraintsNotSatisfiedException(String message)`

Description

All constraints are not satisfied.

Parameter

`message` – a `String` containing the error message

DenseLP.AllConstraintsNotSatisfiedException

`public DenseLP.AllConstraintsNotSatisfiedException(String key, Object[] arguments)`

Description

All constraints are not satisfied.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.SomeConstraintsDiscardedException class

```
static public class com.imsl.math.DenseLP.SomeConstraintsDiscardedException
extends com.imsl.IMSLException
```

Some constraints were discarded because they were too linearly dependent on other active constraints.

Constructors

DenseLP.SomeConstraintsDiscardedException

`public DenseLP.SomeConstraintsDiscardedException()`

Description

Some constraints were discarded because they were too linearly dependent on other active constraints.

DenseLP.SomeConstraintsDiscardedException

```
public DenseLP.SomeConstraintsDiscardedException(String message)
```

Description

Some constraints were discarded because they were too linearly dependent on other active constraints.

Parameter

`message` – a `String` containing the error message

DenseLP.SomeConstraintsDiscardedException

```
public DenseLP.SomeConstraintsDiscardedException(String key, Object[]
arguments)
```

Description

Some constraints were discarded because they were too linearly dependent on other active constraints.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DenseLP.CyclingOccurringException class

```
static public class com.imsl.math.DenseLP.CyclingOccurringException extends
com.imsl.IMSLException
```

The algorithm appears to be cycling. Using refinement may help.

Constructors

DenseLP.CyclingOccurringException

```
public DenseLP.CyclingOccurringException()
```

Description

The algorithm appears to be cycling. Using refinement may help.

DenseLP.CyclingOccurringException

```
public DenseLP.CyclingOccurringException(String message)
```

Description

The algorithm appears to be cycling. Using refinement may help.

Parameter

`message` – a `String` containing the error message

DenseLP.CyclingOccurringException

```
public DenseLP.CyclingOccurringException(String key, Object[] arguments)
```

Description

The algorithm appears to be cycling. Using refinement may help.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

QuadraticProgramming class

```
public class com.imsl.math.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.

If the constraints are infeasible an exception is thrown. See Example 3 where the exception is caught and printed.

Field

EPSILON_SMALL

```
static final public 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
QuadraticProgramming.InconsistentSystemException,
QuadraticProgramming.ProblemUnboundedException,
QuadraticProgramming.NoLPSolutionException,
QuadraticProgramming.SolutionNotFoundException
```

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.

Exception

`InconsistentSystemException` if the system of constraints is inconsistent and there is no solution.

Methods

getDual

```
public double[] getDual()
```

Description

Returns the dual (Lagrange multipliers).

getOptimalValue

```
public double getOptimalValue()
```

Description

Returns the optimal value.

Returns

a `double`, the optimal value.

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 Exception {
        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
```

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 Exception {
        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
```

Example 3: Solve a Quadratic Programming Problem with Inconsistent System Constraints

In the quadratic programming problem variables 2 and 6 are fixed at the value zero by the equality constraints. The inequalities propose that the sums of the variables are at least 5.1 and no more than 4.9. These last two are inconsistent conditions, causing the NoLPSolutionException to be thrown.

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

public class QuadraticProgrammingEx3 {

    public static void main(String[] args) {

        double[][] h = {
            {2.000, 0.000, 0.000, 0.000, 0.000, 0.000},
            {0.000, 2.000, 0.000, 0.000, 0.000, 0.000},
            {0.000, 0.000, 2.000, 0.000, 0.000, 0.000},
            {0.000, 0.000, 0.000, 2.000, 0.000, 0.000},
            {0.000, 0.000, 0.000, 0.000, 2.000, 0.000},
            {0.000, 0.000, 0.000, 0.000, 0.000, 2.000}
```

```
        };
        double[] g = {5.000, 5.000, 5.000, 5.000, 5.000, 5.000};

        double[][] aEquality = {
            {0.000, 1.000, 0.000, 0.000, 0.000, 0.000},
            {0.000, 0.000, 0.000, 0.000, 0.000, 1.000}
        };
        double[] bEquality = {0.000, 0.000};

        double[][] aInequality = {
            {1.000, 1.000, 1.000, 1.000, 1.000, 1.000},
            {-1.000, -1.000, -1.000, -1.000, -1.000, -1.000}
        };
        double delta = 0.1;  // change to 0.0 to pass
        double[] bInequality = {5 + delta, -5 + delta};

        try {
            QuadraticProgramming qp
                    = new QuadraticProgramming(h, g, aEquality, bEquality,
                            aInequality, bInequality);
            double x[] = qp.getSolution();
            new com.imsl.math.PrintMatrix("Solution").print(x);
        } catch (Exception e) {
            WriteCutString(e.getMessage(), 72);
        }
    }

    public static void WriteCutString(String value, int interval) {
        int rem = value.length() % interval;
        int len = value.length() / interval + rem / (1 > rem ? 1 : rem);

        for (int i = 0; i < len; i++) {
            System.out.println(value.substring(i * interval,
                    (i * interval) + (interval < (value.length() - i * interval)
                            ? interval : (value.length() - i * interval))));
        }
    }
}
```

Output

```
No solution for the LP problem was found. All constraints are not satisf
ied. L1 minimization was applied to all constraints (including bounds an
d simple variables) but the equalities, to approximate violated non-equa
lities as well as possible.  If a feasible solution is possible then try
 using refinement.
```

QuadraticProgramming.InconsistentSystemException class

```
static public class
com.imsl.math.QuadraticProgramming.InconsistentSystemException extends
com.imsl.IMSLException
```

The system of constraints is inconsistent. There is no solution.

Constructors

QuadraticProgramming.InconsistentSystemException

```
public QuadraticProgramming.InconsistentSystemException()
```

Description

The system of constraints is inconsistent. There is no solution.

QuadraticProgramming.InconsistentSystemException

```
public QuadraticProgramming.InconsistentSystemException(String message)
```

QuadraticProgramming.ProblemUnboundedException class

```
static public class
com.imsl.math.QuadraticProgramming.ProblemUnboundedException extends
com.imsl.IMSLException
```

The object value for the problem is unbounded.

Constructors

QuadraticProgramming.ProblemUnboundedException

```
public QuadraticProgramming.ProblemUnboundedException()
```

Description

The object value for the problem is unbounded. Numerical difficulty has occurred.

QuadraticProgramming.ProblemUnboundedException

`public QuadraticProgramming.ProblemUnboundedException(String message)`

Description

The object value for the problem is unbounded. Numerical difficulty has occurred.

Parameter

`message` – a `string` containing the exception message.

QuadraticProgramming.NoLPSolutionException class

```
static public class com.imsl.math.QuadraticProgramming.NoLPSolutionException
extends com.imsl.IMSLException
```

No solution for the LP problem with h = 0 was found by `DenseLP`.

Constructors

QuadraticProgramming.NoLPSolutionException

`public QuadraticProgramming.NoLPSolutionException()`

Description

No solution for the LP problem with h = 0 was found by `DenseLP`.

QuadraticProgramming.NoLPSolutionException

`public QuadraticProgramming.NoLPSolutionException(String message)`

Description

No solution for the LP problem with h = 0 was found by `DenseLP`.

Parameter

`message` – a `string` containing the exception message.

QuadraticProgramming.NoLPSolutionException

```
public QuadraticProgramming.NoLPSolutionException(String key, Object[]
arguments)
```

Description

No solution for the LP problem with h = 0 was found by `DenseLP`.

Parameters

`key` – a `string` containing the exception message.

`arguments` – An `Object` containing the exception message from `DenseLP`.

QuadraticProgramming.SolutionNotFoundException class

```
static public class
com.imsl.math.QuadraticProgramming.SolutionNotFoundException extends
com.imsl.IMSLException
```

A solution was not found. Try using DenseLP.

Constructors

QuadraticProgramming.SolutionNotFoundException

```
public QuadraticProgramming.SolutionNotFoundException()
```

Description

A solution was not found. Try using DenseLP.

QuadraticProgramming.SolutionNotFoundException

```
public QuadraticProgramming.SolutionNotFoundException(String message)
```

Description

A solution was not found. Try using DenseLP.

Parameter

`message` – a `string` containing the exception message.

MinConGenLin class

```
public class com.imsl.math.MinConGenLin implements Serializable, Cloneable
```

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, λ^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 λ^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 α^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

$$\left\| \nabla f(x^k) - A^k \lambda^K \right\|_2 \le \tau$$

is satisfied. Here τ is the supplied tolerance. For more details, see Powell (1988, 1989).

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` – An `int` scalar containing the number of variables.

`ncon` – An `int` scalar containing the number of linear constraints (excluding simple bounds).

`neq` – An `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`

Exception

`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

an `int` array containing the indices of the final active constraints.

getFinalActiveConstraintsNum

`public int getFinalActiveConstraintsNum()`

Description

Returns the final number of active constraints.

Returns

an `int` scalar containing the final number of 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.

getNumberOfThreads

`public int getNumberOfThreads()`

Description

Returns the number of `java.lang.Thread` instances used for parallel processing.

Returns

an `int` containing the number of `java.lang.Thread` instances used for parallel processing.

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.

Parameter

`guess` – a `double` array containing an initial guess.

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing. If `numberOfThreads` is greater than 1, then interface `Function.f` is evaluated in parallel and `Function.f` must be thread-safe. Otherwise, unexpected behavior can occur.

Default: `numberOfThreads` = 1.

setTolerance

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

Description

Sets the nonnegative tolerance on the first order conditions at the calculated solution.

Parameter

`tolerance` – a `double` scalar containing the tolerance.

solve

```
final public void solve() throws MinConGenLin.ConstraintsInconsistentException,
MinConGenLin.VarBoundsInconsistentException,
MinConGenLin.ConstraintsNotSatisfiedException,
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
```

MinConGenLin.Function interface

```
public interface com.imsl.math.MinConGenLin.Function
```

Public interface for the user-supplied function to evaluate the function to be minimized.

Method

f

```
public double f(double[] x)
```

Description

Public interface for the function to be minimized.

Parameter

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

MinConGenLin.Gradient interface

```
public interface com.imsl.math.MinConGenLin.Gradient implements
com.imsl.math.MinConGenLin.Function
```

Public interface for the user-supplied function to compute the gradient.

Method

gradient

```
public 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.

MinConGenLin.ConstraintsInconsistentException class

```
static public class com.imsl.math.MinConGenLin.ConstraintsInconsistentException
extends com.imsl.IMSLException
```

The equality constraints are inconsistent.

Constructors

MinConGenLin.ConstraintsInconsistentException

```
public MinConGenLin.ConstraintsInconsistentException(String message)
```

Description

Constructs a `ConstraintsInconsistentException` object.

Parameter

`message` – a `String` containing the error message

MinConGenLin.ConstraintsInconsistentException

```
public MinConGenLin.ConstraintsInconsistentException(String key, Object[]
arguments)
```

Description

Constructs a `ConstraintsInconsistentException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConGenLin.VarBoundsInconsistentException class

`static public class com.imsl.math.MinConGenLin.VarBoundsInconsistentException extends com.imsl.IMSLException`

The equality constraints and the bounds on the variables are found to be inconsistent.

Constructors

MinConGenLin.VarBoundsInconsistentException

`public MinConGenLin.VarBoundsInconsistentException(String message)`

Description

Constructs a `VarBoundsInconsistentException` object.

Parameter

`message` – a `String` containing the error message

MinConGenLin.VarBoundsInconsistentException

`public MinConGenLin.VarBoundsInconsistentException(String key, Object[] arguments)`

Description

Constructs a `VarBoundsInconsistentException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConGenLin.ConstraintsNotSatisfiedException class

`static public class com.imsl.math.MinConGenLin.ConstraintsNotSatisfiedException extends com.imsl.IMSLException`

No vector x satisfies all of the constraints.

Constructors

MinConGenLin.ConstraintsNotSatisfiedException

```
public MinConGenLin.ConstraintsNotSatisfiedException(String message)
```

Description

Constructs a `ConstraintsNotSatisfiedException` object.

Parameter

`message` – a `String` containing the error message

MinConGenLin.ConstraintsNotSatisfiedException

```
public MinConGenLin.ConstraintsNotSatisfiedException(String key, Object[]
arguments)
```

Description

Constructs a `ConstraintsNotSatisfiedException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConGenLin.EqualityConstraintsException class

```
static public class com.imsl.math.MinConGenLin.EqualityConstraintsException
extends com.imsl.IMSLException
```

the variables are determined by the equality constraints.

Constructors

MinConGenLin.EqualityConstraintsException

```
public MinConGenLin.EqualityConstraintsException(String message)
```

Description

Constructs a `EqualityConstraintsException` object.

Parameter

`message` – a `String` containing the error message

MinConGenLin.EqualityConstraintsException

```
public MinConGenLin.EqualityConstraintsException(String key, Object[]
arguments)
```

Description

Constructs a `EqualityConstraintsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

BoundedLeastSquares class

```
public class com.imsl.math.BoundedLeastSquares implements Serializable,
Cloneable
```

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 ≥ 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 μ 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:

$$\|g(x_i)\| \leq \varepsilon, l_i < x_i < u_i$$

$$g(x_i) < 0, x_i = u_i$$

$$g(x_i) > 0, x_i = l_i$$

where ε 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).

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` – an `int` scalar containing the number of functions

`nVariables` – an `int` scalar containing the number of variables

`boundType` – an `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

Exception

`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

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the number of `java.lang.Thread` instances used for parallel processing.

Returns

an `int` containing the number of `java.lang.Thread` instances used for parallel processing.

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.

Parameter

`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.

Parameter

`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.

Parameter

`goodDigit` – an `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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`evaluations` – an `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.

Parameter

`iterations` – an `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.

Parameter

`evaluations` – an `int` scalar containing the maximum number of Jacobian evaluations

setMaximumStepSize

```
public void setMaximumStepSize(double stepSize)
```

Description

Sets the maximum allowable step size.

Parameter

`stepSize` – a `double` scalar containing the maximum allowable step size

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing. If `numberOfThreads` is greater than 1, then interface `Function.compute` is evaluated in parallel and `Function.compute` must be thread-safe. Otherwise, unexpected behavior can occur.

Default: `numberOfThreads` = 1.

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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`trustRegion` – a `double` scalar containing the initial trust region radius

solve

`final public void solve() throws BoundedLeastSquares.FalseConvergenceException`

Description

Solves a nonlinear least-squares problem subject to bounds on the variables using a modified Levenberg-Marquardt algorithm.

Exception

`FalseConvergenceException` is thrown when the iterates appear to be converging to a noncritical point.

Example 1: Bounded Least Squares

The nonlinear least-squares problem

$$\min \frac{1}{2} \sum_{i=0}^{1} f_i(x)^2$$

$$-2 \le x_0 \le 0.5$$

$$-1 \le x_1 \le 2$$

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

$$\min \frac{1}{2} \sum_{i=0}^{1} f_i(x)^2$$

$$-2 \le x_0 \le 0.5$$

$$-1 \le x_1 \le 2$$

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
```

BoundedLeastSquares.Function interface

`public interface com.imsl.math.BoundedLeastSquares.Function`

Public interface for the user-supplied function to evaluate the function that defines the least-squares

problem.

Method

compute

```
public 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`

BoundedLeastSquares.Jacobian interface

```
public interface com.imsl.math.BoundedLeastSquares.Jacobian
```

Public interface for the user-supplied function to compute the Jacobian.

Method

compute

```
public 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`. `fjac.length = mFunctions x nVariables`

BoundedLeastSquares.FalseConvergenceException class

```
static public class com.imsl.math.BoundedLeastSquares.FalseConvergenceException
extends com.imsl.IMSLException
```

False convergence - The iterates appear to be converging to a noncritical point.

Constructors

BoundedLeastSquares.FalseConvergenceException

```
public BoundedLeastSquares.FalseConvergenceException(String message)
```

Description

Constructs an `FalseConvergenceException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

BoundedLeastSquares.FalseConvergenceException

```
public BoundedLeastSquares.FalseConvergenceException(String key, 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

BoundedVariableLeastSquares class

```
public class com.imsl.math.BoundedVariableLeastSquares implements Serializable,
Cloneable
```

Solve a linear least-squares problem with bounds on the variables. `BoundedVariableLeastSquares` solves the least-squares problem

$$\min_x \|Ax - b\|^2$$

subject to the conditions

$$\alpha_k \leq x_k \leq \beta_k$$

for all k.

This algorithm is a generalization of `com.imsl.math.NonNegativeLeastSquares` (p. 456) , that solves the least-squares problem, $Ax = b$, subject to all $x_j \geq 0$. NonNegativeLeastSquares is based on the subroutine NNLS which appeared in Lawson and Hanson (1974). The additional work on bounded variable least squares was published in a later reprint (Lawson and Hanson, 1995).

Constructor

BoundedVariableLeastSquares

```
public BoundedVariableLeastSquares(double[][] a, double[] b, double[]
lowerBound, double[] upperBound)
```

Description

Construct a new BoundedVariableLeastSquares instance to solve *Ax-b* subject to bounds on the variables. Each upper bound must be greater than or equal to the corresponding lower bound.

Parameters

`a` – the `double` input matrix.

`b` – a `double` array of length `a.length`.

`lowerBound` – a `double` array of length n containing lower bounds. Use `Double.NEGATIVE_INFINITY` for variables which are not bounded below.

`upperBound` – a `double` array of length n containing upper bounds. Use `Double.POSITIVE_INFINITY` for variables which are not bounded above.

Methods

getDualSolution

```
public double[] getDualSolution()
```

Description

Returns the dual solution vector, w. If x_j is at neither its upper nor lower bound then $w_j = 0$. If x_j is at its lower bound then $w_j \leq 0$. If x_j is at its upper bound then $w_j \geq 0$. If the upper and lower bound for the j -th variable are equal, fixing the value of x_j, then the value of w_j is arbitrary.

Returns

a `double` array containing the dual solution vector, w.

getIterations

```
public int getIterations()
```

Description

Returns the number of iterations used to find the solution.

Returns

an `int` containing the number of iterations.

getResidualNorm

```
public double getResidualNorm()
```

Description

Returns the euclidean norm of the residual vector, $||Ax - b||^2$.

Returns

a `double` containing the euclidean norm of the residual vector.

getSolution

```
public double[] getSolution()
```

Description

Returns the solution to the problem.

Returns

a `double` array containing the solution.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Sets the maximum number of iterations.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations. The default is three times `a[0].length`.

setTolerance

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

Description

Sets the internal tolerance used to determine the relative linear dependence of a column vector for a variable moved from its initial value.

Parameter

`tolerance` – a `double` value specifying the tolerance. The default value is 1.0e-7.

solve

`public void solve() throws BoundedVariableLeastSquares.TooManyIterException`

Description

Find the solution *x* to the problem for the current constraints.

Example 1: Bounded Value Least Squares

The following example solves a linear least squares problem with bounds on the variables and compares the result to its unbounded solution. The normal of the residuals is 0.0 for the exact solution.

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

public class BoundedVariableLeastSquaresEx1 {

    public static void main(String args[]) throws Exception {
        double a[][] = {
            {1, -3, 2},
            {-3, 10, -5},
            {2, -5, 6}
        };
        double b[] = {27, -78, 64};
        double xlb[] = {-1, -1, -1};
        double xub[] = {5, 5, 5};
        double xl_ub[] = {Double.NEGATIVE_INFINITY, Double.NEGATIVE_INFINITY,
            Double.NEGATIVE_INFINITY};
        double xu_ub[] = {Double.POSITIVE_INFINITY, Double.POSITIVE_INFINITY,
            Double.POSITIVE_INFINITY};

        // compute the bounded solution
        BoundedVariableLeastSquares bvls
                = new BoundedVariableLeastSquares(a, b, xlb, xub);
        bvls.solve();
        double[] x_bounded = bvls.getSolution();
        new PrintMatrix("Bounded Solution").print(x_bounded);
        System.out.println("Norm of the Residuals = " + bvls.getResidualNorm());

        // compute the unbounded solution
        bvls = new BoundedVariableLeastSquares(a, b, xl_ub, xu_ub);
        bvls.solve();
        double[] x_unbounded = bvls.getSolution();
        new PrintMatrix("\nUnbounded Solution").print(x_unbounded);
        System.out.println("Norm of the Residuals = " + bvls.getResidualNorm());
    }
}
```

Output

```
Bounded Solution
```

```
   0
0   5
1  -1
2   5

Norm of the Residuals = 35.0142828000232

Unbounded Solution
   0
0   1
1  -4
2   7

Norm of the Residuals = 0.0
```

BoundedVariableLeastSquares.TooManyIterException class

```
static public class
com.imsl.math.BoundedVariableLeastSquares.TooManyIterException extends
com.imsl.IMSLException
```

Maximum number of iterations exceeded.

Constructors

BoundedVariableLeastSquares.TooManyIterException

```
public BoundedVariableLeastSquares.TooManyIterException(String message)
```

Description

The maximum number of iterations has exceeded.

Parameter

`message` – a `String` containing the error message

BoundedVariableLeastSquares.TooManyIterException

```
public BoundedVariableLeastSquares.TooManyIterException(String key, Object[]
arguments)
```

Description

The maximum number of iterations has exceeded.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

NonNegativeLeastSquares class

```
public class com.imsl.math.NonNegativeLeastSquares implements Serializable,
Cloneable
```

Solves a linear least squares problem with nonnegativity constraints. `NonNegativeLeastSquares` solves the problem

$$\min_x \|Ax - b\|^2$$

subject to the condition $x \geq 0$.

If a starting point x_0 is provided, those entries of x_0 that are > 0 are first combined with a descent gradient component. The start point is the origin. When x_0 is not provided the algorithm uses only the gradient to verify that an optimum has been found. The algorithm completes using only the gradient components to reach an optimum. For more information, see Lawson and Hanson (1974).

Constructor

NonNegativeLeastSquares

```
public NonNegativeLeastSquares(double[][] a, double[] b)
```

Description

Construct a new NonNegativeLeastSquares instance to solve *Ax-b* where x is a vector of n unknowns.

Parameters

`a` – the `double` input matrix.

`b` – a `double` array of length `a.length`.

Methods

getDualSolution

```
public double[] getDualSolution()
```

Description

Returns the dual solution vector, w. If $x_j = 0$ then $w_j \leq 0$, otherwise $w_j = 0$.

Returns

a `double` array containing the dual solution vector, w.

getIterations

`public int getIterations()`

Description

Returns the number of iterations used to find the solution.

Returns

an `int` containing the number of iterations.

getResidualNorm

`public double getResidualNorm()`

Description

Returns the euclidean norm of the residual vector, $||Ax - b||^2$.

Returns

a `double` containing the euclidean norm of the residual vector.

getSolution

`public double[] getSolution()`

Description

Returns the solution to the problem, x.

Returns

a `double` array containing the solution.

setDualTolerance

`public void setDualTolerance(double dualTolerance)`

Description

Sets the dual tolerance. The computation stops if largest gradient is smaller than this. The default value is 0.

Parameter

`dualTolerance` – a `double` containing the dual tolerance.

setGuess

`public void setGuess(double[] guess)`

Description

Sets the initial guess.

Parameter

`guess` – a `double` array containing the initial guess. If set, the guess is used in a two-phase algorithm where the positive components are matched with positive gradients to choose an incoming column. If not set, the algorithm uses only the gradient to verify that an optimum has been found.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Sets the maximum number of iterations.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations. The default is three times `a[0].length`.

setMaximumTime

```
public void setMaximumTime(long maximumTime)
```

Description

Sets the maximum time allowed for the solve step.

Parameter

`maximumTime` – a `long` value specifying the maximum time, in milliseconds, to be allowed for the solve step. If `maximumTime` is less than or equal to zero, then no time limit is imposed. By default, there is no time limit.

setNormTolerance

```
public void setNormTolerance(double normTolerance)
```

Description

Sets the residual norm tolerance.

Parameter

`normTolerance` – a `double` containing the residual norm tolerance. The computation stops if $||\text{rnorm}||^2 \leq \text{normTolerance} \times ||b||$, where *rnorm* is the residual norm. By default, the value is 0.

setRankTolerance

```
public void setRankTolerance(double rankTolerance)
```

Description

Sets the tolerance used for the incoming column rank deficient check.

Parameter

`rankTolerance` – a `double` value used to check for rank deficiency. The default value is 2.220e-016.

solve

```
public void solve() throws NonNegativeLeastSquares.TooManyIterException,
NonNegativeLeastSquares.TooMuchTimeException
```

Description

Finds the solution to the problem for the current constraints.

Exceptions

`TooManyIterationsException` is thrown if the maximum number of iterations is exceeded.

`TooMuchTimeException` is thrown if the maximum time allowed for solve is exceeded.

Example 1: Non-negative Least Squares

Consider the following problem:

$$\begin{bmatrix} 1 & -3 & 2 \\ -3 & 10 & -5 \\ 2 & -5 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 27 \\ -78 \\ 64 \end{bmatrix}$$

Subject to the constraint $x \geq 0$. The NonNegativeLeastSquares class is used to compute a solution, which is compared to the exact solution of $\{1, -4, 7\}$.

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

public class NonNegativeLeastSquaresEx1 {

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

        NonNegativeLeastSquares nnls = new NonNegativeLeastSquares(a, b);
        nnls.solve();
        double[] x = nnls.getSolution();

        new PrintMatrix("Solution").print(x);

        // compare solution with exact answer
        double[][] compare = new double[2][];
        compare[0] = Matrix.multiply(a, x);
        compare[1] = Matrix.multiply(a, new double[]{1, -4, 7});

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setColumnLabels(new String[]{"x >= 0", "exact"});
        PrintMatrix pm = new PrintMatrix("Comparison of 'b'");
        pm.print(pmf, Matrix.transpose(compare));
    }
}
```

Output

```
 Solution
     0
```

```
0  18.449
1   0
2   4.507

 Comparison of 'b'
   x >= 0   exact
0   27.464    27
1  -77.884   -78
2   63.942    64
```

NonNegativeLeastSquares.TooManyIterException class

`static public class com.imsl.math.NonNegativeLeastSquares.TooManyIterException extends com.imsl.IMSLException`

Maximum number of iterations has been exceeded.

Constructors

NonNegativeLeastSquares.TooManyIterException

`public NonNegativeLeastSquares.TooManyIterException(String message)`

Description

The maximum number of iterations has been exceeded.

Parameter

`message` – a `String` containing the error message

NonNegativeLeastSquares.TooManyIterException

`public NonNegativeLeastSquares.TooManyIterException(String key, Object[] arguments)`

Description

The maximum number of iterations has been exceeded.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

NonNegativeLeastSquares.TooMuchTimeException class

`static public class com.imsl.math.NonNegativeLeastSquares.TooMuchTimeException extends com.imsl.IMSLException`

Maximum time allowed for solve is exceeded.

Constructors

NonNegativeLeastSquares.TooMuchTimeException

`public NonNegativeLeastSquares.TooMuchTimeException(String message)`

Description

The maximum time allowed for solve is exceeded.

Parameter

`message` – a `String` containing the error message

NonNegativeLeastSquares.TooMuchTimeException

`public NonNegativeLeastSquares.TooMuchTimeException(String key, Object[] arguments)`

Description

The maximum time allowed for solve is exceeded.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

MinConNLP class

`public class com.imsl.math.MinConNLP implements Serializable, Cloneable`

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 `DONLP2` Users Guide 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.

The solver terminates if there is an error or if one of the following three terminations conditions is satisfied. The method `getTerminationCondition` returns the termination condition index.

- Termination condition 10: Kuhn-Tucker conditions are satisfied.

$$||g(x)^-||_1 \leq \texttt{violationBound}$$

$$||\lambda^-||_\infty \leq \texttt{multiplierError}$$

$$||\nabla L(x,\mu,\lambda)|| \leq \varepsilon_x(1+||\nabla f(x)||)$$

$$|\lambda^T g(x)| \leq \texttt{violationBound} \times \texttt{multiplierError} \times M$$

  where $L(x,\mu,\lambda) = f(x) - \lambda^T g(x)$, M is the number of constraints, and $\varepsilon_x = 10^{-5}$. The notation y^- means a vector whose negative elements are the same as the vector y, but with zeros in place of y's positive values.

- Termination condition 11: Computed correction is small.

$$d \leq \varepsilon_x(||x|| + \varepsilon_x)$$

$$||\nabla L(x,\mu,\lambda)|| \leq \varepsilon_x(1+||\nabla f(x)||)$$

$$||g(x)^-||_1 \leq \texttt{violationBound}$$

$$||\lambda^-||_\infty \leq \texttt{multiplierError}$$

  where d is the computed correction for the current solution x.

- Termination condition 12: x is almost feasible, directional derivative is very small. Further progress cannot be expected.

$$D\Phi(x;d) \geq -100(|\Phi(x)|+1)\varepsilon_x$$

  where Φ is the current penalty function, and $D\Phi$ is the directional derivative of Φ. This usually occurs as a termination condition for ill-conditioned problems.

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.

Constructor

MinConNLP

```
public MinConNLP(int mTotalConstraints, int mEqualityConstraints, int
nVariables) throws 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.

getIterations

```
public int getIterations()
```

Description

Returns the actual number of iterations used.

Returns

the number of iterations used.

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 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.

getMaximumTime

```
public long getMaximumTime()
```

Description

Returns the maximum time allowed for the solve step.

Returns

the maximum time, in milliseconds, to be allowed for the solve step. If less than or equal to zero then no time limit is imposed. The default valus is -1 (no time limit).

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the number of `java.lang.Thread` instances used for parallel processing.

Returns

an `int` containing the number of `java.lang.Thread` instances used for parallel processing.

getOptimalValue

```
public double getOptimalValue()
```

Description

Returns the value of the objective function.

Returns

a `double`, the value of the objective function.

---

getSolution

`public double[] getSolution()`

Description

Returns the last computed solution. This is the same solution as returned by the `solve` method.

Returns

a `double` array containing the solution.

---

getTerminationCriterion

`public int getTerminationCriterion()`

Description

Returns the reason the solve step terminated.

Returns

an `int` that indicates the reason the solve method terminated.

---

getTolerance

`public double getTolerance()`

Description

Returns the desired precision of the solution.

Returns

a `double` that is the the desired precision of the solution.

---

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.

Parameter

`del0` – a `double` scalar value specifying the binding threshold for constraints.

Exception

`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.

Parameter

`taubnd` – a `double` scalar value specifying the amount by which bounds may be violated during numerical differentiation.

Exception

`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.

Parameter

`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 $0.01\left(epsfcn^{1/7}\right)$

Exception

`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.

Parameter

`epsfcn` – a `double` scalar value specifying the relative precision of the function evaluation routine.

Exception

`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.

Parameter

`epsdif` – a `double` scalar value specifying the relative precision in gradients.

Exception

`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.

Parameter

`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.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed

Exception

`IllegalArgumentException` is thrown if `maxIterations` is less than or equal to 0

setMaximumTime

```
public void setMaximumTime(long maximumTime)
```

Description

Sets the maximum time allowed for the solve step.

Parameter

`maximumTime` – is the maximum time, in milliseconds, to be allowed for the solve step. If less than or equal to zero then no time limit is imposed.

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 $e^{2\log\varepsilon/3}$.

Parameter

`smallw` – a `double` scalar value specifying the error allowed in the multipliers.

Exception

`IllegalArgumentException` is thrown if smallw is less than or equal to 0.0

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing. If `numberOfThreads` is greater than 1, then interface `Function.f` is evaluated in parallel and `Function.f` must be thread-safe. Otherwise, unexpected behavior can occur.

Default: `numberOfThreads` = 1.

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.

Parameter

`tau0` – a `double` scalar value specifying the universal bound for describing how much the unscaled penalty-term may deviate from zero.

Exception

`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.

Parameter

`scbnd` – a `double` scalar value specifying the scaling variable for the problem function.

Exception

`IllegalArgumentException` is thrown if scbnd is less than or equal to 0.0

setTolerance

```
public void setTolerance(double epsx)
```

Description

Set the desired precision of the solution.

Parameter

`epsx` – is the the desired precision of the solution. For a well scaled and well-conditioned problem it essentially specifies a desired relative precision in the solution. It should never be chosen less than the square root of the machine precision since the control of progress in the method is based on the comparison of function values usually taken from the constraining manifold where the objective function varies like $O(||x^k - x^*||^2)$. Even this requirement may be too strong. The default value of 1.0e-5 is approximately the third root of the machine precision. The user should be aware of the fact that the precision requirement is automatically relaxed if the solver considers a problem "singular". If the precision seems to be too poor in such a case a decrease of `epsx` might help. Default: 1.0e-5.

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 $min(del0/10, max(epsdif, min(del0/10, max((1.e-6)del0, small_w))))$.

Parameter

`delmin` – a `double` scalar value specifying the allowable constraint violations of the final accepted result.

Exception

`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.

Parameter

`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.

Parameter

`xscale` – a `double` array specifying the internal scaling of the variables.

Exception

`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.

Parameter

`xub` – a `double` array specifying the upper bounds on the variables

solve

```
public double[] solve(MinConNLP.Function F) throws
MinConNLP.ConstraintEvaluationException,
MinConNLP.ObjectiveEvaluationException, MinConNLP.WorkingSetSingularException,
MinConNLP.QPInfeasibleException,
MinConNLP.PenaltyFunctionPointInfeasibleException,
MinConNLP.LimitingAccuracyException, MinConNLP.TooManyIterationsException,
MinConNLP.BadInitialGuessException, MinConNLP.IllConditionedException,
MinConNLP.SingularException, MinConNLP.LinearlyDependentGradientsException,
MinConNLP.NoAcceptableStepsizeException,
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.

Parameter

`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. The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

subject to

$$g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x_1^2/4 - x_2^2 + 1 \geq 0$$

is solved.

```
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.};

        MinConNLP minconnon = new MinConNLP(m, me, n);
        minconnon.setGuess(xinit);
        MinConNLPEx1 fcn = new MinConNLPEx1();
        double[] 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. The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

subject to

$$g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x_1^2/4 - x_2^2 + 1 \geq 0$$

is solved.

```
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);
        } else {
            switch (iact) {
                case 1:
                    result[0] = 1.e0;
                    result[1] = -2.e0;
                    break;
                case 2:
                    result[0] = -0.5e0 * x[0];
                    result[1] = -2.e0 * x[1];
                    break;
            }
        }
    }

    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. The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

subject to

$$g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x_1^2/4 - x_2^2 + 1 \geq 0$$

is solved.

```
import com.imsl.math.*;
import java.util.logging.Logger;
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.};

        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());
        double[] 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= 2  SI= -1


  ITSTEP= 3  FX= 1.2261822533163689   UPSI= 0.09653353175869195  B2N=3.3306690738754696E-16  UMI= 0.0  NR= 2  SI= -


  ITSTEP= 4  FX= 1.393242278445973   UPSI= 1.2061157826948055E-4  B2N=1.336885555457667E-15  UMI= 0.0  NR= 2  SI= -

     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
```

MinConNLP.Function interface

```
public interface com.imsl.math.MinConNLP.Function
```

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

Method

f

```
public 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.

MinConNLP.Gradient interface

```
public interface com.imsl.math.MinConNLP.Gradient implements
com.imsl.math.MinConNLP.Function
```

Public interface for the user supplied function to compute the gradient for `MinConNLP` object.

Method

gradient

`public 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.

MinConNLP.ConstraintEvaluationException class

```
static public class com.imsl.math.MinConNLP.ConstraintEvaluationException
extends com.imsl.IMSLException
```

Constraint evaluation returns an error with current point.

Constructors

MinConNLP.ConstraintEvaluationException

`public MinConNLP.ConstraintEvaluationException(String message)`

Description

Constructs a `ConstraintEvaluationException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.ConstraintEvaluationException

`public MinConNLP.ConstraintEvaluationException(String key, Object[] arguments)`

Description

Constructs a `ConstraintEvaluationException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.ObjectiveEvaluationException class

```
static public class com.imsl.math.MinConNLP.ObjectiveEvaluationException
extends com.imsl.IMSLException
```

Objective evaluation returns an error with current point.

Constructors

MinConNLP.ObjectiveEvaluationException

```
public MinConNLP.ObjectiveEvaluationException(String message)
```

Description

Constructs a `ObjectiveEvaluationException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.ObjectiveEvaluationException

```
public MinConNLP.ObjectiveEvaluationException(String key, Object[] arguments)
```

Description

Constructs a `ObjectiveEvaluationException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.NoAcceptableStepsizeException class

`static public class com.imsl.math.MinConNLP.NoAcceptableStepsizeException extends com.imsl.IMSLException`

No acceptable stepsize in [SIGMA,SIGLA].

Constructors

MinConNLP.NoAcceptableStepsizeException

`public MinConNLP.NoAcceptableStepsizeException(String message)`

Description

Constructs a `NoAcceptableStepsizeException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.NoAcceptableStepsizeException

`public MinConNLP.NoAcceptableStepsizeException(String key, Object[] arguments)`

Description

Constructs a `NoAcceptableStepsizeException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.WorkingSetSingularException class

`static public class com.imsl.math.MinConNLP.WorkingSetSingularException extends com.imsl.IMSLException`

Working set is singular in dual extended QP.

Constructors

MinConNLP.WorkingSetSingularException

`public MinConNLP.WorkingSetSingularException(String message)`

Description

Constructs a `WorkingSetSingularException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.WorkingSetSingularException

`public MinConNLP.WorkingSetSingularException(String key, Object[] arguments)`

Description

Constructs a `WorkingSetSingularException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.QPInfeasibleException class

`static public class com.imsl.math.MinConNLP.QPInfeasibleException extends com.imsl.IMSLException`

QP problem seemingly infeasible.

Constructors

MinConNLP.QPInfeasibleException

`public MinConNLP.QPInfeasibleException(String message)`

Description

Constructs a `QPInfeasibleException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.QPInfeasibleException

`public MinConNLP.QPInfeasibleException(String key, Object[] arguments)`

Description

Constructs a `QPInfeasibleException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.PenaltyFunctionPointInfeasibleException class

```
static public class
com.imsl.math.MinConNLP.PenaltyFunctionPointInfeasibleException extends
com.imsl.IMSLException
```

Penalty function point infeasible.

Constructors

MinConNLP.PenaltyFunctionPointInfeasibleException

```
public MinConNLP.PenaltyFunctionPointInfeasibleException(String message)
```

Description

Constructs a `PenaltyFunctionPointInfeasibleException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.PenaltyFunctionPointInfeasibleException

```
public MinConNLP.PenaltyFunctionPointInfeasibleException(String key, Object[]
arguments)
```

Description

Constructs a `PenaltyFunctionPointInfeasibleException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.LimitingAccuracyException class

`static public class com.imsl.math.MinConNLP.LimitingAccuracyException extends com.imsl.IMSLException`

Limiting accuracy reached for a singular problem.

Constructors

MinConNLP.LimitingAccuracyException

`public MinConNLP.LimitingAccuracyException(String message)`

Description

Constructs a `LimitingAccuracyException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.LimitingAccuracyException

`public MinConNLP.LimitingAccuracyException(String key, Object[] arguments)`

Description

Constructs a `LimitingAccuracyException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.TooManyIterationsException class

`static public class com.imsl.math.MinConNLP.TooManyIterationsException extends com.imsl.IMSLException`

Maximum number of iterations exceeded.

Constructors

MinConNLP.TooManyIterationsException

`public MinConNLP.TooManyIterationsException(String message)`

Description

Constructs a `TooManyIterationsException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.TooManyIterationsException

`public MinConNLP.TooManyIterationsException(String key, Object[] arguments)`

Description

Constructs a `TooManyIterationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.TooMuchTimeException class

`static public class com.imsl.math.MinConNLP.TooMuchTimeException extends com.imsl.math.MinConNLP.TooManyIterationsException`

Maximum time allowed for solve exceeded. This class extends `TooManyIterationsException` to keep the `solve` method backward compatible.

Constructor

MinConNLP.TooMuchTimeException

`public MinConNLP.TooMuchTimeException(long maximumTime)`

Description

Constructs a `TooMuchTimeException` object.

Parameter

`maximumTime` – a `long` containing the maximum allowed time

MinConNLP.BadInitialGuessException class

`static public class com.imsl.math.MinConNLP.BadInitialGuessException extends com.imsl.IMSLException`

Penalty function point infeasible for original problem. Try new initial guess.

Constructors

MinConNLP.BadInitialGuessException

`public MinConNLP.BadInitialGuessException(String message)`

Description

Constructs a `BadInitialGuessException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.BadInitialGuessException

`public MinConNLP.BadInitialGuessException(String key, Object[] arguments)`

Description

Constructs a `BadInitialGuessException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.IllConditionedException class

`static public class com.imsl.math.MinConNLP.IllConditionedException extends com.imsl.IMSLException`

Problem is singular or ill-conditioned.

Constructors

MinConNLP.IllConditionedException

```
public MinConNLP.IllConditionedException(String message)
```

Description

Constructs a `IllConditionedException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.IllConditionedException

```
public MinConNLP.IllConditionedException(String key, Object[] arguments)
```

Description

Constructs a `IllConditionedException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.SingularException class

```
static public class com.imsl.math.MinConNLP.SingularException extends
com.imsl.IMSLException
```

Problem is singular.

Constructors

MinConNLP.SingularException

```
public MinConNLP.SingularException(String message)
```

Description

Constructs a `SingularException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.SingularException

```
public MinConNLP.SingularException(String key, Object[] arguments)
```

Description

Constructs a `SingularException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.LinearlyDependentGradientsException class

```
static public class com.imsl.math.MinConNLP.LinearlyDependentGradientsException
extends com.imsl.IMSLException
```

Working set gradients are linearly dependent.

Constructors

MinConNLP.LinearlyDependentGradientsException

```
public MinConNLP.LinearlyDependentGradientsException(String message)
```

Description

Constructs a `LinearlyDependentGradientsException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.LinearlyDependentGradientsException

```
public MinConNLP.LinearlyDependentGradientsException(String key, Object[]
arguments)
```

Description

Constructs a `LinearlyDependentGradientsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.TerminationCriteriaNotSatisfiedException class

```
static public class
com.imsl.math.MinConNLP.TerminationCriteriaNotSatisfiedException extends
com.imsl.IMSLException
```

Termination criteria are not satisfied.

Constructors

MinConNLP.TerminationCriteriaNotSatisfiedException

```
public MinConNLP.TerminationCriteriaNotSatisfiedException(String message)
```

Description

Constructs a `TerminationCriteriaNotSatisfiedException` object.

Parameter

`message` – a `String` containing the error message

MinConNLP.TerminationCriteriaNotSatisfiedException

```
public MinConNLP.TerminationCriteriaNotSatisfiedException(String key, Object[]
arguments)
```

Description

Constructs a `TerminationCriteriaNotSatisfiedException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MinConNLP.Formatter class

```
static public class com.imsl.math.MinConNLP.Formatter extends
java.util.logging.Formatter
```

Simple formatter for MinConNLP logging

Constructor

MinConNLP.Formatter

```
public MinConNLP.Formatter()
```

Method

format

```
public String format(LogRecord record)
```

NumericalDerivatives class

```
public class com.imsl.math.NumericalDerivatives implements Serializable,
Cloneable
```

Compute the Jacobian matrix for a function $f(y)$ with m components in n independent variables.

`NumericalDerivatives` uses divided finite differences to compute the Jacobian. This class is designed for use in numerical methods for solving nonlinear problems where a Jacobian is evaluated repeatedly at neighboring arguments. For example this occurs in a Gauss-Newton method for solving non-linear least squares problems or a non-linear optimization method.

`NumericalDerivatives` is suited for applications where the Jacobian is a dense matrix. All cases $m < n$, $m = n$, or $m > n$ are allowed. Both one-sided and central divided differences can be used.

The design allows for computation of derivatives in a variety of contexts. Note that a gradient should be considered as the special case with $m = 1$, $n \geq 1$. A derivative of a single function of one variable is the case $m = 1$, $n = 1$. Any non-linear solving routine that optionally requests a Jacobian or gradient can use `NumericalDerivatives`. This should be considered if there are special properties or scaling issues associated with $f(y)$. Use the method `setDifferencingMethods` to specify different differencing options for numerical differentiation. These can be combined with some analytic subexpressions or other known relationships.

The divided differences are computed using values of the independent variables at the initial point $y_e = y$, and differenced points $y_e = y + del \times e_j$. Here the $e_j, j = 1, \ldots, n$, are the unit coordinate vectors. The value for each difference *del* depends on the variable *j*, the differencing method, and the scaling for that variable. This difference is computed internally. See `setPercentageFactor` for computational details. The evaluation of $f(y_e)$ is normally done by the user-provided method `NumericalDerivatives.Function.f`, using the values y_e. The index *j* and values y_e are arguments to `NumericalDerivatives.Function.f`.

The computational kernel of `evaluateJ` performs the following steps:

1. evaluate the equations at the point `y` using `NumericalDerivatives.Function.f`.
2. compute the Jacobian.
3. compute the difference at y_e.

By default, `evaluateJ` uses `NumericalDerivatives.Function.f` in step 3. The user may choose to override the `evaluateF` method to extend the capability of the class beyond the default.

There are six examples provided which illustrate various ways to use `NumericalDerivatives`. A discussion of the expected errors for these difference methods is found in *A First Course in Numerical Analysis*, Anthony Ralston, McGraw-Hill, NY, (1965).

Fields

ACCUMULATE

`static final public int ACCUMULATE`

Indicates the accumulation of the result from whatever type of differences have been specified previously into initial values of the Jacobian.

CENTRAL

`static final public int CENTRAL`

Indicates central differences.

ONE_SIDED

`static final public int ONE_SIDED`

Indicates one sided differences.

SKIP

`static final public int SKIP`

Indicates a variable to be skipped.

Constructor

NumericalDerivatives

`public NumericalDerivatives(NumericalDerivatives.Function fcn)`

Description

Constructor for `NumericalDerivatives`.

Parameter

`fcn` – a `Function` object which is a user-supplied function to evaluate the equations at the point `y`.

Methods

evaluateF

```
protected double[] evaluateF(int varIndex, double[] y)
```

Description

This method is provided by the user to compute the function values at the current independent variable values `y`. If the user does not override the `evaluateF` method, then `NumericalDerivatives.Function.f` is used to compute the function values.

Parameters

`varIndex` – an `int` which indicates the index of the variable to perturb.

`y` – a `double` array of length n, the point at which the function is to be evaluated.

Returns

a `double` array of length m. The equations evaluated at the point `y`.

evaluateJ

```
public double[][] evaluateJ(double[] y)
```

Description

Evaluates the Jacobian for a system of (m) equations in (n) variables.

Parameter

`y` – a `double` array of length n, the point at which the Jacobian is to be evaluated.

Returns

a `double` matrix containing the Jacobian. Columns that are accumulated must have the additive term defined on entry or else be set to zero. Columns that are skipped can be defined either before or after the `evaluateJ` method is invoked.

getPercentageFactor

```
public double[] getPercentageFactor()
```

Description

Returns the percentage factor for differencing.

Returns

a `double` array containing the percentage factor for differencing. See `setPercentageFactor` for more detail.

getScalingFactors

```
public double[] getScalingFactors()
```

Description

Returns the scaling factors for the `y` values.

Returns

a `double` array containing the scaling factors.

getStatus

```
public int[] getStatus()
```

Description

Returns status information. This information might prove useful to the user wanting to gain better control over the differencing parameters. This information can often be ignored.

Returns

an `int` array containing the ten diagnostic values described in the following table. These values can be used to monitor the progress or expense of the Jacobian computation.

index	**Description**
0	the number of times a function evaluation was computed.
1	the number of columns in which three attempts were made to increase a percentage factor for differencing (i.e. a component in the `factor` array) but the computed *del* remained unacceptably small relative to `y[`*j*`-1]` or `scale[`*j*`-1]`. In such cases the percentage factor is set to 1.4901161193847656e-8, which is the square root of machine precision
2	the number of columns in which the computed *del* was zero to machine precision because `y[`*j*`-1]` or `scale[`*j*`-1]` was zero. In such cases *del* is set to 1.4901161193847656e-8, which is the square root of machine precision
3	the number of Jacobian columns which had to be recomputed because the largest difference formed in the column was close to zero relative to scale, where $$scale = \max\left(\lvert f_i(y)\rvert, \lvert f_i(y + del \times e_j)\rvert\right)$$ and *i* denotes the row index of the largest difference in the column currently being processed. *index* = 9 gives the last column where this occurred.
4	the number of columns whose largest difference is close to zero relative to scale after the column has been recomputed.
5	the number of times scale information was not available for use in the roundoff and truncation error tests. This occurs when $$\min\left(\lvert f_i(y)\rvert, \lvert f_i(y + del \times e_j)\rvert\right) = 0$$ where i is the index of the largest difference for the column currently being processed.
6	the number of times the increment for differencing (*del*) was computed and had to be increased because (`scale[`*j*`-1]`+*del*) - `scale[`*j*`-1]`) was too small relative to `y[`*j*`-1]` or `scale[`*j*`-1]`.
7	the number of times a component of the `factor` array was reduced because changes in function values were large and excess truncation error was suspected. *index* = 8 gives the last column in which this occurred.
8	the index of the last column where the corresponding component of the `factor` array had to be reduced because excessive truncation error was suspected.
9	the index of the last column where the difference was small and the column had to be recomputed with an adjusted increment (see *index* = 3). The largest derivative in this column may be inaccurate due to excessive roundoff error.

setDifferencingMethods

```
public void setDifferencingMethods(int[] options)
```

Description

Sets the methods used to compute the derivatives

Parameter

`options` – an `int` array of length *n*, containing the methods used to compute the derivatives.

`options[i]` is the method to be used for the i-th variable. `options[i]` can be one of the values in the table which follows. The default is to use `ONE_SIDED` differences for each variable.

Entry	Description
`ONE_SIDED`	Indicates one sided differences.
`CENTRAL`	Indicates central differences.
`ACCUMULATE`	Indicates the accumulation of the result from whatever type of differences have been specified previously into initial values of the Jacobian.
`SKIP`	Indicates a variable to be skipped.

setInitialF

```
public void setInitialF(double[] valueF)
```

Description

Set the initial function values. Use the values $f(y_0)$, where y_0 is the initial value of the independent variables located in array `y`.

Parameter

`valueF` – a `double` array of length m containing the initial function values, y_0. Default: all values are 0.0.

setPercentageFactor

```
public void setPercentageFactor(double[] factor)
```

Description

Sets the percentage factor for differencing

For each divided difference for variable j the increment used is *del*. The value of *del* is computed as follows: First define $\sigma = sign(\text{scale}[j-1])$. If the user has set the elements of array `scale` to non-default values, then define $y_a = |\text{scale}[j-1]|$. Otherwise $y_a = |y[j-1]|$ and $\sigma = 1$. Finally compute $del = \sigma y_a\ \text{factor}[j-1]$. By changing the sign of `scale[j-1]`, the difference *del* can have any desired orientation, such as staying within bounds on variable j. For central differences, a reduced factor is used for *del* that normally results in relative errors as small as machine precision to the 2/3 power.

Parameter

`factor` – a `double` array of length n containing the percentage factor for differencing. Except for initialization, the `factor` array should not be altered in the `evaluateF` method. The elements of `factor` must be such that

$$1.8189894035458565\text{e-}12 \ \leq \text{factor}[j-1] \leq 0.1$$

where 1.8189894035458565e-12 is machine precision to the three-fourths power.
Default: all elements of `factor` are set to 1.4901161193847656e-8, which is the square root of machine precision.

setScalingFactors

```
public void setScalingFactors(double[] scale)
```

Description

Sets the scaling factors for the `y` values. The user can also use `scale` to provide appropriate signs for the increments.

Parameter

`scale` – a `double` array of length n containing the scaling factors. Default: all values are 1.0.

Example 1: One-Sided Differences

A simple use of `NumericalDerivatives` is shown. The gradient of the function $f(y_1, y_2) = a\exp(by_1) + cy_1y_2^2$ is required for values $a = 2.5e6$, $b = 3.4$, $c = 4.5$, $y_1 = 2.1$, $y_2 = 3.2$.

The numerical gradient is compared to the analytic gradient, cast as a 1 by 2 Jacobian:

$$grad(f) = \begin{bmatrix} ab\exp(by_1) + cy_2^2, & 2cy_1y_2 \end{bmatrix}$$

This analytic gradient is expected to approximately agree with the numerical differentiation gradient. Relative agreement should be approximately the square root of machine precision. That is achieved here. Generally this is the most accuracy one can expect using one-sided divided differences.

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

public class NumericalDerivativesEx1 {

    static private int m = 1, n = 2;
    static private double a, b, c;

    public static void main(String args[]) {
        double u;
        double[] y = new double[n];
        double[] scale = new double[n];
        double[][] actual = new double[m][n];
        double[] re = new double[2];

        // Define data and point of evaluation:
        a = 2.5e6;
        b = 3.4e0;
        c = 4.5e0;
        y[0] = 2.1e0;
        y[1] = 3.2e0;

        // Precision for measuring errors
        u = Math.sqrt(2.220446049250313e-016);

        // Set scaling:
        scale[0] = 1.e0;
        // Increase scale to account for large value of a.
        scale[1] = 8.e3;

        // Compute true values of partials.
        actual[0][0] = a * b * Math.exp(b * y[0]) + c * y[1] * y[1];
        actual[0][1] = 2 * c * y[0] * y[1];
```

```
        // This sets the function value used in forming one-sided
        // differences.
        NumericalDerivatives.Function fcn
                = new NumericalDerivatives.Function() {
                    public double[] f(int varIndex, double[] y) {
                        double[] tmp = new double[m];
                        tmp[0] = a * Math.exp(b * y[0])
                        + c * y[0] * y[1] * y[1];
                        return tmp;
                    }
                };

        NumericalDerivatives derv = new NumericalDerivatives(fcn);
        derv.setScalingFactors(scale);
        double[][] jacobian = derv.evaluateJ(y);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        new PrintMatrix("Numerical gradient:").print(pmf, jacobian);
        new PrintMatrix("Analytic gradient:").print(pmf, actual);

        // Check the relative accuracy of one-sided differences.
        // They should be good to about half-precision.
        jacobian[0][0] = (jacobian[0][0] - actual[0][0]) / actual[0][0];
        jacobian[0][1] = (jacobian[0][1] - actual[0][1]) / actual[0][1];
        re[0] = jacobian[0][0];
        re[1] = jacobian[0][1];

        System.out.println("Relative accuracy:");
        System.out.println("df/dy_1       df/dy_2");
        System.out.printf(" %.2fu        %.2fu\n", re[0] / u, re[1] / u);
        System.out.printf("(%.3e)  (%.3e)\n", re[0], re[1]);
    }
}
```

Output

```
     Numerical gradient:
           0            1
0  10,722,141,696.00  60.48

     Analytic gradient:
           0            1
0  10,722,141,353.42  60.48

Relative accuracy:
df/dy_1       df/dy_2
 2.14u        -0.00u
(3.195e-08)  (-1.175e-16)
```

Example 2: Skipping A Gradient Component

This example uses the same data as in the One-Sided Differences example. Here we assume that the second component of the gradient is analytically known. Therefore only the first gradient component needs numerical approximation. The input values of array `options` specify that numerical differentiation with respect to y_2 is skipped.

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

public class NumericalDerivativesEx2 {

    static private int m = 1, n = 2;
    static private double a, b, c;

    public static void main(String args[]) {
        int[] options = new int[n];
        double u;
        double[] y = new double[n];
        double[] valueF = new double[m];
        double[] scale = new double[n];
        double[][] actual = new double[m][n];
        double[] re = new double[2];

        // Define data and point of evaluation:
        a = 2.5e6;
        b = 3.4e0;
        c = 4.5e0;
        y[0] = 2.1e0;
        y[1] = 3.2e0;

        // Precision, for measuring errors
        u = Math.sqrt(2.220446049250313e-016);

        // Set scaling:
        scale[0] = 1.e0;
        // Increase scale to account for large value of a.
        scale[1] = 8.e3;

        // compute true values of partials.
        actual[0][0] = a * b * Math.exp(b * y[0]) + c * y[1] * y[1];
        actual[0][1] = 2 * c * y[0] * y[1];

        options[0] = NumericalDerivatives.ONE_SIDED;
        options[1] = NumericalDerivatives.SKIP;

        valueF[0] = a * Math.exp(b * y[0]) + c * y[0] * y[1] * y[1];

        NumericalDerivatives.Jacobian fcn
                = new NumericalDerivatives.Jacobian() {
                    public double[] f(int varIndex, double[] y) {
                        double[] tmp = new double[m];
                        tmp[0] = a * Math.exp(b * y[0]) + c * y[0] * y[1] * y[1];
                        return tmp;
                    }
```

```
                public double[][] jacobian(double[] y) {
                    double[][] tmp = new double[m][n];

            // The second component partial is skipped,
                    // since it is known analytically
                    tmp[0][1] = 2.e0 * c * y[0] * y[1];

                    return tmp;
                }
            };

        NumericalDerivatives derv = new NumericalDerivatives(fcn);
        derv.setDifferencingMethods(options);
        derv.setScalingFactors(scale);
        derv.setInitialF(valueF);
        double[][] jacobian = derv.evaluateJ(y);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        new PrintMatrix("Numerical gradient:").print(pmf, jacobian);
        new PrintMatrix("Analytic gradient:").print(pmf, actual);

        jacobian[0][0] = (jacobian[0][0] - actual[0][0]) / actual[0][0];
        jacobian[0][1] = (jacobian[0][1] - actual[0][1]) / actual[0][1];
        re[0] = jacobian[0][0];
        re[1] = jacobian[0][1];

        System.out.println("Relative accuracy:");
        System.out.println("df/dy_1       df/dy_2");
        System.out.printf(" %.2fu        %.2fu\n", re[0] / u, re[1] / u);
        System.out.printf("(%.3e)  (%.3e)\n", re[0], re[1]);
    }
}
```

Output

```
     Numerical gradient:
          0            1
0  10,722,141,696.00  60.48

     Analytic gradient:
          0            1
0  10,722,141,353.42  60.48

Relative accuracy:
df/dy_1       df/dy_2
 2.14u        0.00u
(3.195e-08)  (0.000e+00)
```

Example 3: Accumulation Of A Component

This example uses the same data as in the One-Sided Differences example. An alternate examination of the function $f(y_1, y_2) = a\exp(by_1) + cy_1y_2^2$ shows that the first term on the right-hand side need be evaluated just when computing the first partial. The additive term cy_2^2 occurs when computing the partial with respect to y_1. Also the first term does not depend on the second variable. Thus the first term can be left out of the function evaluation when computing the partial with respect to y_2, potentially avoiding cancellation errors. The input values of array `options` allow `NumericalDerivatives` to use these facts and obtain greater accuracy using a minimum number of computations of the exponential function.

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

public class NumericalDerivativesEx3 {

    static private int m = 1, n = 2;
    static private double a, b, c, f2 = 0.0;

    public static void main(String args[]) {
        int[] options = new int[n];
        double u;
        double[] y = new double[n];
        double[] valueF = new double[m];
        double[] scale = new double[n];
        double[][] actual = new double[m][n];
        double[] re = new double[2];

        // Define data and point of evaluation:
        a = 2.5e6;
        b = 3.4e0;
        c = 4.5e0;
        y[0] = 2.1e0;
        y[1] = 3.2e0;

        // Precision, for measuring errors
        u = Math.sqrt(2.220446049250313e-016);

        // Set scaling:
        scale[0] = 1.e0;
        // Increase scale to account for large value of a.
        scale[1] = 8.e3;

        // Compute true values of partials.
        actual[0][0] = a * b * Math.exp(b * y[0]) + c * y[1] * y[1];
        actual[0][1] = 2 * c * y[0] * y[1];

        options[0] = NumericalDerivatives.ACCUMULATE;
        options[1] = NumericalDerivatives.ONE_SIDED;

        valueF[0] = a * Math.exp(b * y[0]);
        scale[1] = 1.e0;

        NumericalDerivatives.Jacobian fcn
                = new NumericalDerivatives.Jacobian() {
```

```
                public double[] f(int varIndex, double[] y) {
                    double[] tmp = new double[m];

                    if (varIndex != 2) {
                        tmp[0] = a * Math.exp(b * y[0]);
                    } else {
                        // This is the function value for the partial wrt y_2.
                        tmp[0] = c * y[0] * y[1] * y[1];
                    }

                    return tmp;
                }

                public double[][] jacobian(double[] y) {
                    double[][] tmp = new double[m][n];

                    // Start with part of the derivative that is known.
                    tmp[0][0] = c * y[1] * y[1];

                    return tmp;
                }
            };

        NumericalDerivatives derv = new NumericalDerivatives(fcn);
        derv.setDifferencingMethods(options);
        derv.setScalingFactors(scale);
        derv.setInitialF(valueF);
        double[][] jacobian = derv.evaluateJ(y);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        new PrintMatrix("Numerical gradient:").print(pmf, jacobian);
        new PrintMatrix("Analytic gradient:").print(pmf, actual);

        jacobian[0][0] = (jacobian[0][0] - actual[0][0]) / actual[0][0];
        jacobian[0][1] = (jacobian[0][1] - actual[0][1]) / actual[0][1];
        re[0] = jacobian[0][0];
        re[1] = jacobian[0][1];

        System.out.println("Relative accuracy:");
        System.out.println("df/dy_1       df/dy_2");
        System.out.printf(" %.2fu        %.2fu\n", re[0] / u, re[1] / u);
        System.out.printf("(%.3e)  (%.3e)\n", re[0], re[1]);
    }
}
```

Output

```
     Numerical gradient:
           0           1
0  10,722,141,710.08  60.48
```

```
     Analytic gradient:
           0            1
0  10,722,141,353.42  60.48

Relative accuracy:
df/dy_1       df/dy_2
 2.23u        -0.51u
(3.326e-08)  (-7.569e-09)
```

Example 4: Central Differences

This example uses the same data as in the One-Sided Differences example. Agreement should be approximately the two-thirds power of machine precision. That agreement is achieved here. Generally this is the *most* accuracy one can expect using central divided differences. Note that using central differences requires essentially twice the number of evaluations of the function compared with obtaining one-sided differences. This can be a significant issue for functions that are expensive to evaluate. This example shows how to override `evaluateF`.

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

public class NumericalDerivativesEx4 extends NumericalDerivatives {

    static private int m = 1, n = 2;
    static private double a, b, c, v = 0.0;

    public NumericalDerivativesEx4(NumericalDerivatives.Function fcn) {
        super(fcn);
    }

    // Override evaluateF.
    public double[] evaluateF(int varIndex, double[] y) {
        double[] valueF = new double[m];

        valueF[0] = a * Math.exp(b * y[0]) + c * y[0] * y[1] * y[1];
        return valueF;
    }

    public static void main(String args[]) {
        int[] options = new int[n];
        double u;
        double[] y = new double[n];
        double[] scale = new double[n];
        double[][] actual = new double[m][n];
        double[] re = new double[2];

        // Define data and point of evaluation:
        a = 2.5e6;
        b = 3.4e0;
        c = 4.5e0;
        y[0] = 2.1e0;
        y[1] = 3.2e0;
```

```
        // Machine precision, for measuring errors
        u = 2.220446049250313e-016;
        v = Math.pow(3.e0 * u, 2.e0 / 3.e0);

        // Set scaling:
        scale[0] = 1.e0;
        // Increase scale to account for large value of a.
        scale[1] = 8.e3;

        // Compute true values of partials.
        actual[0][0] = a * b * Math.exp(b * y[0]) + c * y[1] * y[1];
        actual[0][1] = 2 * c * y[0] * y[1];

        options[0] = NumericalDerivatives.CENTRAL;
        options[1] = NumericalDerivatives.CENTRAL;

        // Set the increment used at the default value.
        scale[1] = 8.e3;

        NumericalDerivatives.Function fcn
                = new NumericalDerivatives.Function() {
                    public double[] f(int varIndex, double[] y) {
                        return new double[m];
                    }
                };

        NumericalDerivativesEx4 derv = new NumericalDerivativesEx4(fcn);
        derv.setDifferencingMethods(options);
        derv.setScalingFactors(scale);
        double[][] jacobian = derv.evaluateJ(y);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        new PrintMatrix("Numerical gradient:").print(pmf, jacobian);
        new PrintMatrix("Analytic gradient:").print(pmf, actual);

        // Since the function is never evaluated at the
        // initial point, hold back until the request is made.
        // Check the relative accuracy of central differences.
        // They should be good to about two thirds-precision.
        jacobian[0][0] = (jacobian[0][0] - actual[0][0]) / actual[0][0];
        jacobian[0][1] = (jacobian[0][1] - actual[0][1]) / actual[0][1];
        re[0] = jacobian[0][0];
        re[1] = jacobian[0][1];

        System.out.println("Relative accuracy:");
        System.out.println("df/dy_1       df/dy_2");
        System.out.printf(" %.2fv        %.2fv\n", re[0] / v, re[1] / v);
        System.out.printf("(%.3e)  (%.3e)\n", re[0], re[1]);
    }
}
```

Output

```
     Numerical gradient:
           0           1
0  10,722,141,354.39  60.48

     Analytic gradient:
           0           1
0  10,722,141,353.42  60.48

Relative accuracy:
df/dy_1       df/dy_2
 1.19v        0.27v
(9.100e-11)  (2.045e-11)
```

Example 5: Hessian Approximation

This example uses the same data as in the One-Sided Differences example. In this example numerical differentiation is used to approximate the Hessian matrix of $f(y_1, y_2)$. This symmetric Hessian matrix is

$$Hf = \begin{bmatrix} \frac{\partial^2 f}{\partial y_1^2} & \frac{\partial^2 f}{\partial y_1 \partial y_2} \\ \frac{\partial^2 f}{\partial y_1 \partial y_2} & \frac{\partial^2 f}{\partial y_2^2} \end{bmatrix}$$

Our method is based on casting the matrix Hf as the 2 by 2 Jacobian matrix of the gradient function. Each inner evaluation of the gradient function is itself computed using `NumericalDerivatives`. Central differences are used for both the inner and outer numerical differentiation. Because of the inherent error in both processes, the expected accuracy is about the $4/9 = (2/3)^2$ power of machine precision. Note that the approximation obtained is not symmetric. However, the difference between the off-diagonal elements provides an error estimate of that term.

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

public class NumericalDerivativesEx5 extends NumericalDerivatives {

    static private int m = 1, n = 2;
    static private double a, b, c, v = 0.0;

    class InnerNumericalDerivatives extends NumericalDerivatives {

        public InnerNumericalDerivatives(NumericalDerivatives.Function fcn) {
            super(fcn);
        }

        // Override evaluateF.
        public double[] evaluateF(int varIndex, double[] y) {
            double[] valueF = new double[m];

            valueF[0] = a * Math.exp(b * y[0]) + c * y[0] * y[1] * y[1];
```

```
            return valueF;
        }
    }

    public NumericalDerivativesEx5(NumericalDerivatives.Function fcn) {
        super(fcn);
    }

    // Override evaluateF.
    public double[] evaluateF(int varIndex, double[] y) {
        int[] iopt = new int[n];
        double[] valueF = new double[m];
        double[] scale = new double[n];
        double[] fac = new double[n];

        // This is the analytic gradient.  for comparison only.
        //        stateh(1)=a*b*exp(b*y(1))+c*y(2)**2
        //        stateh(2)=2*c*y(1)*y(2)
        //
        // Each request for a gradient evaluation uses the
        // functionality of numerical evaluation, but with
        // the same numerical code.
        iopt[0] = NumericalDerivatives.CENTRAL;
        iopt[1] = NumericalDerivatives.CENTRAL;

        // Set the increment used at the default value.
        // Set defaults for increments and scaling:
        fac[0] = 1.4901161193847656E-8;
        fac[1] = 1.4901161193847656E-8;
        // Change scale to account for large value of a.
        switch (varIndex) {
            case 1:
                scale[0] = 1.e0;
                scale[1] = 8.e8;
                break;
            case 2:
                scale[0] = 1.e4;
                scale[1] = 8.e8;
                break;
        }

        NumericalDerivatives.Function fcn
                = new NumericalDerivatives.Function() {
                    public double[] f(int varIndex, double[] y) {
                        return new double[m];
                    }
                };

        InnerNumericalDerivatives derv = new InnerNumericalDerivatives(fcn);
        derv.setPercentageFactor(fac);
        derv.setDifferencingMethods(iopt);
        derv.setScalingFactors(scale);
        derv.setInitialF(valueF);
        double[][] fjac = derv.evaluateJ(y);

        // Since the function is never evaluated at the
```

```
        // initial point, hold back until the request is made.
        // Copy gradient value into array expected by
        // outer loop computing the Hessian matrix.
        double[] tmp = new double[n];
        for (int i = 0; i < n; i++) {
            tmp[i] = fjac[0][i];
        }
        return tmp;
    }

    public static void main(String args[]) {
        int[] iopth = new int[n];
        double u;
        double[] fach = new double[n];
        double[] stateh = new double[n];
        double[] scaleh = new double[n];
        double[][] actual = new double[n][n];
        double[] y = new double[n];

        // Define data and point of evaluation:
        a = 2.5e6;
        b = 3.4e0;
        c = 4.5e0;
        y[0] = 2.1e0;
        y[1] = 3.2e0;

        // Machine precision, for measuring errors
        u = 2.220446049250313e-016;

        // Compute expected relative error using two applications
        // of central differences.
        v = Math.pow(3.e0 * u, 2.e0 / 3.e0);
        v = Math.pow(3 * v, 2. / 3.);

        // Set increments and scaling:
        fach[0] = 1.4901161193847656E-8;
        fach[1] = 1.4901161193847656E-8;
        iopth[0] = NumericalDerivatives.CENTRAL;
        iopth[1] = NumericalDerivatives.CENTRAL;

        // Compute true values of partials.
        actual[0][0] = a * b * b * Math.exp(b * y[0]);
        actual[1][0] = 2 * c * y[1];
        actual[0][1] = 2 * c * y[1];
        actual[1][1] = 2 * c * y[0];

        // Set the increment used at the default value.
        scaleh[0] = 1;
        scaleh[1] = 8.e5;

        NumericalDerivatives.Function fcn
                = new NumericalDerivatives.Function() {
                    public double[] f(int varIndex, double[] y) {
                        return new double[n];
                    }
                };
```

```
        NumericalDerivativesEx5 derv2 = new NumericalDerivativesEx5(fcn);
        derv2.setPercentageFactor(fach);
        derv2.setDifferencingMethods(iopth);
        derv2.setScalingFactors(scaleh);
        derv2.setInitialF(stateh);
        double[][] h = derv2.evaluateJ(y);

        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(2);
        nf.setMinimumFractionDigits(2);

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(nf);
        new PrintMatrix("Numerical Hessian:").print(pmf, h);
        new PrintMatrix("Analytic Hessian:").print(pmf, actual);

        // Since the function is never evaluated at the
        // initial point, hold back until the request is made.
        // Subtract the actual hessian matrix values and check.
        h[0][0] = (h[0][0] - actual[0][0]) / h[0][0] / v;
        h[1][0] = (h[1][0] - actual[1][0]) / h[1][0] / v;
        h[0][1] = (h[0][1] - actual[0][1]) / h[0][1] / v;
        h[1][1] = (h[1][1] - actual[1][1]) / h[1][1] / v;

        new PrintMatrix("Hessian Matrix, Expected Normalized "
                + "Relative Error, |all entries|").print(h);
    }
}
```

Output

```
     Numerical Hessian:
           0           1
0  36,455,292,905.82  28.80
1              28.80  18.90

      Analytic Hessian:
           0           1
0  36,455,280,444.94  28.80
1              28.80  18.90

Hessian Matrix, Expected Normalized Relative Error, |all entries|
     0      1
0  0.914  0.037
1  0.036  0
```

Example 6: Usage With Class `MinUnconMultiVar`

The minimum of $100(x_2 - x_1^2)^2 + (1 - x_1)^2$ is found using `MinUnconMultiVar`. `NumericalDerivatives` is used to compute the numerical gradients.

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

public class NumericalDerivativesEx6 {

    static private int m = 1, n = 2;

    static double fcnEvaluation(double[] x) {
        return 100. * ((x[1] - x[0] * x[0]) * (x[1] - x[0] * x[0]))
                + (1. - x[0]) * (1. - x[0]);
    }

    static class MyFunction implements MinUnconMultiVar.Gradient {

        public double f(double[] x) {
            return fcnEvaluation(x);
        }

        public void gradient(double[] x, double[] gp) {
            NumericalDerivatives.Function fcn
                    = new NumericalDerivatives.Function() {
                        public double[] f(int varIndex, double[] y) {
                            double[] tmp = new double[m];
                            tmp[0] = fcnEvaluation(y);
                            return tmp;
                        }
                    };

            NumericalDerivatives nderv = new NumericalDerivatives(fcn);
            double[][] jacobian = nderv.evaluateJ(x);

            gp[0] = jacobian[0][0];
            gp[1] = jacobian[0][1];
        }
    }

    public static void main(String args[]) throws Exception {
        MinUnconMultiVar solver = new MinUnconMultiVar(n);
        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 (0.9999986118580241, 0.9999972746481575)
```

NumericalDerivatives.Function interface

```
public interface com.imsl.math.NumericalDerivatives.Function
```

Public interface function.

Method

f

```
public double[] f(int varIndex, double[] y)
```

Description

Returns the equations evaluated at the point `y`. If the user does not override the `evaluateF` method, then `f` is also used to compute the function values at the current independent variable values y_e.

Parameters

`varIndex` – an `int` indicating the index of the variable to perturb. `varIndex` = 1 indicates variable 1 in `y[0]`.

`y` – a `double` array of length n, the point at which the Jacobian is to be evaluated.

Returns

a `double` array of length m. The equations evaluated at the point `y`.

NumericalDerivatives.Jacobian interface

```
public interface com.imsl.math.NumericalDerivatives.Jacobian implements
com.imsl.math.NumericalDerivatives.Function
```

Public interface for the user-supplied function to compute the Jacobian.

Method

jacobian

```
public double[][] jacobian(double[] y)
```

Description

User-supplied function to compute the Jacobian.

Parameter

`y` – a `double` array of length n, the point at which the Jacobian is to be evaluated.

Returns

a `double` m by n matrix containing the Jacobian. Columns that are accumulated must have the analytic part defined on entry or else be set to zero. Columns that are skipped can be defined either before or after the `evaluateJ` method is invoked.

Chapter 10: Special Functions

Types

class Sfun 511
class Bessel 528
class JMath 534
class IEEE 542
class Hyperbolic 544

Sfun class

`public class com.imsl.math.Sfun`

Collection of special functions.

Fields

EPSILON_LARGE

`static final public double EPSILON_LARGE`

The largest relative spacing for doubles.

EPSILON_SMALL

`static final public double EPSILON_SMALL`

The smallest relative spacing for doubles.

Methods

beta

```
static public 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 `com.imsl.math.Sfun.gamma` (p. 521) 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

```
static public 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 `com.imsl.math.Sfun.beta` (p. 512) 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

```
static public double cot(double x)
```

Description

Returns the cotangent of a `double`.

Parameter

`x` – a `double` value

Returns

a `double` value specifying the cotangent of `x`. If `x` is NaN, the result is NaN.

erf

```
static public double erf(double x)
```

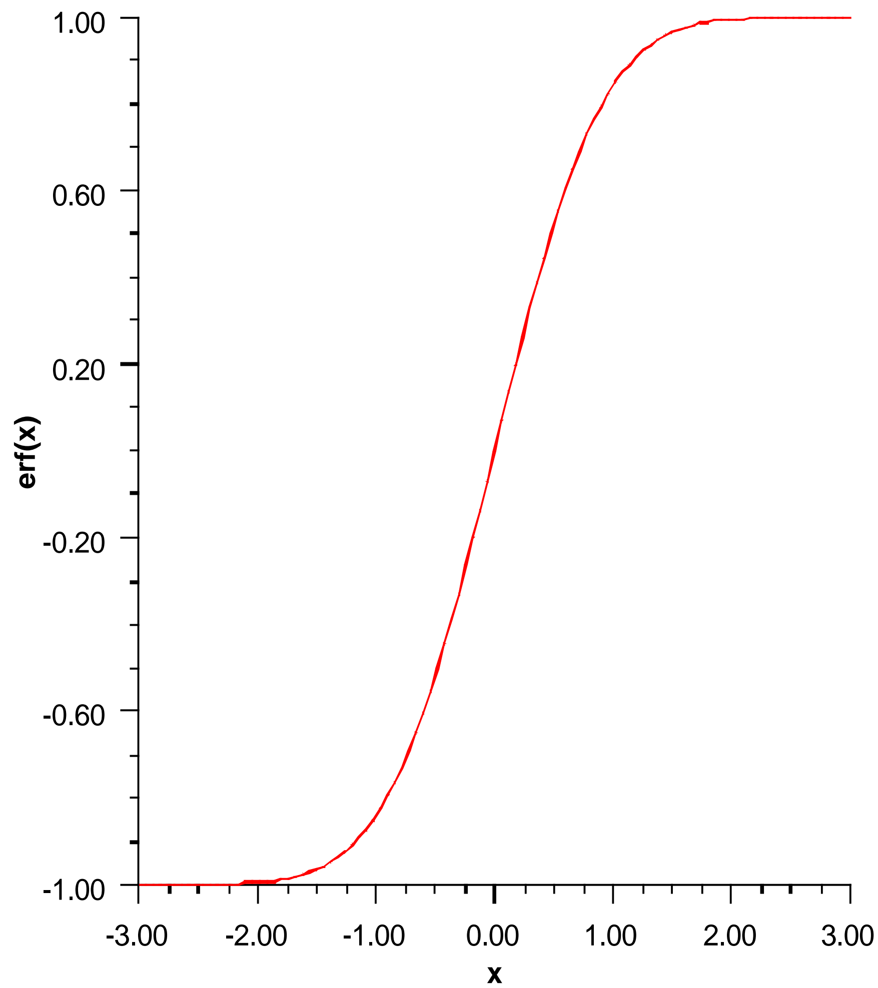
Description

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.



Parameter

x – a double value

Returns

a double value specifying the error function of x

erfInverse

```
static public double erfInverse(double x)
```

Description

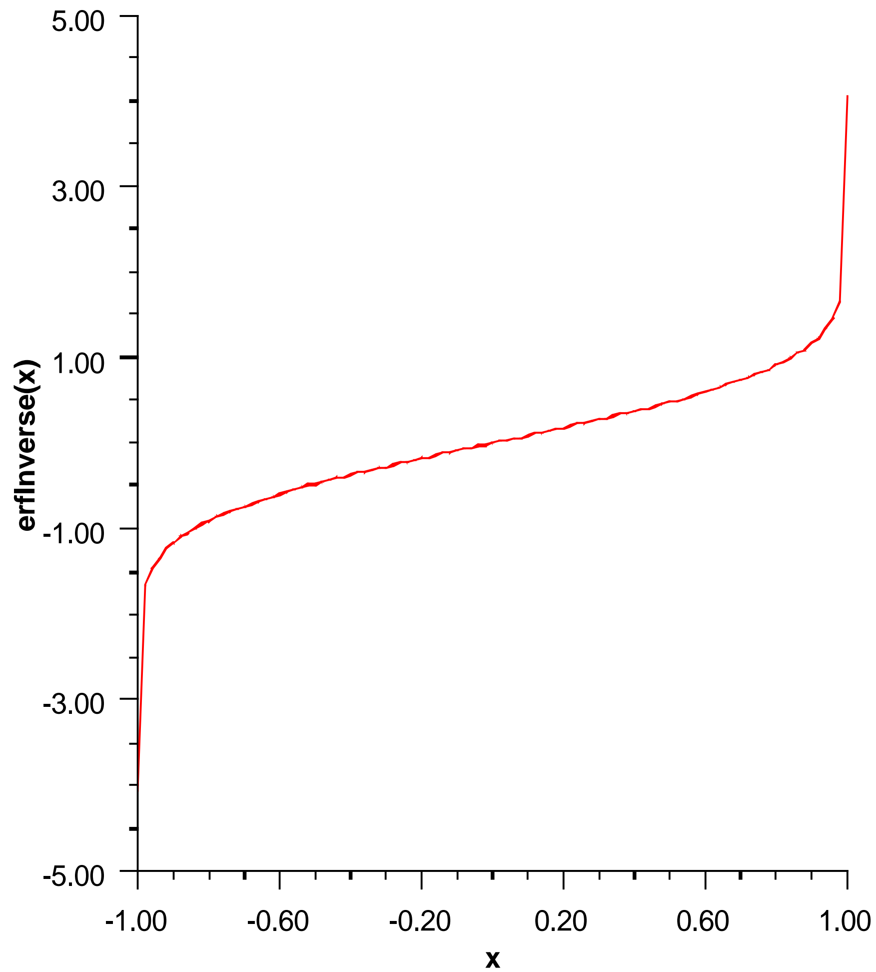
Returns the inverse of the error function.

The `erfInverse` method computes the inverse of the error function erf x, defined in `com.imsl.math.Sfun.erf` (p. 513) .

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 ε is the machine precision (approximately 1.11e-16).



Parameter

x – a double value

Returns

a double value specifying the inverse of the error function of x.

erfc

```
static public double erfc(double x)
```

Description

Returns the complementary error function of a `double`.

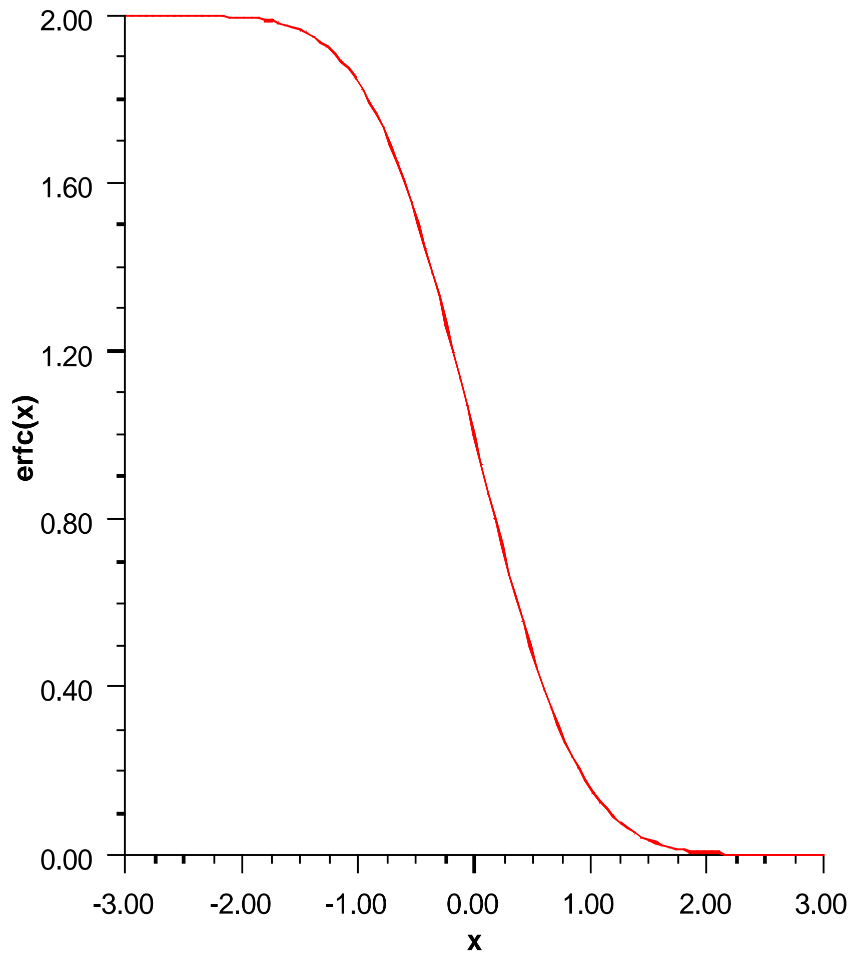
The complementary error function method, `erfc` (x), is defined to be

$$\mathrm{erfc}\,(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

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 $s =$ `java.lang.Double.MIN_VALUE` is the smallest representable positive floating-point number.



Parameter

x – a double value

Returns

a double value specifying the complementary error function of x

erfcInverse

```
static public double erfcInverse(double x)
```

Description

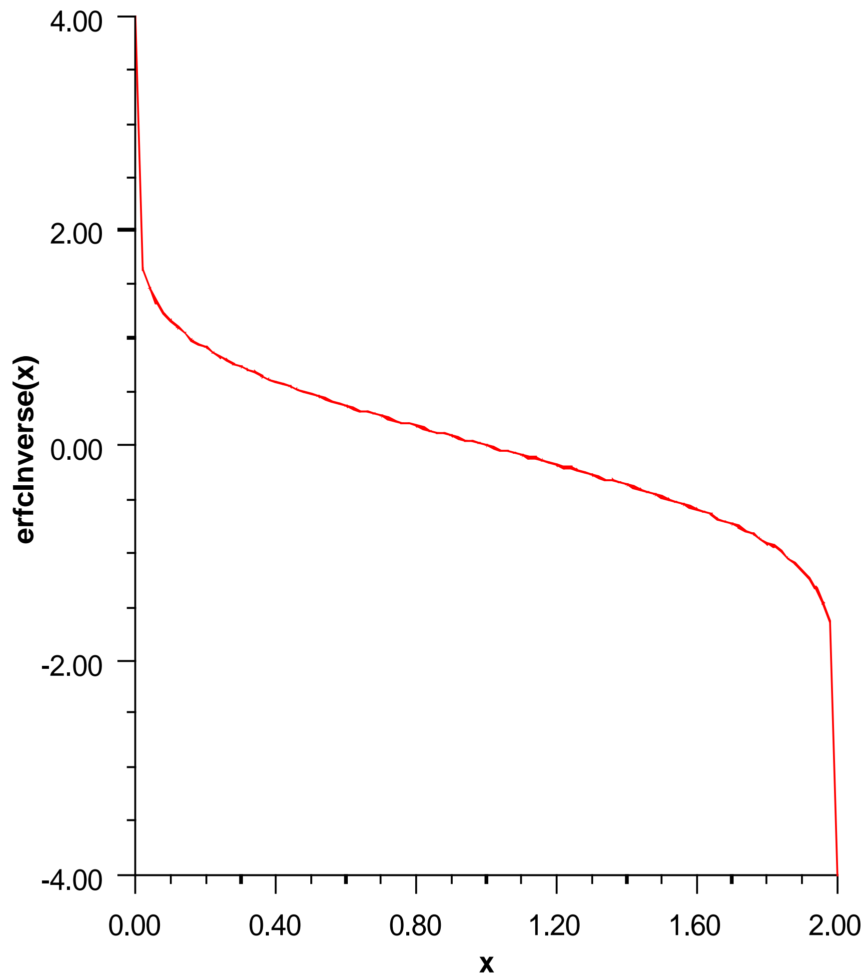
Returns the inverse of the complementary error function.

The `erfcinverse` 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 ε = machine precision (approximately 1.11e-16).



Parameter

x – a `double` value, $0 \leq x \leq 2$.

Returns

a `double` value specifying the inverse of the error function of x.

erfce

```
static public double erfce(double x)
```

Description

Returns the exponentially scaled complementary error function.

The exponentially scaled complementary error function is defined as

$$e^{x^2}\operatorname{erfc}(x)$$

where erfc(x) is the complementary error function. See `com.imsl.math.Sfun.erfc` (p. 517) for its definition.

To prevent the answer from underflowing, x must be greater than

$$x_{\min} \simeq -\sqrt{\ln(b/2)} = -26.618735713751487$$

where b = `java.lang.Double.MAX_VALUE` is the largest representable double precision number.

Parameter

`x` – a `double` value for which the function value is desired.

Returns

a `double` value specifying the exponentially scaled complementary error function of `x`.

fact

```
static public double fact(int n)
```

Description

Returns the factorial of an integer.

Parameter

`n` – an `int` value

Returns

a `double` value specifying the factorial of n, n!. If n is negative, the result is NaN.

gamma

```
static public double gamma(double x)
```

Description

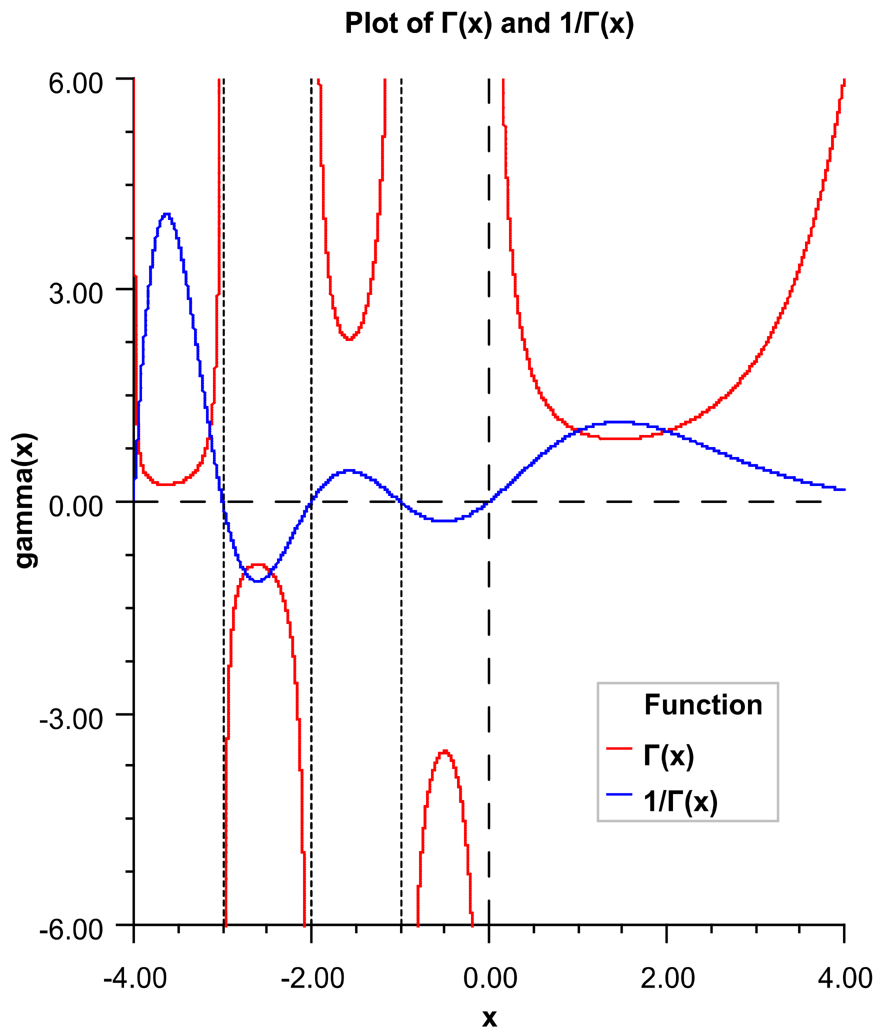
Returns the Gamma function of a `double`.

The gamma function, $\Gamma(x)$, is defined to be

$$\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt \;\; for x > 0$$

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.



Parameter

x – a `double` value

Returns

a `double` value specifying the Gamma function of `x`. If `x` is a negative integer, the result is NaN.

gammaIncomplete

```
static public double gammaIncomplete(double a, double x)
```

Description

Evaluates the incomplete gamma function.

The lower limit of integration of the incomplete gamma function, $\gamma(a,x)$, is defined to be

$$\gamma(a,x) = \int_0^x t^{a-1} e^{-t} dt \quad \text{for } x \geq 0 \text{ and } a > 0$$

Although $\gamma(a,x)$ is well defined for $x > -\infty$, this algorithm does not calculate $\gamma(a,x)$ for negative x. For large a and sufficiently large x, $\gamma(a,x)$ may overflow. $\gamma(a,x)$ is bounded by $\Gamma(a)$, and users may find this bound a useful guide in determining legal values for a.

Note that the upper limit of integration of the incomplete gamma, $\Gamma(a,x)$, is defined to be

$$\Gamma(a,x) = \int_x^\infty t^{a-1} e^{-t} dt$$

Therefore, by definition, the two incomplete gamma function forms satisfy the relationship

$$\Gamma(a,x) + \gamma(a,x) = \Gamma(a)$$

Parameters

a – a `double` value representing the integrand exponent parameter of the incomplete gamma function. It must be positive.

x – a `double` value specifying the point at which the incomplete gamma function is to be evaluated. It must be nonnegative.

Returns

a `double` value specifying the incomplete gamma function.

log10

```
static public double log10(double x)
```

Description

Returns the common (base 10) logarithm of a `double`.

Parameter

x – a `double` value

Returns

a `double` value specifying the common logarithm of `x`.

logBeta

```
static public 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 `com.imsl.math.Sfun.beta` (p. 512) 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

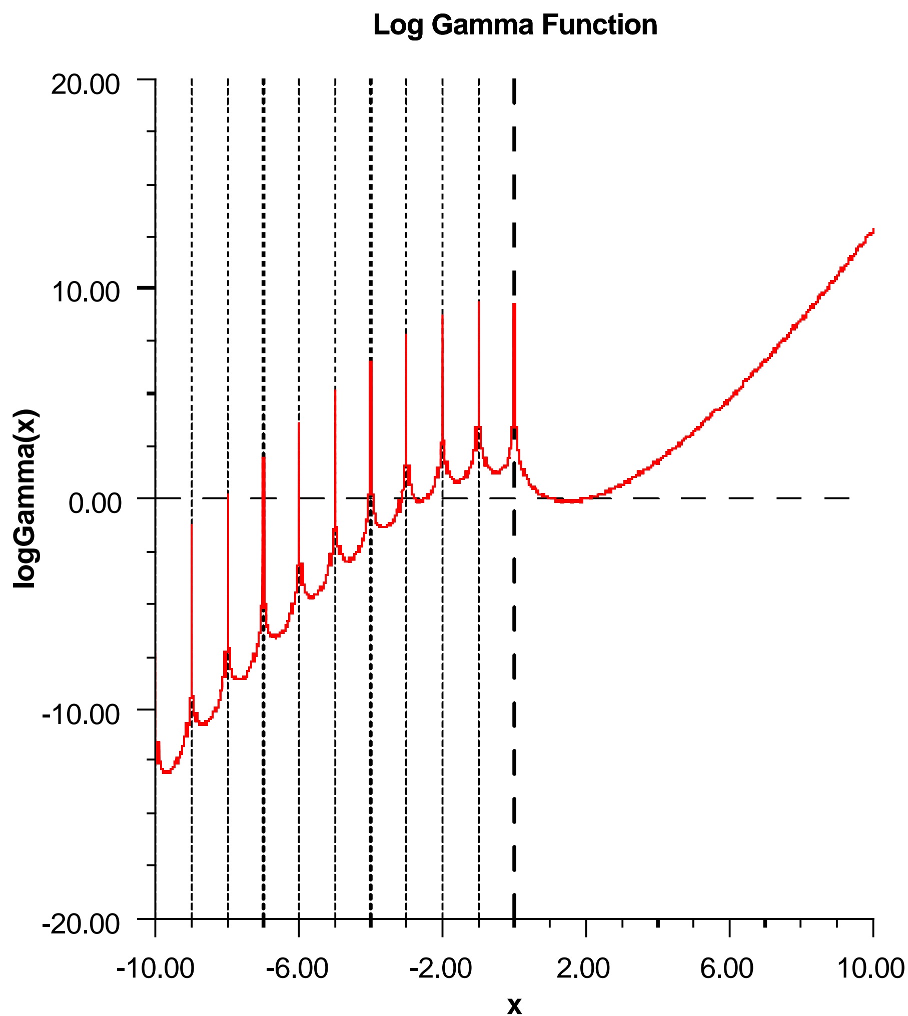
```
static public double logGamma(double x)
```

Description

Returns the logarithm of the absolute value of the Gamma function.

Method `logGamma` computes $\ln|\Gamma(x)|$. See `com.imsl.math.Sfun.gamma` (p. 521) 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.



Parameter

x – a double value

Returns

a double, the natural logarithm of the Gamma function of x. If x is a negative integer, the result is NaN.

poch

```
static public double poch(double a, double x)
```

Description

Returns a generalization of Pochhammer's symbol.

Method `poch` evaluates Pochhammer's symbol $(a)_n = (a)(a-1)\dots(a-n+1)$ for n a nonnegative integer. Pochhammer's generalized symbol is defined to be

$$(a)_x = \frac{\Gamma(a+x)}{\Gamma(a)}$$

See `com.imsl.math.Sfun.gamma` (p. 521) 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, $\frac{\Gamma(a+x)}{\Gamma(a)}$

psi

```
static public double psi(double x)
```

Description

Returns the derivative of the log gamma function, also called the digamma function.

The psi function is defined to be

$$\psi(x) = \frac{d}{dx} \ln \Gamma(x) = \frac{\Gamma'(x)}{\Gamma(x)}$$

See `com.imsl.math.Sfun.gamma` (p. 521) for the definition of $\Gamma(x)$.

The argument x must not be exactly zero or a negative integer, or $\psi(x)$ is undefined. Also, x must not be too close to a negative integer such that the accuracy of the result is less than half precision.

Parameter

`x` – a `double` value, the point at which the digamma function is to be evaluated.

Returns

a `double` value specifying the logarithmic derivative of the gamma function of x. If x is a zero or a negative integer, the result is NaN. If x is too close to a negative integer the accuracy of the result will be less than half precision.

psi1

```
static public double psi1(double x)
```

Description

Returns the ψ_1 function, also known as the trigamma function.

The trigamma function, $\psi_1(x)$, is defined to be

$$\psi_1(x) = \frac{d^2}{dx^2} \ln \Gamma(x)$$

The trigamma function is not defined for integers less than or equal to zero.

Parameter

x – a double value, the point at which the trigamma function is to be evaluated.

Returns

a double value specifying the trigamma function of x. If x is a negative integer or zero, the result is NaN.

sign

```
static public 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);
    }
}
```

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
```

Bessel class

`public class com.imsl.math.Bessel`

Collection of Bessel functions.

Methods

I

```
static public 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 $|x| \le \log(b)$ 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.

I

```
static public 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 $I_\nu(x)$, is defined to be

$$I_\nu(x) = \frac{1}{\pi}\int_0^{\pi} e^{x\cos\theta}\cos(\nu\theta)d\theta - \frac{\sin(\nu\pi)}{\pi}\int_0^{\infty} e^{-x\cosh t - \nu t}dt$$

Here, argument xnu is represented by ν 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 ν = xnu must satisfy $0 \le \nu \le 1$.

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`.

J

```
static public 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

$$J_n(x) = \frac{1}{\pi}\int_0^{\pi} \cos(x \sin\theta - n\theta)\, d\theta$$

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`.

J

```
static public 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 $J_\nu(x)$, 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`.

K

```
static public 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 $K_\nu(x)$, 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

K

```
static public 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 $K_\nu(x)$ is defined to be

$$K_\nu(x) = \frac{\pi}{2} e^{\nu\pi i/2} \left[i J_\nu(ix) - Y_\nu(ix) \right] \text{ for} - \pi < \arg x \leq \frac{\pi}{2}$$

Currently, `xnu` (represented by ν 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`$[0] = K_\nu(x)$, `K`$[1] = K_{\nu+1}(x)$, ..., `K`$[n-1] = K_{\nu+n-1}(x)$.

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`.

Y

```
static public 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 ν in the above equation) must satisfy $0 \leq \nu < 1$. 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`.

scaledK

```
static public 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 $e^x K_{\nu+i-1}(x)$, 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.

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.printf("  %d   %10.4f %10.4f  %10.4f\n",
                    i, bi[i], bj[i], bk[i]);
        }
    }
}
```

Output

```
Order  Bessel.I     Bessel.J     Bessel.K
  0    2815.7166    -0.2459      0.0000
  1    2670.9883     0.0435      0.0000
  2    2281.5190     0.2546      0.0000
  3    1758.3807     0.0584      0.0000
```

JMath class

```
public final class com.imsl.math.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.

Fields

E

```
static final public double E
```

PI

```
static final public double PI
```

Methods

IEEEremainder

```
static public 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.

abs

```
static public double abs(double x)
```

Description

Returns the absolute value of a `double`.

Parameter

x – a `double`

Returns

a `double` representing $|x|$.

---

abs

`static public float abs(float x)`

Description

Returns the absolute value of a `float`.

Parameter

`x` – a `float`

Returns

a `float` representing $|x|$.

---

abs

`static public int abs(int x)`

Description

Returns the absolute value of an `int`.

Parameter

`x` – an `int`

Returns

an `int` representing $|x|$.

---

abs

`static public long abs(long x)`

Description

Returns the absolute value of a `long`.

Parameter

`x` – a `long`

Returns

a `long` representing $|x|$.

---

acos

`static public double acos(double x)`

Description

Returns the inverse (arc) cosine of a `double`.

Parameter

`x` – a `double`

Returns

a `double` representing the angle, in radians, whose cosine is x. It is in the range $[0, \pi]$.

---

asin

```
static public double asin(double x)
```

Description

Returns the inverse (arc) sine of a `double`.

Parameter

`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

```
static public double atan(double x)
```

Description

Returns the inverse (arc) tangent of a `double`.

Parameter

`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

```
static public 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

```
static public double ceil(double x)
```

Description

Returns the value of a `double` rounded toward positive infinity to an integral value.

Parameter

x – a double

Returns

the smallest double, not less than x, that is an integral value

---

cos

```
static public double cos(double x)
```

Description

Returns the cosine of a double.

Parameter

x – a double, assumed to be in radians

Returns

a double, the cosine of x

---

exp

```
static public double exp(double x)
```

Description

Returns the exponential of a double. Special cases: e^{∞} is ∞, e^{NaN} is NaN; $e^{-\infty}$ is 0, and for finite argument, only $e^0 = 1$ is exact.

Parameter

x – a double.

Returns

a double representing e^x.

---

floor

```
static public double floor(double x)
```

Description

Returns the value of a double rounded toward negative infinity to an integral value.

Parameter

x – a double

Returns

the smallest double, not greater than x, that is an integral value

---

log

```
static public double log(double x)
```

Description

Returns the natural logarithm of a double.

Parameter

x – a `double`

Returns

a `double` representing the natural (base e) logarithm of x

---

max

```
static public 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

```
static public 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

```
static public 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

```
static public long max(long x, long y)
```

Description

Returns the larger of two `longs`.

Parameters

`x` – a `long`

`y` – a `long`

Returns

a `long`, the larger of x and y

---

min

```
static public double min(double x, double y)
```

Description

Returns the smaller of two `doubles`.

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

```
static public float min(float x, float y)
```

Description

Returns the smaller of two `floats`.

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

```
static public int min(int x, int y)
```

Description

Returns the smaller of two `ints`.

Parameters

`x` – an `int`

`y` – an `int`

Returns

an `int` representing the smaller of x and y

min

```
static public 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

```
static public 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

```
static public double random()
```

Description

Returns a random number from a uniform distribution.

Returns

a `double` representing a random number from a uniform distribution

rint

```
static public double rint(double x)
```

Description

Returns the value of a `double` rounded toward the closest integral value.

Parameter

x – a `double`

Returns

the `double` closest to x that is an integral value

round

```
static public long round(double x)
```

Description

Returns the `long` closest to a given `double`.

Parameter

x – a `double`

Returns

the `long` closest to x

round

```
static public int round(float x)
```

Description

Returns the integer closest to a given `float`.

Parameter

x – a `float`

Returns

the `int` closest to x

sin

```
static public double sin(double x)
```

Description

Returns the sine of a `double`.

Parameter

x – a `double`, assumed to be in radians

Returns

a `double`, the sine of x

sqrt

```
static public double sqrt(double x)
```

Description

Returns the square root of a `double`.

Parameter

x – a `double`

Returns

a `double` representing the square root of x

tan

`static public double tan(double x)`

Description

Returns the tangent of a `double`.

Parameter

x – a `double`, assumed to be in radians

Returns

a `double`, the tangent of x

IEEE class

`public class com.imsl.math.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.

Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
Developed at SunSoft, a Sun Microsystems, Inc. business. Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

Methods

copysign

`static public 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

```
static public boolean finite(double x)
```

Description

Finite number test on an argument of type double.

Parameter

`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

```
static public int ilogb(double x)
```

Description

Return the binary exponent of non-zero x.

Parameter

`x` – a `double`

Returns

an `int` representing the binary exponent of x. Special cases `ilogb(0) = -Integer.MAX_VALUE` and $\mathrm{ilogb}(\infty) = \mathrm{ilogb}(-\infty) = \mathrm{ilogb}(NaN) =$ Integer.MAX_VALUE.

isNaN

```
static public boolean isNaN(double x)
```

Description

NaN test on an argument of type double.

Parameter

`x` – the `double` which is to be tested

Returns

true if x is a NaN, false otherwise

nextAfter

```
static public 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

```
static public 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

```
static public 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

Hyperbolic class

```
public class com.imsl.math.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.

Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.

Developed at SunSoft, a Sun Microsystems, Inc. business. Permission to use, copy, modify, and distribute this software is freely granted, provided that this notice is preserved.

Methods

acosh

`static public 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 $|x| < 1$.

Parameter

`x` – a `double` value representing the argument.

Returns

a `double` value representing the number whose hyperbolic cosine is x.

asinh

`static public double asinh(double x)`

Description

Returns the inverse hyperbolic sine of its argument. Specifically,
asinh(± 0) returns $\pm\infty$
asinh($\pm\infty$) returns $\pm\infty$

Parameter

`x` – a `double` value representing the argument.

Returns

a `double` value representing the number whose hyperbolic sine is x.

atanh

`static public double atanh(double x)`

Description

Returns the inverse hyperbolic tangent of its argument. Specifically,
atanh(± 0) returns ± 0
atanh(± 1) returns $+\infty$
atanh(x) returns NaN, if $|x| > 1$.

Parameter

x – a double value representing the argument.

Returns

a double value representing the number whose hyperbolic tangent is x.

cosh

```
static public double cosh(double x)
```

Description

Returns the hyperbolic cosine of its argument. Specifically,
$\cosh(\pm 0)$ returns 1.
$\cosh(\pm\infty)$ returns $+\infty$

Parameter

x – a double value representing the argument.

Returns

a double value representing the hyperbolic cosine of x.

expm1

```
static public double expm1(double x)
```

Description

Returns exp(x)-1, the exponential of x minus 1. Specifically,
expm1(± 0) returns ± 0
expm1($+\infty$) returns $\pm\infty$
expm1($-\infty$) returns -1.

Parameter

x – a double specifying the argument.

Returns

a double value representing exp(x)-1.

log1p

```
static public double log1p(double x)
```

Description

Returns log(1+x), the logarithm of (x plus 1). Specifically,
log1p(± 0) returns ± 0
log1p(-1) returns $-\infty$
log1p(x) returns NaN, if $x < -1$.
log1p($\pm\infty$) returns $\pm\infty$

Parameter

`x` – a `double` value representing the argument.

Returns

a `double` value representing log(1+x).

---

sinh

`static public double sinh(double x)`

Description

Returns the hyperbolic sine of its argument. Specifically,
$\sinh(\pm 0)$ returns ± 0
$\sinh(\pm\infty)$ returns $\pm\infty$

Parameter

`x` – a `double` value representing the argument.

Returns

a `double` value representing the hyperbolic sine of x.

---

tanh

`static public double tanh(double x)`

Description

Returns the hyperbolic tangent of its argument. Specifically,
$\tanh(\pm 0)$ returns ± 0
$\tanh(\pm\infty)$ returns ± 1.

Parameter

`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(0.0));
        System.out.println("cosh(0.) is " + Hyperbolic.cosh(0.0));
```

```
        System.out.println("tanh(0.) is " + Hyperbolic.tanh(0.0));
        System.out.println("asinh(0.) is " + Hyperbolic.asinh(0.0));
        System.out.println("acosh(0.) is " + Hyperbolic.acosh(0.0));
        System.out.println("atanh(0.) is " + Hyperbolic.atanh(0.0));
    }
}
```

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 11: Miscellaneous

Types

class Complex 549
class Physical 569
class EpsilonAlgorithm 580

Complex class

`public class com.imsl.math.Complex extends java.lang.Number implements Serializable, Cloneable`

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.

Fields

i

`static final public Complex i`

The imaginary unit. This constant is set to new Complex(0,1).

suffix

`static public 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.

Parameter

`z` – a `Complex` object

Exception

`NullPointerException` is thrown if `z` is null

Complex

`public Complex(double re)`

Description

Constructs a `Complex` with a zero imaginary part.

Parameter

`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

```
static public double abs(Complex z)
```

Description

Returns the absolute value (modulus) of a `Complex`, $|z|$.

Parameter

`z` – a `Complex` object

Returns

a `double` value equal to the absolute value of the argument

acos

```
static public 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 $\pi/2 + i\mathrm{NaN}$.
$\mathrm{acos}(\mathrm{NaN} + i\infty)$ returns $\mathrm{NaN} - i\infty$.
$\mathrm{acos}(x + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for nonzero finite x.
$\mathrm{acos}(\mathrm{NaN} + iy)$ returns $\mathrm{NaN} + i\mathrm{NaN}$, for finite y.
$\mathrm{acos}(\mathrm{NaN} + i\mathrm{NaN})$ returns $\mathrm{NaN} + i\mathrm{NaN}$.

Parameter

`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

```
static public 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 $z = x+iy$,
$\text{acosh}(\bar{z}) = \overline{\text{acosh}(z)}$.
$\text{acosh}(\pm 0 + i0)$ returns $+0 + i\pi/2$.
$\text{acosh}(-\infty + i\infty)$ returns $+\infty + i3\pi/4$.
$\text{acosh}(+\infty + i\infty)$ returns $+\infty + i\pi/4$.
$\text{acosh}(x + i\infty)$ returns $+\infty + i\pi/2$, for finite x.
$\text{acosh}(-\infty + iy)$ returns $+\infty + i\pi$, for positive-signed finite y.
$\text{acosh}(+\infty + iy)$ returns $+\infty + i0$, for positive-signed finite y.
$\text{acosh}(\text{NaN} + i\infty)$ returns $+\infty + i\text{NaN}$.
$\text{acosh}(\pm\infty + i\text{NaN})$ returns $+\infty + i\text{NaN}$.
$\text{acosh}(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$, for finite x.
$\text{acosh}(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$, for finite y.
$\text{acosh}(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

Parameter

`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

```
static public 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

```
static public 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

```
static public 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

```
static public double argument(Complex z)
```

Description

Returns the argument (phase) of a `Complex`, in radians, with a branch cut along the negative real axis.

Parameter

`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

```
static public 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 $\text{asin}(z) = -i\,\text{asinh}(iz)$.

Parameter

`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

```
static public 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 $z = x+iy$,
$\text{asinh}(\bar{z}) = \overline{\text{asinh}(z)}$ and asinh is odd.
$\text{asinh}(+0+i0)$ returns $0+i0$.
$\text{asinh}(\infty+i\infty)$ returns $+\infty+i\pi/4$.
$\text{asinh}(x+i\infty)$ returns $+\infty+i\pi/2$ for positive-signed finite x.
$\text{asinh}(+\infty+iy)$ returns $+\infty+i0$ for positive-signed finite y.
$\text{asinh}(\text{NaN}+i\infty)$ returns $\pm\infty+i\text{NaN}$ (where the sign of the real part of the result is unspecified).
$\text{asinh}(+\infty+i\text{NaN})$ returns $+\infty+i\text{NaN}$.
$\text{asinh}(\text{NaN}+i0)$ returns $\text{NaN}+i0$.
$\text{asinh}(\text{NaN}+iy)$ returns $\text{NaN}+i\text{NaN}$, for finite nonzero y.
$\text{asinh}(x+i\text{NaN})$ returns $\text{NaN}+i\text{NaN}$, for finite x.
$\text{asinh}(\text{NaN}+i\text{NaN})$ returns $\text{NaN}+i\text{NaN}$.

Parameter

`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

```
static public 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 $\text{atan}(z) = -i\,\text{atanh}(iz)$.

Parameter

`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

```
static public 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 $z = x+iy$,
$\mathrm{atanh}(\bar{z}) = \overline{\mathrm{atanh}(z)}$ and atanh is odd.
$\mathrm{atanh}(+0+i0)$ returns $+0+i0$.
$\mathrm{atanh}(+\infty+i\infty)$ returns $+0+i\pi/2$.
$\mathrm{atanh}(+\infty+iy)$ returns $+0+i\pi/2$, for finite positive-signed y.
$\mathrm{atanh}(x+i\infty)$ returns $+0+i\pi/2$, for finite positive-signed x.
$\mathrm{atanh}(+0+i\mathrm{NaN})$ returns $+0+i\mathrm{NaN}$.
$\mathrm{atanh}(\mathrm{NaN}+i\infty)$ returns $\pm 0+ipi/2$ (where the sign of the real part of the result is unspecified).
$\mathrm{atanh}(+\infty+i\mathrm{NaN})$ returns $+0+i\mathrm{NaN}$.
$\mathrm{atanh}(\mathrm{NaN}+iy)$ returns $\mathrm{NaN}+i\mathrm{NaN}$, for finite y.
$\mathrm{atanh}(x+i\mathrm{NaN})$ returns $\mathrm{NaN}+i\mathrm{NaN}$, for nonzero finite x.
$\mathrm{atanh}(\mathrm{NaN}+i\mathrm{NaN})$ returns $\mathrm{NaN}+i\mathrm{NaN}$.

Parameter

`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.

Parameter

`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(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).

Parameter

`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.

Exception

`ClassCastException` is thrown if obj is not a `Complex` object

conjugate

`static public Complex conjugate(Complex z)`

Description

Returns the complex conjugate of a `Complex` object.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the complex conjugate of `Complex` argument, z

cos

`static public Complex cos(Complex z)`

Description

Returns the cosine of a `Complex`. The value of cos is defined in terms of the function cosh, by $\cos(z) = \cosh(iz)$.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the cosine of the argument

cosh

`static public 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 $\mathrm{NaN}+i\mathrm{NaN}$.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the hyperbolic cosine of the argument

divide

```
static public 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

```
static public 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

```
static public 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.*

Parameter

`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(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*

Parameter

`obj` – the object to compare with

Returns

true if the objects are the same; false otherwise

---

exp

```
static public 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 $\pm\infty + i\text{NaN}$ (where the sign of the real part of the result is unspecified).
$\exp(x + i\infty)$ returns $\text{NaN} + i\text{NaN}$, 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.
$\exp(-\infty + i\text{NaN})$ returns $\pm 0 \pm i0$ (where the signs of the real and imaginary parts of the result are unspecified).
$\exp(+\infty + i\text{NaN})$ returns $\pm\infty + i\text{NaN}$ (where the sign of the real part of the result is unspecified).
$\exp(\text{NaN} + i0)$ returns $\text{NaN} + i0$.
$\exp(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$, for all non-zero numbers y.
$\exp(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$, for finite x.

Parameter

`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

```
static public double imag(Complex z)
```

Description

Returns the imaginary part of a `Complex` object.

Parameter

`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

```
static public 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.
$\log(\pm\infty + i\text{NaN})$ returns $+\infty + i\text{NaN}$.
$\log(\text{NaN} + i\infty)$ returns $+\infty + i\text{NaN}$.
$\log(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$, for finite x.
$\log(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$, for finite y.
$\log(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

Parameter

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

`static public 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

`static public 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

```
static public 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

```
static public 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

```
static public 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

```
static public Complex negate(Complex z)
```

Description

Returns the negative of a `Complex` object, -z.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the negative of the `Complex` argument, z

pow

```
static public 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

```
static public 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

```
static public double real(Complex z)
```

Description

Returns the real part of a `Complex` object.

Parameter

`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

```
static public Complex sin(Complex z)
```

Description

Returns the sine of a `Complex`. The value of sin is defined in terms of the function sinh, by $\sin(z) = -i\sinh(iz)$.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the sine of the argument

sinh

```
static public 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 $\pm 0+i\text{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(+\infty+i0)$ returns $+\infty+i0$.
$\sinh(+\infty+i\infty)$ returns $\pm\infty+i\text{NaN}$ (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 $\text{NaN}+i\text{NaN}$, for positive finite x.
$\sinh(+0+i\text{NaN})$ returns $\pm 0+i\text{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(+\infty+i\text{NaN})$ returns $\pm\infty+i\text{NaN}$ (where the sign of the real part of the result is unspecified).
$\sinh(x+i\text{NaN})$ returns $\text{NaN}+i\text{NaN}$, for finite nonzero x.

$\sinh(\text{NaN} + i0)$ returns $\text{NaN} + i0$.
$\sinh(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$, for all nonzero numbers y.
$\sinh(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the hyperbolic sine of the argument

sqrt

```
static public Complex sqrt(Complex z)
```

Description

Returns the square root of a `Complex`, with a branch cut along the negative real axis.

Specifically, if $z = x+iy$,
$\text{sqrt}(\bar{z}) = \overline{\text{sqrt}(z)}$.
$\text{sqrt}(\pm 0 + i0)$ returns $+0 + i0$.
$\text{sqrt}(-\infty + iy)$ returns $+0 + i\infty$, for finite positive-signed y.
$\text{sqrt}(+\infty + iy)$ returns $+\infty + i0$, for finite positive-signed y.
$\text{sqrt}(x + i\infty)$ returns $+\infty + i\infty$, for all x (including NaN).
$\text{sqrt}(-\infty + i\text{NaN})$ returns $\text{NaN} \pm i\infty$ (where the sign of the imaginary part of the result is unspecified).
$\text{sqrt}(+\infty + i\text{NaN})$ returns $+\infty + i\text{NaN}$.
$\text{sqrt}(x + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$ and optionally raises the invalid exception, for finite x.
$\text{sqrt}(\text{NaN} + iy)$ returns $\text{NaN} + i\text{NaN}$ and optionally raises the invalid exception, for finite y.
$\text{sqrt}(\text{NaN} + i\text{NaN})$ returns $\text{NaN} + i\text{NaN}$.

Parameter

`z` – a `Complex` object

Returns

A newly constructed `Complex` initialized to square root of z.

subtract

```
static public 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

```
static public 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

```
static public 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

```
static public Complex tan(Complex z)
```

Description

Returns the tangent of a `Complex`. The value of tan is defined in terms of the function tanh, by $\tan(z) = -i\tanh(iz)$.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the tangent of the argument

---

tanh

```
static public 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 $\mathrm{NaN}+i\mathrm{NaN}$.

Parameter

`z` – a `Complex` object

Returns

a newly constructed `Complex` initialized to the hyperbolic tangent of the argument

toString

```
public String toString()
```

Description

Returns a `String` representation for the specified `Complex`.

Returns

a `String` representation for this object

valueOf

```
static public Complex valueOf(String s) throws NumberFormatException
```

Description

Parses a `String` into a `Complex`.

Parameter

`s` – the `String` to be parsed

Returns

a newly constructed `Complex` initialized to the value represented by the `String` argument

Exceptions

`NumberFormatException` if the string does not contain a parsable `Complex` number

`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
```

Physical class

```
public class com.imsl.math.Physical extends java.lang.Number implements
Serializable, Cloneable
```

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]

The reference for constants are indicated by the code in the [] comment above.

[1]	Cohen and Taylor (1986)
[2]	Liepman (1964)
[3]	Precomputed mathematical constants

The constants marked with an E before the [] are exact (to machine precision).

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

Fields

CURRENT

`static final protected int CURRENT`

LENGTH

`static final protected int LENGTH`

MASS

`static final protected int MASS`

TEMPERATURE

`static final protected int TEMPERATURE`

TIME

`static final protected int TIME`

dim

`protected int[] dim`

value

`protected double value`

Constructors

Physical

`public Physical()`

Description

Constructs a new 0-valued, dimensionless object.

Physical

`public Physical(Physical copy)`

Description

Constructs a copy of a `Physical` object.

Parameter

`copy` – `Physical` object from which a copy is made

Physical

`public Physical(double value, 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(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

Methods

add

`static public 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

Exception

`IllegalArgumentException` is thrown if x and y are not compatible

checkCompatibility

```
static public 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

Exception

`IllegalArgumentException` is thrown if the two `Physical` objects are incompatible

constant

```
static public Physical constant(String name)
```

Description

Returns the value of a constant, given its name.

Parameter

`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

Exception

`IllegalArgumentException` is thrown when the name given is undefined

constant

```
static public double constant(String name, 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

Exception

`IllegalArgumentException` is thrown if the constant name is undefined

---

convert

`static public Physical convert(Physical pOld, 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

Exception

`IllegalArgumentException` is thrown if the new and old units are incompatible

---

defineConstant

`static public void defineConstant(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

`static public void definePrefix(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

`static public void defineUnit(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

```
static public 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

divide

```
static public 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

```
static public 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

doubleValue

```
public double doubleValue()
```

Description

Returns the value of this dimensionless object.

Returns

the `double` value of the dimensionless object

Exception

`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

Exception

`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

Exception

`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

Exception

`IllegalArgumentException` is thrown if the this object is not dimensionless

---

multiply

```
static public 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

multiply

```
static public 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

```
static public 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

negate

```
static public Physical negate(Physical x)
```

Description

Negate a `Physical` object.

Parameter

x – `Physical` object which is to be negated

Returns

Physical object which has been negated

subtract

static public 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

Exception

IllegalArgumentException is thrown if x and y are not compatible

toString

public 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 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
```

EpsilonAlgorithm class

`public class com.imsl.math.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.

Constructors

EpsilonAlgorithm

`public EpsilonAlgorithm()`

Description

Initializes an EpsilonAlgorithm with a maximum table size of 50.

EpsilonAlgorithm

`public EpsilonAlgorithm(int maxTableSize)`

Description

Initializes an EpsilonAlgorithm.

Parameter

`maxTableSize` – The maximum table size.

Methods

extrapolate

`public double extrapolate(double x)`

Description

Extrapolates the convergence limit of a sequence.

Parameter

`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.

Example: The Epsilon Algorithm

The Epsilon algorithm is used to accelerate a series of partial sums of an infinite sum. The error shows that the Epsilon algorithm is effective.

```
import com.imsl.math.*;
import java.text.DecimalFormat;

public class EpsilonAlgorithmEx1 {

    static public void main(String arg[]) {
        EpsilonAlgorithm eps = new EpsilonAlgorithm();

        int n = 100;
        double sum = 0.0;
        for (int i = 1; i < n; i++) {
            sum += 1.0 / (i * i);
            eps.extrapolate(sum);
        }
        sum += 1.0 / (n * n);
        double extrapolated = eps.extrapolate(1.0 / (n * n));
        double expected = Math.PI * Math.PI / 6.0;
        double exError = expected - extrapolated;
        double sumError = expected - sum;
        DecimalFormat format = new DecimalFormat("0.00000");
        System.out.println("Expected     " + format.format(expected));
        System.out.print("Extrapolated " + format.format(extrapolated));
        System.out.println("   error =" + format.format(exError));
        System.out.print("Sum          " + format.format(sum));
        System.out.println("   error =" + format.format(sumError));
    }
}
```

Output

```
Expected     1.64493
Extrapolated 1.64278   error =0.00216
Sum          1.63498   error =0.00995
```

Chapter 12: Printing Functions

Types

class PrintMatrix 583
class PrintMatrixFormat 588

PrintMatrix class

```
public class com.imsl.math.PrintMatrix
```

Matrix printing utilities.

Fields

FULL

```
static final public int FULL
```

This flag as the argument to setMatrixType, indicates that the full matrix is to be printed.

LOWER_TRIANGULAR

```
static final public 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.

STRICT_LOWER_TRIANGULAR

```
static final public 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.

STRICT_UPPER_TRIANGULAR

```
static final public 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.

UPPER_TRIANGULAR

```
static final public 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.

Constructors

PrintMatrix

```
public PrintMatrix()
```

Description

Creates an instance of the PrintMatrix class.

PrintMatrix

```
public PrintMatrix(PrintStream out)
```

Description

Creates an instance of the PrintMatrix class with the specified PrintStream.

Parameter

`out` – a `PrintStream`

PrintMatrix

```
public PrintMatrix(String title)
```

Description

Creates a PrintMatrix object and sets its title.

Parameter

`title` – a `String` specifying the title

PrintMatrix

```
public PrintMatrix(PrintStream out, 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

Methods

print

`public void print(Object array)`

Description

Prints a matrix with a default format.

Parameter

`array` – a two-dimensional, non-empty, rectangular array

print

`protected void print(String string)`

Description

Print a string. This function can be overridden to print to something other than a PrintStream.

Parameter

`string` – the `String` to be printed

print

`public void print(PrintMatrixFormat pmf, Object array)`

Description

Prints a matrix with specified format.

Parameters

`pmf` – a `PrintMatrixFormat` matrix format

`array` – a two-dimensional, non-empty, rectangular array

printHTML

`public void printHTML(PrintMatrixFormat pmf, 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.

Parameter

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.

Parameter

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.

Parameter

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.

Parameter

`pageWidth` – an `int` specifying the page width, default is the largest possible integer

Returns

the `PrintMatrix` object

setTitle

`public PrintMatrix setTitle(String title)`

Description

Sets matrix title

Parameter

`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
```

PrintMatrixFormat class

`public class com.imsl.math.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.

Fields

BEGIN_COLUMN_LABEL

`static final public 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.

BEGIN_COLUMN_LABELS

`static final public 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.

BEGIN_ENTRY

`static final public int BEGIN_ENTRY`

This flag as the type argument to format, indicates that the formatted string for beginning an entry is to be returned.

BEGIN_MATRIX

```
static final public int BEGIN_MATRIX
```

This flag as the type argument to format, indicates that the formatting string for beginning a matrix is to be returned.

BEGIN_ROW

```
static final public int BEGIN_ROW
```

This flag as the type argument to format, indicates that the formatting string for beginning a row is to be returned.

BEGIN_ROW_LABEL

```
static final public 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.

COLUMN_LABEL

```
static final public 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.

END_COLUMN_LABEL

```
static final public 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.

END_COLUMN_LABELS

```
static final public 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.

END_ENTRY

```
static final public int END_ENTRY
```

This flag as the type argument to format, indicates that the formatted string for ending an entry is to be returned.

END_MATRIX

```
static final public int END_MATRIX
```

This flag as the type argument to format, indicates that the formatting string for ending a matrix is to be returned.

END_ROW

```
static final public int END_ROW
```

This flag as the type argument to format, indicates that the formatting string for ending a row is to be returned.

END_ROW_LABEL

```
static final public 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.

ENTRY

```
static final public int ENTRY
```

This flag as the type argument to format, indicates that the formatted string for a given entry is to be returned.

ROW_LABEL

```
static final public 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.

numberFormat

```
protected 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 String format(int type, Object entry, int row, int col, ParsePosition
pos)
```

Description

Returns a formatted string. This method is used by the methods `print` and `printHTML` in `PrintMatrix`. This method can be overridden to gain finer control over printing.

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 NumberFormat getNumberFormat()
```

Description

Returns the NumberFormat to be used in formatting double and Complex entries.

setColumnLabels

```
public void setColumnLabels(String[] columnLabels)
```

Description

Turns on column labeling using the given labels.

Parameter

`columnLabels` – is an array of `Strings` 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.

Parameter

`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.

Parameter

`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(NumberFormat numberFormat)
```

Description

Sets the NumberFormat to be used in formatting double and Complex entries.

Parameter

`numberFormat` – a `NumberFormat` or null. If `null` then numbers will be formatted using `java.lang.Integer.toString` , or `java.lang.Object.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.*;

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
```

Example: Matrix Formatting

A matrix is printed in CSV (comma separated value) format. This is done by overriding the `format` method of `PrintMatrixFormat` to add commas after all but the last number in each row.

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

public class PrintMatrixFormatEx2 extends PrintMatrixFormat {

    private int ncols;

    public PrintMatrixFormatEx2(int ncols) {
        this.ncols = ncols;
    }

    public String format(int type, Object entry,
            int row, int col, ParsePosition pos) {
        String text = super.format(type, entry, row, col, pos);
        if (type == ENTRY) {
            if (col < ncols - 1) {
                text += ",";
            }
        }
        return text;
    }

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

        PrintMatrixFormat mf = new PrintMatrixFormatEx2(3);
        mf.setNoRowLabels();
        mf.setNoColumnLabels();

        //  Print the matrix
        new PrintMatrix().print(mf, a);
    }
}
```

Output

```
0,  1,  2
4,  5,  6
8,  9,  8
6,  3,  4
```

Chapter 13: Basic Statistics

Types

class Summary . . . 595
class Covariances . . . 607
class PartialCovariances . . . 617
class PooledCovariances . . . 623
class NormOneSample . . . 632
class NormTwoSample . . . 638
class Sort . . . 650
class Ranks . . . 659
class EmpiricalQuantiles . . . 668
class TableOneWay . . . 671
class TableTwoWay . . . 676
class TableMultiWay . . . 682

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.

Summary class

```
public class com.imsl.stat.Summary implements Serializable, Cloneable
```

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,

$$\frac{\sum f_i w_i \left(x_i - \bar{x}_w\right)^3 / n}{[\sum f_i w_i \left(x_i - \bar{x}_w\right)^2 / n]^{3/2}}$$

Excess or Kurtosis,

$$\frac{\sum f_i w_i \left(x_i - \bar{x}_w\right)^4 / n}{[\sum f_i w_i \left(x_i - \bar{x}_w\right)^2 / n]^2} - 3$$

Minimum,

$$x_{\min} = \min(x_i)$$

Maximum,

$$x_{\max} = \max(x_i)$$

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).

Parameter

`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).

Parameter

`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

---

getNumberOfObservations

```
public int getNumberOfObservations()
```

Description

Returns the number of non-missing observations.

Returns

an `int`, the number of non-missing observations in the `Summary` object.

---

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

`static public double kurtosis(double[] x)`

Description

Returns the kurtosis of the given data set.

Parameter

`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

`static public 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

```
static public double maximum(double[] x)
```

Description

Returns the maximum of the given data set.

Parameter

`x` – a `double` array containing the data set whose maximum is to be found

Returns

a `double`, the maximum of the given data set

maximum

```
static protected int maximum(int[] x)
```

Description

Returns the maximum of the given data set.

Parameter

`x` – an `int` array containing the data set whose maximum is to be found

Returns

an `int`, the maximum of the given data set

mean

```
static public double mean(double[] x)
```

Description

Returns the mean of the given data set.

Parameter

`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

```
static public 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

```
static public double median(double[] x)
```

Description

Returns the median of the given data set.

Parameter

x – a double array containing the data set whose median is to be found

Returns

a double, the median of the given data set

median

```
static public double median(double[] x, double[] weight)
```

Description

Returns the weighted median of the given data set and associated weights.

Parameters

x – a double array containing the data set whose median is to be found

weight – a double array containing the weights associated with the data

Returns

a double, the weighted median of the given data set

minimum

```
static public double minimum(double[] x)
```

Description

Returns the minimum of the given data set.

Parameter

x – a double array containing the data set whose minimum is to be found.

Returns

a double, the minimum of the given data set.

minimum

```
static protected int minimum(int[] x)
```

Description

Returns the minimum of the given data set.

Parameter

x – an int array containing the data set whose minimum is to be found

Returns

an int, the minimum of the given data set

mode

```
static public double mode(double[] x)
```

Description

Returns the mode of the given data set. Ties are broken at random.

Parameter

x – a double array containing the data set whose mode is to be found

Returns

a double, the mode of the given data set

numberOfObservations

```
static public int numberOfObservations(double[] x)
```

Description

Returns the number of non-missing observations in the given data set.

Parameter

x – a double array containing the data set.

Returns

an int, the number of non-missing observations in the given data set.

quantile

```
static public double quantile(double[] x, double[] weight, double alpha)
```

sampleStandardDeviation

```
static public double sampleStandardDeviation(double[] x)
```

Description

Returns the sample standard deviation of the given data set.

Parameter

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

```
static public 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

```
static public double sampleVariance(double[] x)
```

Description

Returns the sample variance of the given data set.

Parameter

`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

```
static public 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

```
static public double skewness(double[] x)
```

Description

Returns the skewness of the given data set.

Parameter

`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

```
static public 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

```
static public double standardDeviation(double[] x)
```

Description

Returns the population standard deviation of the given data set.

Parameter

`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

```
static public 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 void update(double x)
```

Description

Adds an observation to the `Summary` object.

Parameter

`x` – a `double`, the data observation to be added

---

update

```
public void update(double[] x)
```

Description

Adds a set of observations to the `Summary` object.

Parameter

`x` – a `double` array of data observations to be added

---

update

```
public 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

---

update

```
public 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

---

variance

```
static public double variance(double[] x)
```

Description

Returns the population variance of the given data set.

Parameter

`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

```
static public 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[] = summary.confidenceMean(0.95);
```

```
        System.out.println("The confidence Mean is {" + confmn[0]
                + ", " + confmn[1] + "}");
        System.out.println();

        double 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 {33.782618727206625, 194.0685332772439}
```

Covariances class

`public class com.imsl.stat.Covariances implements Serializable, Cloneable`

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.

Fields

CORRECTED_SSCP_MATRIX

```
static final public int CORRECTED_SSCP_MATRIX
```

Indicates corrected sums of squares and crossproducts matrix.

CORRELATION_MATRIX

```
static final public int CORRELATION_MATRIX
```

Indicates correlation matrix.

STDEV_CORRELATION_MATRIX

```
static final public int STDEV_CORRELATION_MATRIX
```

Indicates correlation matrix except for the diagonal elements which are the standard deviations

VARIANCE_COVARIANCE_MATRIX

```
static final public int VARIANCE_COVARIANCE_MATRIX
```

Indicates variance-covariance matrix.

Constructor

Covariances

`public Covariances(double[][] x)`

Description

Constructor for `Covariances`.

Parameter

`x` – A `double` matrix containing the data.

Exception

`IllegalArgumentException` is thrown if `x.length`, and `x[0].length` are equal to 0.

Methods

compute

```
final public double[][] compute(int matrixType) throws
Covariances.NonnegativeFreqException, Covariances.NonnegativeWeightException,
Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, Covariances.DiffObsDeletedException
```

Description

Computes the matrix.

Parameter

`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.

Exceptions

`NonnegativeFreqException` is thrown if the frequencies are negative.

`NonnegativeWeightException` is thrown if the weights are negative.

`TooManyObsDeletedException` is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative.

`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.

`DiffObsDeletedException` is thrown if different observations are being deleted than were originally entered.

getIncidenceMatrix

`public int[][] getIncidenceMatrix()`

Description

Returns the incidence matrix. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

An `int` matrix containing the incidence matrix. If `method` is 0, incidence matrix is 1×1 and contains the number of valid observations; otherwise, incidence matrix is $x[0]$.length $\times$ $x[0]$.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`. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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`). Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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.

Parameter

`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.

Parameter

`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.

Parameter

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.*;

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
```

Covariances.NonnegativeFreqException class

`static public class com.imsl.stat.Covariances.NonnegativeFreqException extends com.imsl.IMSLException`

Frequencies must be nonnegative.

Constructors

Covariances.NonnegativeFreqException

`public Covariances.NonnegativeFreqException(String message)`

Description

Constructs a `NonnegativeFreqException` object.

Parameter

`message` – a `String` containing the error message

Covariances.NonnegativeFreqException

`public Covariances.NonnegativeFreqException(String key, Object[] arguments)`

Description

Constructs a `NonnegativeFreqException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

Covariances.NonnegativeWeightException class

```
static public class com.imsl.stat.Covariances.NonnegativeWeightException
extends com.imsl.IMSLException
```

Weights must be nonnegative.

Constructors

Covariances.NonnegativeWeightException

`public Covariances.NonnegativeWeightException(String message)`

Description

Constructs a `NonnegativeWeightException` object.

Parameter

`message` – a `String` containing the error message

Covariances.NonnegativeWeightException

`public Covariances.NonnegativeWeightException(String key, Object[] arguments)`

Description

Constructs a `NonnegativeWeightException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

Covariances.TooManyObsDeletedException class

```
static public class com.imsl.stat.Covariances.TooManyObsDeletedException
extends com.imsl.IMSLException
```

Constructors

Covariances.TooManyObsDeletedException

```
public Covariances.TooManyObsDeletedException(String message)
```

Description

Constructs a `TooManyObsDeletedException` object.

Parameter

`message` – a `String` containing the error message

Covariances.TooManyObsDeletedException

```
public Covariances.TooManyObsDeletedException(String key, Object[] arguments)
```

Description

Constructs a `TooManyObsDeletedException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

Covariances.MoreObsDelThanEnteredException class

```
static public class com.imsl.stat.Covariances.MoreObsDelThanEnteredException
extends com.imsl.IMSLException
```

Constructors

Covariances.MoreObsDelThanEnteredException

```
public Covariances.MoreObsDelThanEnteredException(String message)
```

Description

Constructs a `MoreObsDelThanEnteredException` object.

Parameter

`message` – a `String` containing the error message

Covariances.MoreObsDelThanEnteredException

```
public Covariances.MoreObsDelThanEnteredException(String key, Object[]
arguments)
```

Description

Constructs a `MoreObsDelThanEnteredException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

Covariances.DiffObsDeletedException class

```
static public class com.imsl.stat.Covariances.DiffObsDeletedException extends
com.imsl.IMSLException
```

Constructors

Covariances.DiffObsDeletedException

```
public Covariances.DiffObsDeletedException(String message)
```

Description

Constructs a `DiffObsDeletedException` object.

Parameter

`message` – a `String` containing the error message

Covariances.DiffObsDeletedException

```
public Covariances.DiffObsDeletedException(String key, Object[] arguments)
```

Description

Constructs a `DiffObsDeletedException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

PartialCovariances class

```
public class com.imsl.stat.PartialCovariances implements Serializable
```

Class `PartialCovariances` computes the partial covariances or partial correlations from an input covariance or correlation matrix. If the “independent” variables (the linear “effect” of the independent variables is removed in computing the partial covariances/correlations) are linearly related to one another, `PartialCovariances` detects the linearity and eliminates one or more of the independent variables from the list of independent variables. The number of variables eliminated, if any, can be determined from `getPartialDegreesOfFreedom`.

Given a covariance or correlation matrix Σ partitioned as

$$\begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}$$

class `PartialCovariances` computes the partial covariances (of the standardized variables if Σ is a correlation matrix) as

$$\Sigma_{22|1} = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}$$

A positive semidefinite solver is used to compute $\Sigma_{11}^{-1}\Sigma_{12}$.

If partial correlations are desired, these are computed as

$$P_{22|1} = \left[\text{diag}(\Sigma_{22|1})\right]^{-1/2}\Sigma_{22|1}\left[\text{diag}(\Sigma_{22|1})\right]^{-1/2}$$

where $\text{diag}(\Sigma)$ denotes the matrix containing the diagonal of its argument along its diagonal with zeros off the diagonal. If Σ_{11} is singular, then as many variables as required are deleted from Σ_{11} (and Σ_{12}) in order to eliminate the linear dependencies. The computations then proceed as above.

The p-value for a partial covariance tests the null hypothesis $H_0 : \sigma_{ij|1} = 0$, where $\sigma_{ij|1}$ is the (i, j) element in matrix $\Sigma_{22|1}$. The p-value for a partial correlation tests the null hypothesis $H_0 : \rho_{ij|1} = 0$, where $\rho_{ij|1}$ is the (i, j) element in matrix $P_{22|1}$. The p-values are returned by `getPValues`. If the degrees of freedom for `sigma`, `df`, is not known, the resulting p-values may be useful for comparison, but they should not by used as an approximation to the actual probabilities.

Constructors

PartialCovariances

```
public PartialCovariances(int nIndependent, double[][] sigma, int df) throws
PartialCovariances.InvalidMatrixException,
PartialCovariances.InvalidPartialCorrelationException
```

Description

Creates a `PartialCovariances` object from a covariance or correleation matrix with a the independent variables in the initial columns and the dependent variables in the final columns.

Parameters

`nIndependent` – is the number of “independent” variables to be used in the partial covariances/correlations. The partial covariances/correlations are the covariances/correlations between the dependent variables after removing the linear effect of the independent variables.

`sigma` – is a correlation or covariance matrix. The rows/columns must be ordered such that the first `nIndependent` rows/columns contain the independent variables, followed by the row/columns containing the dependent variables. The matrix must always be symmetric, positive semidefinite.

`df` – is an `int` indicating the number of degrees of freedom associated with the input matrix. If the number of degrees of freedom in the matrix varies from element to element, then a conservative choice for `df` is the minimum degrees of freedom for all elements in the matrix. The value of `df` must be at least one.

Exceptions

`com.imsl.stat.PartialCovariances.InvalidMatrixException` is thrown if a computed correlation is greater than one for some pair of variables.

`com.imsl.stat.PartialCovariances.InvalidPartialCorrelationException` is thrown if a computed partial correlation is greater than one for some pair of variables. The input matrix to the constructor was not positive semidefinite.

PartialCovariances

```
public PartialCovariances(int[] xIndices, double[][] sigma, int df) throws
PartialCovariances.InvalidMatrixException,
PartialCovariances.InvalidPartialCorrelationException
```

Description

Creates a `PartialCovariances` object from a covariance or correleation matrix with a mix of dependent and independent variables.

Parameters

`xIndices` – is an array containing values indicating the status of the variable. If the i-th entry is 0 then the i-th column of the matrix contains a dependent variable. If the i-th entry is positive then the i-th column of the matrix contains an independent variable. If the i-th entry is negative then the i-th column of the matrix is not used in the analysis.

`sigma` – is a correlation or covariance matrix. The number of rows and columns in `sigma` must equal the length of the array `xIndices`. The matrix must always be symmetric, positive semidefinite.

`df` – is an `int` indicating the number of degrees of freedom associated with the input matrix. If the number of degrees of freedom in the matrix varies from element to element, then a conservative choice for `df` is the minimum degrees of freedom for all elements in the matrix. The value of `df` must be at least one.

Exceptions

`com.imsl.stat.PartialCovariances.InvalidMatrixException` is thrown if a computed correlation is greater than one for some pair of variables.

`com.imsl.stat.PartialCovariances.InvalidPartialCorrelationException` is thrown if a computed partial correlation is greater than one for some pair of variables. The input matrix to the constructor was not positive semidefinite.

Methods

getPValues

`public double[][] getPValues()`

Description

Calculates the *p*-values for testing the null hypothesis that the associated partial covariance/correlation is zero. It is assumed that the observations from which `sigma` was computed follows a multivariate normal distribution and that each element in `sigma` has `df` degrees of freedom.

Returns

A square array of type `double` containing the *p*-values. The order of the matrix equals the number of dependent variables.

If the partial degrees of freedom is not greater than one then there are not enough degrees of freedom for hypothesis testing. The returned matrix will be set to all NaN values in this case. A warning is also issued in this case.

getPartialCorrelationMatrix

`public double[][] getPartialCorrelationMatrix()`

Description

Returns the partial correlation matrix. This is valid only if the input to the constructor was a correlation matrix.

Returns

The partial correlation matrix.

getPartialCovarianceMatrix

`public double[][] getPartialCovarianceMatrix()`

Description

Returns the partial covariance matrix. This is valid only if the input to the constructor was a covariance matrix.

Returns

The partial covariance matrix.

getPartialDegreesOfFreedom

`public int getPartialDegreesOfFreedom()`

Description

Returns the degrees of freedom in the test that the partial correlation (covariance) is zero. This will usually be `df` minus the rank of the the independent variables. number of independent variables, but will be greater than this value if the independent variables are computationally linearly related.

Returns

An `int` scalar value representing the degrees of freedom. If this value is not greater than one then there are not enough degrees of freedom for hypothesis testing. A warning is also issued in this case.

Example 1

This example computes partial covariances, scaled from a nine-variable correlation matrix originally given by Emmett (1949). The first three rows and columns contain the independent variables and the final six rows and columns contain the dependent variables.

```
import com.imsl.stat.PartialCovariances;
import com.imsl.math.PrintMatrix;

public class PartialCovariancesEx1 {

    static public void main(String arg[]) throws Exception {
        double sigma[][] = {
            {6.300, 3.050, 1.933, 3.365, 1.317, 2.293, 2.586, 1.242, 4.363},
            {3.050, 5.400, 2.170, 3.346, 1.473, 2.303, 2.274, 0.750, 4.077},
            {1.933, 2.170, 3.800, 1.970, 0.798, 1.062, 1.576, 0.487, 2.673},
            {3.365, 3.346, 1.970, 8.100, 2.983, 4.828, 2.255, 0.925, 3.910},
            {1.317, 1.473, 0.798, 2.983, 2.300, 2.209, 1.039, 0.258, 1.687},
            {2.293, 2.303, 1.062, 4.828, 2.209, 4.600, 1.427, 0.768, 2.754},
            {2.586, 2.274, 1.576, 2.255, 1.039, 1.427, 3.200, 0.785, 3.309},
            {1.242, 0.750, 0.487, 0.925, 0.258, 0.768, 0.785, 1.300, 1.458},
            {4.363, 4.077, 2.673, 3.910, 1.687, 2.754, 3.309, 1.458, 7.400}
        };
        int nIndependent = 3, df = 30;

        PartialCovariances pcov
                = new PartialCovariances(nIndependent, sigma, df);

        double covar[][] = pcov.getPartialCovarianceMatrix();
        new PrintMatrix("Partial Covariances").print(covar);

        int pdf = pcov.getPartialDegreesOfFreedom();
        System.out.println("Partial Degrees of Freedom " + pdf);
        System.out.println();

        double pvalues[][] = pcov.getPValues();
        new PrintMatrix("p Values").print(pvalues);
    }
}
```

Output

```
        Partial Covariances
```

```
   0   1  2    3       4      5
0   0  0  0  -0      0      0
1   0  0  0   0      0      0
2   0  0  0   0      0      0
3  -0  0  0   5.495  1.895  3.084
4   0  0  0   1.895  1.841  1.476
5   0  0  0   3.084  1.476  3.403

Partial Degrees of Freedom 27

            p Values
   0  1  2   3      4      5
0  0  0  0  0      0      0
1  0  0  0  0      0      0
2  0  0  0  0      0      0
3  0  0  0  0      0.001  0
4  0  0  0  0.001  0      0.001
5  0  0  0  0      0.001  0
```

Example 2

This example computes partial correlations from a 9 variable correlation matrix originally given by Emmett (1949). The partial correlations between the remaining variables, after adjusting for variables 1, 3 and 9, are computed. Note in the output that the row and column labels are numbers, not variable numbers. The corresponding variable numbers would be 2, 4, 5, 6, 7 and 8, respectively.

```
import com.imsl.stat.PartialCovariances;
import com.imsl.math.PrintMatrix;

public class PartialCovariancesEx2 {

    static public void main(String arg[]) throws Exception {
        double sigma[][] = {
            {1.000, 0.523, 0.395, 0.471, 0.346, 0.426, 0.576, 0.434, 0.639},
            {0.523, 1.000, 0.479, 0.506, 0.418, 0.462, 0.547, 0.283, 0.645},
            {0.395, 0.479, 1.000, 0.355, 0.270, 0.254, 0.452, 0.219, 0.504},
            {0.471, 0.506, 0.355, 1.000, 0.691, 0.791, 0.443, 0.285, 0.505},
            {0.346, 0.418, 0.270, 0.691, 1.000, 0.679, 0.383, 0.149, 0.409},
            {0.426, 0.462, 0.254, 0.791, 0.679, 1.000, 0.372, 0.314, 0.472},
            {0.576, 0.547, 0.452, 0.443, 0.383, 0.372, 1.000, 0.385, 0.680},
            {0.434, 0.283, 0.219, 0.285, 0.149, 0.314, 0.385, 1.000, 0.470},
            {0.639, 0.645, 0.504, 0.505, 0.409, 0.472, 0.680, 0.470, 1.000}
        };

        int xIndices[] = {1, 0, 1, 0, 0, 0, 0, 0, 1};
        int df = 30;

        PartialCovariances pcov = new PartialCovariances(xIndices, sigma, df);

        double correl[][] = pcov.getPartialCorrelationMatrix();
        new PrintMatrix("Partial Correlations").print(correl);

        double pValues[][] = pcov.getPValues();
```

```
        new PrintMatrix("P-Values").print(pValues);
    }
}
```

Output

```
              Partial Correlations
     0       1      2       3      4      5
0   1       0.224   0.194  0.211  0.125  -0.061
1   0.224  1        0.605  0.72   0.092   0.025
2   0.194  0.605   1       0.598  0.123  -0.077
3   0.211  0.72    0.598  1       0.035   0.086
4   0.125  0.092   0.123  0.035  1        0.062
5  -0.061  0.025  -0.077  0.086  0.062   1

                  P-Values
     0      1      2      3      4      5
0  0      0.252  0.323  0.28   0.525  0.758
1  0.252  0      0.001  0      0.642  0.9
2  0.323  0.001  0      0.001  0.533  0.698
3  0.28   0      0.001  0      0.86   0.665
4  0.525  0.642  0.533  0.86   0      0.753
5  0.758  0.9    0.698  0.665  0.753  0
```

PartialCovariances.InvalidMatrixException class

`static public class com.imsl.stat.PartialCovariances.InvalidMatrixException extends com.imsl.IMSLException`

Exception thrown if a computed correlation is greater than one for some pair of variables.

Constructor

PartialCovariances.InvalidMatrixException

`public PartialCovariances.InvalidMatrixException(int var1, int var2)`

Description

Creates an InvalidMatrixException thrown if a computed correlation is greater than one for some pair of variables.

Parameters

`var1` – is the index of the first variable in the pair.

`var2` – is the index of the second variable in the pair.

PartialCovariances.InvalidPartialCorrelationException class

```
static public class
com.imsl.stat.PartialCovariances.InvalidPartialCorrelationException extends
com.imsl.IMSLException
```

Exception thrown if a computed partial correlation is greater than one for some pair of variables.

Constructor

PartialCovariances.InvalidPartialCorrelationException

```
public PartialCovariances.InvalidPartialCorrelationException(int var1, int
var2)
```

Description

Creates an InvalidPartialCorrelationException thrown if a computed partial correlation is greater than one for some pair of variables.

Parameters

`var1` – is the index of the first variable in the pair.

`var2` – is the index of the second variable in the pair.

PooledCovariances class

```
public class com.imsl.stat.PooledCovariances implements Serializable, Cloneable
```

Computes a pooled variance-covariance matrix from one or more sets of observations.

Class `PooledCovariances` computes the pooled variance-covariance matrix from one or more matrices of observations. The within-groups means are also computed. Listwise deletion of missing values is assumed so that all observations used are complete; for any row of `x`, if any element of the observation is missing (with a value of `Double.NaN`), the row is not used. This class should be used whenever the user suspects that the data has been sampled from populations with different means but identical

variance-covariance matrices. If these assumptions cannot be made, a different variance-covariance matrix should be estimated within each group.

Group observation totals, T_i for i = 1, ..., g, where g is the number of groups, are computed as:

$$T_i = \sum_j w_{ij} f_{ij} x_{ij}$$

where w_{ij} is the observation weight, x_{ij} is the j-th observation in the i-th group, and f_{ij} is the observation frequency.

Modified Givens rotations are used in computing the Cholesky decomposition of the pooled sums of squares and crossproducts matrix (Golub and Van Loan 1983).

The group means and the pooled sample covariance matrix S are computed from intermediate results. These quantities are defined by

$$\bar{x}_i = \frac{T_i}{\sum_j w_{ij} f_{ij}}$$

$$S = \frac{1}{\sum_{ij} f_{ij} - g} \sum_{ij} w_{ij} f_{ij} \left(x_{ij} - \bar{x}_i\right)\left(x_{ij} - \bar{x}_i\right)^T$$

Constructor

PooledCovariances

`public PooledCovariances(int nGroups)`

Description

Constructor for `PooledCovariances`.

Parameter

`nGroups` – an `int`, the number of groups in the data. The groups are numbered 1, 2,..., `nGroups`.

Methods

getGroupCounts

`public int[] getGroupCounts()`

Description

Returns the number of observations in each group.

Returns

an `int` array of length `nGroups` containing the number of observations in each group

getMeans

```
public double[][] getMeans()
```

Description

Returns the means of each group.

Note that one of the `update` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a `double` matrix with `nGroups` rows. The i-th row contains the group i variable means.

getNumberOfGroups

```
public int getNumberOfGroups()
```

Description

Returns the number of groups used in the analysis.

Returns

an `int`, the number of groups

getNumberOfMissingRows

```
public int getNumberOfMissingRows()
```

Description

Returns the total number of observations that contain missing values (`Double.NaN` or `group[i] == 0`).

Returns

an `int` containing the total number of observations with missing values

getNumberOfVariables

```
public int getNumberOfVariables()
```

Description

Returns the number of variables used in the analysis.

Returns

an `int`, the number of variables

getPooledCovariances

```
public double[][] getPooledCovariances()
```

Description

Computes and returns the pooled covariances.

Note that one of the `update` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a square `double` matrix of order `nVar`, the number of observation variables, containing the pooled covariances

getSumOfWeights

```
public double[] getSumOfWeights()
```

Description

Returns the sum of the weights times the frequencies in the groups.

Returns

a `double` array of length `nGroups` containing the sum of the weights times the frequencies in the groups

getTotalNumberOfObservations

```
public int getTotalNumberOfObservations()
```

Description

Returns the total number of observations used in the analysis.

Returns

an `int`, the total number of observations from all `update` invocations

getU

```
public double[][] getU()
```

Description

Returns the lower matrix U, the lower triangular for the pooled sample crossproducts matrix. U is computed from the pooled sample covariance matrix, S, as $S = U^T U$.

Note that one of the `update` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a square `double` matrix of order `nVar`, the number of observation variables, containing U

update

```
public void update(double[][] x)
```

Description

Updates the pooled covariances with new observations from one group.

Parameter

> `x` – a `double` matrix containing the observed data. Each row of `x` contains one observation consisting of `x[0].length` variables. If `x[i][j]` has value `Double.NaN`, then row i of the observations will be skipped and counted as missing.
>
> The number of observation variables is determined in the first call to any of the `update` methods. In all subsequent `update` calls, the number of observation variables must be the same.
>
> This method assumes that all observations belong to group 1 and have frequencies and weights of 1.0.

update

```
public void update(double[][] x, int[] groups)
```

Description

Updates the pooled covariances with new group observations.

Parameters

`x` – a `double` matrix containing the observed data. Each row of `x` contains one observation consisting of `x[0].length` variables. If `x[i][j]` has value `Double.NaN`, then row *i* of the observations will be skipped and counted as missing.

The number of observation variables is determined in the first call to any of the `update` methods. In all subsequent `update` calls, the number of observation variables must be the same.

This method assumes that all observations have frequencies and weights of 1.0.

`groups` – an `int` array containing the group number of the observations in `x`. Group numbers must be numbered 1, 2,..., `nGroups`. If `groups[i] == 0`, the row of observations will be skipped and counted as missing. For `groups[i]` < 0 or `groups[i]` > `nGroups`, a warning will be issued indicating that the row of observations will be skipped (not marked as missing).

update

```
public void update(double[][] x, int[] groups, double frequency, double weight)
```

Description

Updates the pooled covariances with new group observations and a scalar frequency and weight.

Parameters

`x` – a `double` matrix containing the observed data. Each row of `x` contains one observation consisting of `x[0].length` variables. If `x[i][j]` has value `Double.NaN`, then row *i* of the observations will be skipped and counted as missing.

The number of observation variables is determined in the first call to any of the `update` methods. In all subsequent `update` calls, the number of observation variables must be the same.

`groups` – an `int` array containing the group number of the observations in `x`. Group numbers must be numbered 1, 2,..., `nGroups`. If `groups[i] == 0`, the row of observations will be skipped and counted as missing. For `groups[i]` < 0 or `groups[i]` > `nGroups`, a warning will be issued indicating that the row of observations will be skipped (not marked as missing).

`frequency` – a positive `double` containing the frequency valid for each observation

`weight` – a positive `double` containing the weight valid for each observation

update

```
public void update(double[][] x, int[] groups, double frequency, double[]
weights)
```

Description

Updates the pooled covariances with new group observations, a scalar frequency and weights.

Parameters

`x` – a `double` matrix containing the observed data. Each row of `x` contains one observation consisting of `x[0].length` variables. If `x[i][j]` has value `Double.NaN`, then row *i* of the observations will be skipped and counted as missing.

The number of observation variables is determined in the first call to any of the `update` methods. In all subsequent `update` calls, the number of observation variables must be the same.

`groups` – an `int` array containing the group number of the observations in `x`. Group numbers must be numbered 1, 2,..., `nGroups`. If `groups[i] == 0`, the row of observations will be skipped and counted as missing. For `groups[i]` < 0 or `groups[i]` > `nGroups`, a warning will be issued indicating that the row of observations will be skipped (not marked as missing).

`frequency` – a positive `double` containing the frequency valid for each observation

`weights` – a `double` array of size `x.length` containing the weight for each observation. Each value must be positive. Any `Double.NaN` value results in that observation being skipped and marked missing.

update

`public void update(double[][] x, int[] groups, double[] frequencies, double weight)`

Description

Updates the pooled covariances with new group observations, frequencies and a scalar weight.

Parameters

`x` – a `double` matrix containing the observed data. Each row of `x` contains one observation consisting of `x[0].length` variables. If `x[i][j]` has value `Double.NaN`, then row *i* of the observations will be skipped and counted as missing.

The number of observation variables is determined in the first call to any of the `update` methods. In all subsequent `update` calls, the number of observation variables must be the same.

`groups` – an `int` array containing the group number of the observations in `x`. Group numbers must be numbered 1, 2,..., `nGroups`. If `groups[i] == 0`, the row of observations will be skipped and counted as missing. For `groups[i]` < 0 or `groups[i]` > `nGroups`, a warning will be issued indicating that the row of observations will be skipped (not marked as missing).

`frequencies` – a `double` array of size `x.length` containing the frequency for each observation. Each value must be positive. Any `Double.NaN` value results in that observation being skipped and marked missing.

`weight` – a positive `double` containing the weight valid for each observation

update

`public void update(double[][] x, int[] groups, double[] frequencies, double[] weights)`

Description

Updates the pooled covariances with new group observations, frequencies and weights.

Parameters

x – a double matrix containing the observed data. Each row of x contains one observation consisting of x[0].length variables. If x[i][j] has value Double.NaN, then row *i* of the observations will be skipped and counted as missing.

The number of observation variables is determined in the first call to any of the update methods. In all subsequent update calls, the number of observation variables must be the same.

groups – an int array containing the group number of the observations in x. Group numbers must be numbered 1, 2,..., nGroups. If groups[i] == 0, the row of observations will be skipped and counted as missing. For groups[i] < 0 or groups[i] > nGroups, a warning will be issued indicating that the row of observations will be skipped (not marked as missing).

frequencies – a double array of size x.length containing the frequency for each observation. Each value must be positive. Any Double.NaN value results in that observation being skipped and marked missing.

weights – a double array of size x.length containing the weight for each observation. Each value must be positive. Any Double.NaN value results in that observation being skipped and marked missing.

Example 1

This example computes a pooled variance-covariance matrix.

```
import com.imsl.stat.PooledCovariances;
import com.imsl.math.PrintMatrix;

public class PooledCovariancesEx1 {

    static public void main(String arg[]) {
        double[][] x = {
            {2.2, 5.6}, {3.4, 2.3}, {1.2, 7.8},
            {3.2, 2.1}, {4.1, 1.6}, {3.7, 2.2}
        };
        int[] groups = {1, 1, 1, 2, 2, 2};
        int nGroups = 2;

        PooledCovariances pc = new PooledCovariances(nGroups);
        pc.update(x, groups);

        double covar[][] = pc.getPooledCovariances();
        new PrintMatrix("Pooled Covariances").print(covar);
    }
}
```

Output

```
Pooled Covariances
     0       1
0   0.708  -1.575
1  -1.575   3.883
```

Example 2

This example computes a pooled variance-covariance matrix for the Fisher iris data. The data are not processed as a whole but consecutively in blocks of 10 observations. In each update call, the pooled variance-covariance matrix is internally updated with the new block of observations. The final matrix and the group variable means are printed.

```
import com.imsl.stat.PooledCovariances;
import com.imsl.math.PrintMatrix;

public class PooledCovariancesEx2 {

    static public void main(String arg[]) {
        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}
};

// Fisher Iris dataset contains three groups of 50 observations
// Group 1: Setosa, Group 2: Versicolor, Group 3: Virginica, 50 of each
int nGroups = 3;
int[] groups = new int[x.length];
for (int i = 0; i < 50; i++) {
    groups[i] = 1;
    groups[i + 50] = 2;
    groups[i + 100] = 3;
}

int blockGroups[] = new int[10];
int nrows = blockGroups.length, ncols = x[0].length;
double[][] blockObservations = new double[nrows][ncols];

// Create PooledCovariances object
PooledCovariances pc = new PooledCovariances(nGroups);

// Add 10 consecutive observations to the pooled Covariance matrix at
// a time
for (int i = 0; i < 15; i++) {
```

```
            for (int j = 0; j < nrows; j++) {
                blockGroups[j] = groups[i * 10 + j];
                System.arraycopy(x[i * 10 + j], 0, blockObservations[j], 0, 4);
            }
            // Update pooled covariance matrix with the new observations
            pc.update(blockObservations, blockGroups);
        }

        double covar[][] = pc.getPooledCovariances();
        new PrintMatrix("Pooled Covariances").print(covar);
        new PrintMatrix("Means").print(pc.getMeans());
        System.out.println("Total number of observations: "
                + pc.getTotalNumberOfObservations());
    }
}
```

Output

```
      Pooled Covariances
     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

             Means
     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

Total number of observations: 150
```

NormOneSample class

`public class com.imsl.stat.NormOneSample implements Serializable, Cloneable`

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 χ^2 distribution with *n - 1* degrees of freedom. The method `getChiSquaredTestDF` returns the degrees of freedom.

Constructor

NormOneSample

```
public NormOneSample(double[] x)
```

Description

Constructor to compute statistics for mean and variance inferences using a sample from a normal population.

Parameter

`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 ω_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 ω_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 ω_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.

Parameter

`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.

Parameter

`confidenceMean` – `double` containing the confidence level of the mean.

`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* less than 50 percent, set `confidenceMean` =1.0-2.0*(*c*/100).

This effectively gives the one-sided confidence interval for both *c*% and (100-*c*)%. For example, for a one-sided t-test with confidence level of 40, set `confidenceMean` =.2 . This means that 40% of the distribution is lower than confidence limit for the mean (getLowerCIMean) and 40% of the distribution is greater than the upper confidence limit for the mean (see getUpperCIMean). It also means that 60% of the distribution is greater than the lower confidence limit for the mean and 60% is lower than upper confidence limit for the mean. If the confidence mean is not specified, a 95-percent confidence interval is computed.

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.

Parameter

`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.

Parameter

`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, t, pvalue;
        int df;
        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 = 3.2147173398634027E-9

CI variance is 1.3360926049992239    6.199863467239491
Chi-squared Test results
Chi-squared df = 14
Chi-squared t = 34.89733333333332
Chi-squared pvalue = 0.0015223176141821789
```

NormTwoSample class

`public class com.imsl.stat.NormTwoSample implements Serializable, Cloneable`

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`. Missing values, that is, values equal to NaN (not a number), are excluded from the computations. For inferences concerning parameters of a single normal population, see class `NormOneSample`.

Let μ_1 and σ_1^2 be the mean and variance of the first population, and let μ_2 and σ_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

$$s_1^2 = \sum (x_{1i} - \bar{x}_1)^2 / (n_1 - 1), \quad s_2^2 = \sum (x_{2i} - \bar{x}_2)^2 / (n_2 - 1)$$

Inferences about the Means

The test that the difference in means equals a certain value, for example, μ_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:

$$s^2 = \frac{(n_1 - 1)\, s_1 + (n_2 - 1)\, s_2}{n_1 + n_2 - 2}$$

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

$$t' = (\bar{x}_1 - \bar{x}_2 - \mu_0)\,/\,s_d$$

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:

$$\text{df} = \frac{s_d^4}{\frac{\left(s_1^2/n_1\right)^2}{n_1-1} + \frac{\left(s_2^2/n_2\right)^2}{n_2-1}}$$

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.

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.

Parameter

`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.

Parameter

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 ω_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 ω_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 ω_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

an `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

an `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.

Parameter

`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 $\text{confidenceMean} = 1.0 - 2.0(1.0 - c)$. If the confidence mean is not specified, a 95-percent confidence interval is computed. Default: confidenceMean = .95

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

This method updates the test results to include a new subset of the data. This is useful when the data is too large to fit into memory or when all of the data is not available at one time or location.

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.

This method updates the test results to include a new subset of the data. This is useful when the data is too large to fit into memory or when all of the data is not available at one time or location.

Parameter

`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.

This method updates the test results to include a new subset of the data. This is useful when the data is too large to fit into memory or when all of the data is not available at one time or location.

Parameter

`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 $\mu_1 \le \mu_2$ 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 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);
        double 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
```

Example 2: NormTwoSample

The same data is used for this example as for the initial example.

This example demonstrates how the analysis can be applied to subsets of the original data sets using different update methods. These techniques may be useful when analyzing data sets too large to fit into memory.

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

public class NormTwoSampleEx2 {

    public static void main(String args[]) {
        // Bring in first group of observations on x1 and x2.
        // 2 observations first and then 1 observation.
        double[] x1blk = {72.0, 75.0};
        double[] x2blk = {111.0, 118.0};
        NormTwoSample n2samp = new NormTwoSample(x1blk, x2blk);

        // Use different update methods.
        double[] x1blk1 = {77.0};
        double[] x2blk1 = {128.0};
        n2samp.updateX(x1blk1);
        n2samp.updateY(x2blk1);

        // Second group.
        double[] x1blk2 = {80.0, 104.0, 110.0, 125.0};
        double[] x2blk2 = {138.0, 140.0, 150.0, 163.0};
        n2samp.update(x1blk2, x2blk2);

        // Third group.
        double[] x2blk3 = {164.0, 169.0};
        n2samp.updateY(x2blk3);

        double 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
```

Sort class

`public class com.imsl.stat.Sort`

A collection of sorting functions.

Class `Sort` contains ascending and descending methods for sorting elements of an array or a matrix.

The QuickSort algorithm is used, except for short sequences that are handled using an insertion sort.

The QuickSort algorithm is a randomized QuickSort with 3-way partitioning. Basic QuickSort is slow if the sequence to be sorted contains many duplicate keys. The 3-way partitioning algorithm eliminates this problem. The pivot is chosen as the middle element of three potential pivots chosen at random.

The matrix ascending method sorts the rows of real matrix `x` using a particular column or columns 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 $i = 0, 1, \ldots, \text{nObs} - 2, \text{x}[i][\text{indKeys }[0]] \leq \text{x}[i+1][\text{indKeys}[0]]$, where nObs is the number of observations.
- For $k = 1, \ldots, \text{nKeys} - 1$, if $\text{x[i][indKeys[j]]} = \text{x[i+1][indKeys[j]]}$ for $j = 0, 1, \ldots, k-1$, then $\text{x}[i][\text{indKeys}[k]] = \text{x}[i+1][\text{indKeys}[k]]$

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`.

If all of the sort keys in a pair of rows are equal then the rows keep their original relative order.

The sorting algorithm is based on a quicksort method given by Singleton (1969) with modifications, see Bentley and Sedgewick (1997).

Methods

ascending

`static public void ascending(double[] ra)`

Description

Sorts an array into ascending order.

Parameter

`ra` – a `double` array to be sorted into ascending order

ascending

`static public void ascending(int[] ia)`

Description

Sorts an integer array into ascending order.

Parameter

`ia` – an `int` array to be sorted into ascending order

ascending

`static public void ascending(double[] ra, int[] iPerm)`

Description

Sorts an array into ascending order and returns the permutation vector.

Parameters

ra – a double array to be sorted into ascending order

iPerm – an int array that on input is the same length as ra and contains the values 0, 1, ... On output, the values represent the permutation of the values in ra, that is, the re-ordering of those values in terms of their previous position.

ascending

```
static public void ascending(double[][] ra, int nKeys)
```

Description

Sorts a matrix into ascending order by the first nKeys.

Parameters

ra – a double matrix to be sorted into ascending order

nKeys – an int, the number of keys. The first nKeys columns of ra are to be used as the sorting keys.

ascending

```
static public void ascending(double[][] ra, int[] indKeys)
```

Description

Sorts a matrix into ascending order by specified keys.

Parameters

ra – a double matrix to be sorted into ascending order

indKeys – an int array containing the order into which the columns of ra are to be sorted. These values must be between 0 and one less than the number of columns in ra.

ascending

```
static public void ascending(int[] ia, int[] iPerm)
```

Description

Sorts an integer array into ascending order and returns the permutation vector.

Parameters

ia – an int array to be sorted into ascending order

iPerm – an int array that on input is the same length as ia and contains the values 0, 1, ... On output, the values represent the permutation of the values in ia, that is, the re-ordering of those values in terms of their previous position.

ascending

```
static public void ascending(int[][] ia, int nKeys)
```

Description

Sorts a matrix into ascending order by the first nKeys.

Parameters

`ia` – an `int` matrix to be sorted into ascending order

`nKeys` – an `int`, the number of keys. The first `nKeys` columns of `ia` are to be used as the sorting keys.

ascending

```
static public void ascending(double[][] ra, int nKeys, int[] iPerm)
```

Description

Sorts a matrix into ascending order according to the first `nKeys` keys and returns the permutation vector.

Parameters

`ra` – a `double` matrix to be sorted into ascending order

`nKeys` – an `int`, the number of keys. The first `nKeys` columns of `ra` are to be used as the sorting keys.

`iPerm` – an `int` array that on input is the same length as `ra` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ra`, that is, the re-ordering of those values in terms of their previous position.

ascending

```
static public void ascending(double[][] ra, int[] indKeys, int[] iPerm)
```

Description

Sorts a matrix into ascending order by specified keys and returns the permutation vector.

Parameters

`ra` – a `double` matrix to be sorted into ascending order

`indKeys` – an `int` array containing the order into which the columns of `ra` are to be sorted. These values must be between 0 and one less than the number of columns in `ra`.

`iPerm` – an `int` array that on input is the same length as `ra` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ra`, that is, the re-ordering of those values in terms of their previous position.

ascending

```
static public void ascending(int[][] ia, int nKeys, int[] iPerm)
```

Description

Sorts a matrix into ascending order according to the first `nKeys` keys and returns the permutation vector.

Parameters

`ia` – an `int` matrix to be sorted into ascending order

`nKeys` – an `int`, the number of keys. The first `nKeys` columns of `ia` are to be used as the sorting keys.

`iPerm` – an `int` array that on input is the same length as `ia` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ia`, that is, the re-ordering of those values in terms of their previous position.

ascending

```
static public void ascending(int[][] ia, int[] indKeys, int[] iPerm)
```

Description

Sorts a matrix into ascending order by specified keys and returns the permutation vector.

Parameters

`ia` – an `int` matrix to be sorted into ascending order

`indKeys` – an `int` array containing the order into which the columns of `ia` are to be sorted. These values must be between 0 and one less than the number of columns in `ia`.

`iPerm` – an `int` array that on input is the same length as `ia` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ia`, that is, the re-ordering of those values in terms of their previous position.

descending

```
static public void descending(double[] ra)
```

Description

Sorts an array into descending order.

Parameter

`ra` – a `double` array to be sorted into descending order

descending

```
static public void descending(int[] ra)
```

Description

Sorts an integer array into descending order.

Parameter

`ra` – an `int` array to be sorted into descending order

descending

```
static public void descending(double[] ra, int[] iPerm)
```

Description

Sorts an array into descending order and returns the permutation vector.

Parameters

`ra` – a `double` array to be sorted into descending order

`iPerm` – an `int` array that on input is the same length as `ra` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ra`, that is, the re-ordering of those values in terms of their previous position.

descending

```
static public void descending(double[][] ra, int nKeys)
```

Description

Sorts a matrix into descending order by the first `nkeys`.

Parameters

`ra` – a `double` matrix to be sorted into descending order

`nKeys` – an `int`, the number of keys. The first `nKeys` columns of `ra` are to be used as the sorting keys.

descending

```
static public void descending(double[][] ra, int[] indKeys)
```

Description

Sorts a matrix into descending order by specified keys.

Parameters

`ra` – a `double` matrix to be sorted into descending order

`indKeys` – an `int` array containing the order into which the columns of `ra` are to be sorted. These values must be between 0 and one less than the number of columns in `ra`.

descending

```
static public void descending(int[] ra, int[] iPerm)
```

Description

Sorts an integer array into descending order and returns the permutation vector.

Parameters

`ra` – an `int` array to be sorted into descending order

`iPerm` – an `int` array that on input is the same length as `ra` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ra`, that is, the re-ordering of those values in terms of their previous position.

descending

```
static public void descending(double[][] ra, int nKeys, int[] iPerm)
```

Description

Sorts a matrix into descending order by the first `nkeys` and returns the permutation vector.

Parameters

`ra` – a `double` matrix to be sorted into descending order

`nKeys` – an `int`, the number of keys. The first `nKeys` columns of `ra` are to be used as the sorting keys.

`iPerm` – an `int` array that on input is the same length as `ra` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ra`, that is, the re-ordering of those values in terms of their previous position.

descending

```
static public void descending(double[][] ra, int[] indKeys, int[] iPerm)
```

Description

Sorts a matrix into descending order by specified keys and return the permutation vector.

Parameters

`ra` – a `double` matrix to be sorted into descending order

`indKeys` – an `int` array containing the order into which the columns of `ra` are to be sorted. These values must be between 0 and one less than the number of columns in `ra`.

`iPerm` – an `int` array that on input is the same length as `ra` and contains the values 0, 1, ... On output, the values represent the permutation of the values in `ra`, that is, the re-ordering of those values in terms of their previous position.

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[] = new int[x.length];
        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  3
1  1  4
1  1  8
1  1  9
1  2  6
2  1  2
2  2  9
2  ?  5
?  2  7

The permutation array

0
2
3
7
9
5
1
8
4
6
```

Ranks class

`public class com.imsl.stat.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
if *TieBreaker* = 1, *Ranks[i]* = *Ranks[j]* = 2.0
if *TieBreaker* = 2, *Ranks[i]* = *Ranks[j]* = 1.0
if *TieBreaker* = 3, *Ranks[i]* = 1.0 and *Ranks[j]* = 2.0
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.

Fields

TIE_AVERAGE

```
static final public int TIE_AVERAGE
```

In case of ties, use the average of the scores of the tied observations.

TIE_HIGHEST

```
static final public int TIE_HIGHEST
```

In case of ties, use the highest score in the group of ties.

TIE_LOWEST

```
static final public int TIE_LOWEST
```

In case of ties, use the lowest score in the group of ties.

TIE_RANDOM

```
static final public 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

```
static public 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.

Parameter

- `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).

Parameter

- `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.

Parameter

`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).

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`fuzz` – a `double` which represents the fuzz factor

setRandom

`public void setRandom(Random random)`

Description

Sets the `Random` object.

Parameter

`random` – a `Random` object used in breaking ties

setTieBreaker

`public void setTieBreaker(int iTie)`

Description

Sets the tie breaker for Ranks.

Parameter

`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
-0.865
-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
-0.753
 0.989
 0.287
-0.46
```

```
-0.122
 1.849
 0.865
-0.552
 0.372
-0.649
 0.552
```

EmpiricalQuantiles class

`public class com.imsl.stat.EmpiricalQuantiles implements Serializable, Cloneable`

Computes empirical quantiles.

The class EmpiricalQuantiles determines the empirical quantiles, as indicated in the array `qProp`, from the data in `x`. The algorithm first checks to see if `x` is sorted; if `x` is not sorted, the algorithm does either a complete or partial sort, depending on how many order statistics are required to compute the quantiles requested. The algorithm returns the empirical quantiles and, for each quantile, the two order statistics from the sample that are at least as large and at least as small as the quantile. For a sample of size n, the quantile corresponding to the proportion p is defined as

$$Q(p) = (1-f)x_{(j)} + fx_{(j+1)}$$

where $j = \lfloor p(n+1) \rfloor$, $f = p(n+1) - j$, and $x_{(j)}$, is the j-th order statistic, if $1 \leq j \leq n$; otherwise, the empirical quantile is the smallest or largest order statistic.

Constructor

EmpiricalQuantiles

`public EmpiricalQuantiles(double[] x, double[] qProp)`

Description

Constructor for `EmpiricalQuantiles`.

Parameters

`x` – A `double` array containing the data.

`qProp` – A `double` array containing the quantile proportions.

Methods

getQ

```
final public double[] getQ()
```

Description

Returns the empirical quantiles.

Returns

A `double` array containing the empirical quantiles. `Q[i]` corresponds to the empirical quantile at proportion `qProp[i]`. The quantiles are determined by linear interpolation between adjacent ordered sample values.

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 `x`.

getXHi

```
final public double[] getXHi()
```

Description

Returns the smallest element of `x` greater than or equal to the desired quantile.

Returns

A `double` array containing the smallest element of `x` greater than or equal to the desired quantile.

getXLo

```
final public double[] getXLo()
```

Description

Returns the largest element of `x` less than or equal to the desired quantile.

Returns

A `double` array containing the largest element of `x` less than or equal to the desired quantile.

Example 1: Empirical Quantiles

In this example, five empirical quantiles from a sample of size 30 are obtained. Notice that the 0.5 quantile corresponds to the sample median. The data are from Hinkley (1977) and Velleman and Hoaglin (1981). They are the measurements (in inches) of precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

```
import java.text.*;
import com.imsl.stat.*;

public class EmpiricalQuantilesEx1 {

    public static void main(String args[]) {
        String fmt = "0.00";
        DecimalFormat df = new DecimalFormat(fmt);

        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
        };
        double[] qProp = {0.01, 0.5, 0.90, 0.95, 0.99};

        EmpiricalQuantiles eq = new EmpiricalQuantiles(x, qProp);
        double[] Q = eq.getQ();
        double[] XLo = eq.getXLo();
        double[] XHi = eq.getXHi();
        System.out.println("               Smaller        "
                + "Empirical        Larger");
        System.out.println(" Quantile       Datum          "
                + "Quantile         Datum");
        for (int i = 0; i < qProp.length; i++) {
            System.out.println(df.format(qProp[i]) + "           "
                    + df.format(XLo[i]) + "             " + df.format(Q[i])
                    + "           " + df.format(XHi[i]));
        }
    }
}
```

Output

```
               Smaller        Empirical        Larger
 Quantile       Datum          Quantile         Datum
0.01           0.32             0.32           0.32
0.50           1.43             1.47           1.51
0.90           3.00             3.08           3.09
0.95           3.37             3.99           4.75
0.99           4.75             4.75           4.75
```

EmpiricalQuantiles.ScaleFactorZeroException class

`static public class com.imsl.stat.EmpiricalQuantiles.ScaleFactorZeroException extends com.imsl.IMSLException`

The computations cannot continue because a scale factor is zero.

Constructor

EmpiricalQuantiles.ScaleFactorZeroException

`public EmpiricalQuantiles.ScaleFactorZeroException(int index)`

Description

Constructs a `ScaleFactorZeroException`.

Parameter

`index` – An `int` which specifies the index of the scale factor array at which scale factor is zero.

TableOneWay class

`public class com.imsl.stat.TableOneWay implements Serializable, Cloneable`

Class `TableOneWay` calculates a frequency table for a data array. A one-way frequency table can be used to visualize the shape of the data distribution and look for anomalies in the data. There are many approaches to constructing frequency tables. Four approaches are implemented in this class:

1. equal width class intervals based upon the smallest and largest observations,
2. equal width class intervals based upon a user provided minimum and maximum,
3. class intervals defined from user provided class midpoints, and
4. class intervals defined from user provided class boundaries.

The `TableOneWay` class implements the first two approaches by overloading the `getFrequencyTable` method. If `getFrequencyTable()` is used without input arguments, `nIntervals` of equal length are formed between the minimum and maximum values in the data. The frequency table returned from this

method contains tallies of the number of observations in each interval. The data minimum and maximum can be obtained using `getMinimum` and `getMaximum`.

Instead of using the minimum and maximum to define the boundaries of the smallest and largest classes, specified boundaries can be used by calling `getFrequencyTable(lower_bound, upper_bound)`. This method tallies all data less than or equal to the `lower_bound` into the first class, and all data greater than or equal to `upper_bound` into the last class.

The third approach is implemented using the `getFrequencyTableUsingClassmarks` method. Equally spaced intervals can be defined using class marks. In this approach a double precision array of length nIntervals containing the class midpoints is passed to the `getFrequencyTableUsingClassmarks(classmarks[])`. The class marks, or midpoints, must be equally spaced.

Finally in those applications where unequal length intervals are preferred, the `getFrequencyTableUsingCutpoints(cutpoints[])` method can be used. The double precision array `cutpoints` has length `nIntervals-1` and contains the class boundaries listed in ascending order. The first cut point defines the first class which is used to tally all data less than or equal to the first cut point value. The last cut point defines the last class which is used to tally all data greater than or equal to the last cut point value.

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.

Parameter

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.

Parameter

`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
```

TableTwoWay class

```
public class com.imsl.stat.TableTwoWay implements Serializable, Cloneable
```

Class `TableTwoWay` calculates a two-dimensional frequency table for a data array based upon two variables.

A two-way frequency table can be used to visualize the shape of the bivariate distribution and look for anomalies in the data. There are many approaches to constructing two-way frequency tables. Four approaches are implemented in this class:

1. equal width class intervals based upon the smallest and largest observations,
2. equal width class intervals based upon a user provided minimum and maximum,
3. class intervals defined from user provided class midpoints, and
4. class intervals defined from user provided class boundaries.

The `TableTwoWay` class implements the first two approaches by overloading the getFrequencyTable method. If `getFrequencyTable()` is used without input arguments, `xIntervals` intervals of equal length are formed between the minimum and maximum values in x, and similarly, `yIntervals` intervals are formed for y. The frequency table returned from this method contains tallies of the number of observations in each interval. The data minimum and maximum can be obtained using `getMinimumX`, `getMinimumY`, `getMaximumX` and `getMaximumY`.

Instead of using the minimum and maximum to define the boundaries of the smallest and largest classes, specified boundaries can be used by calling `getFrequencyTable(xLowerBound, xUpperBound, yLowerBound, yUpperBound))`. This method tallies all data less than or equal to the `xLowerBound` and `yLowerBound` into the first class, and all data greater than or equal to `xUpperBound` and `YUpperBround` into the last class

The third approach is implemented using the `getFrequencyTableUsingClassmarks` method. Equally spaced intervals can be defined using class marks. In this approach two double precision arrays of length `xIntervals` and `yIntervals` containing the class midpoints for x and y respectively are passed to the `getFrequencyTableUsingClassmarks(cx[], cy[])`. The class marks, or midpoints, must be equally spaced.

Finally in those applications where unequal length intervals are preferred, the `getFrequencyTableUsingCutpoints(cx[], cy[])` method can be used. The double precision arrays `cx` and `cy` with lengths `xIntervals-1` and `yIntervals-1` respectively contain the class boundaries listed in ascending order. The first cut point defines the first class which is used to tally all data less than or equal to the first cut point value. The last cut point defines the last class which is used to tally all data greater than or equal to the last cut point value.

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 class marks. 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 the class marks for `x`.

`cy` – `double` array containing 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 the cutpoints for `x`.

`cy` – `double` array containing the cutpoints 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, 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
```

TableMultiWay class

```
public class com.imsl.stat.TableMultiWay implements Serializable, Cloneable
```

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.

Constructors

TableMultiWay

```
public TableMultiWay(double[][] x, int nKeys)
```

Description

Constructor for TableMultiWay.

Parameters

x – A double matrix containing the observations and variables.

nKeys – int equal to the number of keys. The first nKeys variables (columns) of x are to be used as the sorting keys.

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) which are to be used as the sorting keys.

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)`

Description

Sets the frequencies for each observation in x. The length of the input must be the same as the number of observations or number of rows in `x`.

Parameter

`frequencies` – a `double` array containing the frequency for each observation in x. Default `frequencies`[] =1.

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] + "  ");
        }
        System.out.println();
    }
}
```

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.*;

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);

        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
```

TableMultiWay.BalancedTable class

`public class com.imsl.stat.TableMultiWay.BalancedTable`

Tallies the number of unique values of each variable.

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.

TableMultiWay.UnbalancedTable class

`public class com.imsl.stat.TableMultiWay.UnbalancedTable`

Tallies the frequency of each cell in `x`.

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.

Chapter 14: Regression

Types

class RegressorsForGLM 698
class LinearRegression 708
class NonlinearRegression 720
class UserBasisRegression 736
interface RegressionBasis 741
class SelectionRegression 741
class StepwiseRegression 756

Usage Notes

The regression models in this chapter include the simple and multiple linear regression models, the multivariate general linear model, and the nonlinear regression model. Functions for fitting regression models, computing summary statistics from a fitted regression, computing diagnostics, and computing confidence intervals for individual cases are provided. This chapter also provides methods for building a model from a set of candidate variables.

Simple and Multiple Linear Regression

The simple linear regression model is

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_1 \; i = 1, 2, \ldots, n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable, the x_i's are the settings of the independent (explanatory) variable, β_0 and β_1 are the intercept and slope parameters (respectively) and the ε_1's are independently distributed normal errors, each with mean 0 and variance σ^2.

The multiple linear regression model is

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \varepsilon_1 \; i = 1, 2, \ldots, n$$

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,..., x_{ik}'s are the settings of the k independent (explanatory) variables; $\beta_0, \beta_1, \ldots, \beta_k$ are the regression coefficients; and the ε_1's are independently distributed normal errors, each with mean 0 and variance σ^2.

The class `LinearRegression` fits both the simple and multiple linear regression models using a fast Given's transformation and includes an option for excluding the intercept β_0. The responses are input in array `y`, and the independent variables are input in array `x`, where the individual cases correspond to the rows and the variables correspond to the columns.

After the model has been fitted using the `LinearRegression` class, member "get" methods such as `getCoefficientTTests()` can be used to retrieve summary statistics. Predicted values, confidence intervals, and case statistics for the fitted model can be obtained from inner class `LinearRegression.CaseStatistics`.

No Intercept Model

Several functions provide the option for excluding the intercept from a model. In most practical applications, the intercept should be included in the model. For functions that use the sums of squares and crossproducts matrix as input, the no-intercept case can be handled by using the raw sums of squares and crossproducts matrix as input in place of the corrected sums of squares and crossproducts. The raw sums of squares and crossproducts matrix can be computed as $(x_1, x_2, \ldots, x_k, y)^T (x_1, x_2, \ldots, x_k, y)$.

Specification of X for the General Linear Model

Variables used in the general linear model are either continuous or classification variables. Typically, multiple regression models use continuous variables, whereas analysis of variance models use classification variables. Although the notation used to specify analysis of variance models and multiple regression models may look quite different, the models are essentially the same. The term *general linear model* emphasizes that a common notational scheme is used for specifying a model that may contain both continuous and classification variables.

A general linear model is specified by its effects (sources of variation). An effect is referred to in this text as a single variable or a product of variables. (The term *effect* is often used in a narrower sense, referring only to a single regression coefficient.) In particular, an *effect* is composed of one of the following:

1. a single continuous variable
2. a single classification variable
3. several different classification variables
4. several continuous variables, some of which may be the same
5. continuous variables, some of which may be the same, and classification variables, which must be distinct

Effects of the first type are common in multiple regression models. Effects of the second type appear as main effects in analysis of variance models. Effects of the third type appear as interactions in analysis of variance models. Effects of the fourth type appear in polynomial models and response surface models as powers and crossproducts of some basic variables. Effects of the fifth type appear in one-way analysis of

covariance models as regression coefficients that indicate lack of parallelism of a regression function across the groups.

The analysis of a general linear model occurs in two stages. The first stage calls class `RegressorsForGLM` to specify all regressors except the intercept. The second stage uses `LinearRegression`, at which point the model will be specified as either having or not having an intercept.

Suppose the data matrix has as its first four columns two continuous variables in Columns 0 and 1 and two classification variables in Columns 2 and 3. The data might appear as follows:

Column 0	Column 1	Column 2	Column 3
11.23	1.23	1.0	5.0
12.12	2.34	1.0	4.0
12.34	1.23	1.0	4.0
4.34	2.21	1.0	5.0
5.67	4.31	2.0	4.0
4.12	5.34	2.0	1.0
4.89	9.31	2.0	1.0
9.12	3.71	2.0	1.0

Each distinct value of a classification variable determines a level. The classification variable in Column 2 has two levels. The classification variable in Column 3 has three levels. (Integer values are recommended, but not required, for values of the classification variables. The values of the classification variables corresponding to the same level must be identical.) Some examples of regression functions and their specifications are as follows:

	Intercept	Class Columns
$\beta_0 + \beta_1 x_1$	`true`	{}
$\beta_0 + \beta_1 x_1 + \beta_2 x_1^2$	`true`	{}
$\mu + \alpha_i$	`true`	{2}
$\mu + \alpha_i + \beta_j + \gamma_{ij}$	`true`	{2, 3}
μ_{ij}	`false`	{2, 3}
$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$	`true`	{}

	Effects
$\beta_0 + \beta_1 x_1$	{ {0} }
$\beta_0 + \beta_1 x_1 + \beta_2 x_1^2$	{{0}, {0,0}}
$\mu + \alpha_i$	{ {2} }
$\mu + \alpha_i + \beta_j + \gamma_{ij}$	{{2}, {3}, {2, 3}}
μ_{ij}	{ {2, 3} }
$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2$	{{0}, {1}, {0, 1}}
$\mu + \alpha_i + \beta_j + \gamma_{ij}$	{{2}, {0}, {0, 2}}

Variable Selection

Variable selection can be performed by `SelectionRegression`, which computes all best-subset

regressions, or by `StepwiseRegression` , which computes stepwise regression. The method used by `SelectionRegression` is generally preferred over that used by `StepwiseRegression` because `SelectionRegression` implicitly examines all possible models in the search for a model that optimizes some criterion while stepwise does not examine all possible models. However, the computer time and memory requirements for `SelectionRegression` can be much greater than that for `StepwiseRegression` when the number of candidate variables is large.

Nonlinear Regression Model

The nonlinear regression model is

$$y_i = f(x_i; \theta) + \varepsilon_i \; i = 1, 2, \ldots, n$$

where the observed values of the y_i's constitute the responses or values of the dependent variable, the x_i's are the known vectors of values of the independent (explanatory) variables, f is a known function of an unknown regression parameter vector θ, and the ε_i's are independently distributed normal errors each with mean 0 and variance σ^2.

Class `NonlinearRegression` performs the least-squares fit to the data for this model.

Weighted Least Squares

Classes throughout the chapter generally allow weights to be assigned to the observations. A `weight` argument is used throughout to specify the weighting for particular rows of X.

Computations that relate to statistical inference-e.g., t tests, F tests, and confidence intervals-are based on the multiple regression model except that the variance of ε_i is assumed to equal σ^2 times the reciprocal of the corresponding weight.

If a single row of the data matrix corresponds to n_i observations, the vector `frequencies` can be used to specify the frequency for each row of X. Degrees of freedom for error are affected by frequencies but are unaffected by weights.

Summary Statistics

Methods `LinearRegression.getANOVA()`, `LinearRegression.getCoefficientTTests()`, `NonlinearRegression.getR()` and `StepwiseRegression.getCoefficientVIF()` can be used to compute statistics related to a regression for each of the dependent variables fitted by the indicated regression. The summary statistics include the model analysis of variance table, sequential sums of squares and F-statistics, coefficient estimates, estimated standard errors, t-statistics, variance inflation factors and estimated variance-covariance matrix of the estimated regression coefficients.

The summary statistics are computed under the model $y = X\beta + \varepsilon$, where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors with rank (X) = r, is the $p \times 1$ vector of regression coefficients, and $\varepsilon$is the $n \times 1$ vector of errors whose elements are independently normally distributed with mean 0 and variance σ^2/w_i.

Given the results of a weighted least-squares fit of this model (with the w_i's as the weights), most of the computed summary statistics are output in the following variables:

`ANOVA` Class

The `getANOVA()` methods in several of the regression classes return an `ANOVA` object. Summary statistics can be retrieved via specific "get" methods or the `ANOVA.getArray()` method. This returns a one-dimensional array. In `StepwiseRegression`, `ANOVA.getArray()` returns `Double.NaN` for the last two elements of the array because they cannot be computed from the input. The array contains statistics related to the analysis of variance. The sources of variation examined are the regression, error, and total. The first 10 elements of the `ANOVA.getArray()` and the notation frequently used for these is described in the following table (here, `AOV = ANOVA.getArray()`):

Model Analysis of Variance Table

Variation Src.	Deg. of Freedom	Sum of Squares	Mean Square	F	p-value
Regression	DFR = AOV[0]	SSR = AOV[3]	MSR = AOV[6]	`AOV[8]`	`AOV[9]`
Error	DFE = AOV[1]	SSE = AOV[4]	s^2 = AOV[7]		
Total	DFT = AOV[2]	SST = AOV[5]			

If the model has an intercept (default), the total sum of squares is the sum of squares of the deviations of y_i from its (weighted) mean $\bar{y}$–the so-called *corrected total sum of squares*, denoted by the following:

$$\text{SST} = \sum_{i=1}^{n} w_i (y_i - \bar{y})^2$$

If the model does not have an intercept (`hasIntercept = false`), the total sum of squares is the sum of squares of y_i-the so-called *uncorrected total sum of squares*, denoted by the following:

$$\text{SST} = \sum_{i=1}^{n} w_i y_i^2$$

The error sum of squares is given as follows:

$$\text{SSE} = \sum_{i=1}^{n} w_i \left(y_i - \hat{y}_i\right)^2$$

The error degrees of freedom is defined by $\text{DFE} = n - r$.

The estimate of σ^2 is given by $s^2 = \text{SSE}/\text{DFE}$, which is the error mean square.

The computed F statistic for the null hypothesis, $H_0 : \beta_1 = \beta_2 = \ldots \beta_k = 0$, versus the alternative that at least one coefficient is nonzero is given by $F = s^2 = \text{MSR}/s^2$. The p-value associated with the test is the probability of an F larger than that computed under the assumption of the model and the null hypothesis. A small p-value (less than 0.05) is customarily used to indicate there is sufficient evidence from the data to reject the null hypothesis.

The remaining five elements in `AOV` frequently are displayed together with the actual analysis of variance table. The quantities R-squared ($R^2 = \text{AOV}[10]$) and adjusted R-squared

$$R_a^2 = (\text{AOV}[11])$$

are expressed as a percentage and are defined as follows:

$$R^2 = 100\,(\mathrm{SSR/SST}) = 100\,(1 - \mathrm{SSE/SST})$$

$$R_a^2 = 100 \max\left\{0, 1 - \frac{s^2}{\mathrm{SST/DFT}}\right\}$$

The square root of s^2 ($s = \mathrm{AOV}[12]$) is frequently referred to as the estimated standard deviation of the model error.

The overall mean of the responses $\bar{y}$ is output in `AOV[13]`.

The coefficient of variation ($CV = \mathrm{AOV}[14]$) is expressed as a percentage and defined by $\mathrm{CV} = 100s/\bar{y}$.

`LinearRegression.CoefficientTTests`

A nested class within the `LinnearRegression` and `StepwiseRegression` classes. The statistics (estimated standard error, t statistic and p-value) associated with each coefficient can be retrieved via associated "get" methods.

`getR()`

Estimated variance-covariance matrix of the estimated regression coefficients.

Diagnostics for Individual Cases

Diagnostics for individual cases (observations) are computed by the `LinearRegression.CaseStatistics` class for linear regression.

Statistics computed include predicted values, confidence intervals, and diagnostics for detecting outliers and cases that greatly influence the fitted regression.

The diagnostics are computed under the model $y = X\beta + \varepsilon$, where y is the $n \times 1$ vector of responses, X is the $n \times p$ matrix of regressors with rank(X) = r, β is the $p \times 1$ vector of regression coefficients, and ε is the $n \times 1$ vector of errors whose elements are independently normally distributed with mean 0 and variance ϕ^2/w_i.

Given the results of a weighted least-squares fit of this model (with the w_i's as the weights), the following five diagnostics are computed:

1. leverage
2. standardized residual
3. jackknife residual
4. Cook's distance
5. DFFITS

The definition of these terms is given in the discussion that follows:Let x_i be a column vector containing the elements of the i-th row of X. A case can be unusual either because of x_i or because of the response y_i. The leverage h_i is a measure of uniqueness of the x_i. The leverage is defined by

$$h_i = [x_i^T \left(X^T W X\right)^- x_i] w_i$$

where $W = \text{diag}\left(w_1, w_2 \ldots, w_n\right)$ and $\left(X^T W T\right)^-$ denotes a generalized inverse of $X^T W T$. The average value of the h_i's is r/n. Regression functions declare x_i unusual if $h_i > 2r/n$. Hoaglin and Welsch (1978) call a data point highly influential (i.e., a leverage point) when this occurs.

Let e_i denote the residual

$$y_i - \hat{y}_i$$

for the *i*-th case. The estimated variance of e_i is $(1 - h_i) s^2 w_i$, where s^2 is the residual mean square from the fitted regression. The *i*-th *standardized residual* (also called the internally studentized residual) is by definition

$$r_i = e_i \sqrt{\frac{w_i}{s^2 \left(1 - h_i\right)}}$$

and r_i follows an approximate standard normal distribution in large samples.

The *i*-th *jackknife residual or deleted residual* involves the difference between y_i and its predicted value, based on the data set in which the *i*-th case is deleted. This difference equals $e_i / \left(1 - h_i\right)$. The jackknife residual is obtained by standardizing this difference. The residual mean square for the regression in which the *i*-th case is deleted is as follows:

$$s_i^2 = \frac{(n - r) s^2 - w_i e_i^2 / \left(1 - h_i\right)}{n - r - 1}$$

The jackknife residual is defined as

$$t_i = e_i \sqrt{\frac{w_i}{s_i^2 \left(1 - h_i\right)}}$$

and t_i follows a t_i distribution with $n - r \times 1$ degrees of freedom.

Cook's distance for the *i*-th case is a measure of how much an individual case affects the estimated regression coefficients. It is given as follows:

$$D_i = \frac{w_i h_i e_i^2}{r s^2 \left(1 - h_i\right)^2}$$

Weisberg (1985) states that if D_i exceeds the 50-th percentile of the *F(r, n - r)* distribution, it should be considered large. (This value is about 1. This statistic does not have an *F* distribution.)

DFFITS, like Cook's distance, is also a measure of influence. For the *i*-th case, DFFITS is computed by the formula below.

$$\mathrm{DFFITS}_i = e_i \sqrt{\frac{w_i h_i}{s_i^2 (1 - h_i)^2}}$$

Hoaglin and Welsch (1978) suggest that DFFITS greater than

$$2\sqrt{r/n}$$

is large.

Transformations

Transformations of the independent variables are sometimes useful in order to satisfy the regression model. The inclusion of squares and crossproducts of the variables

$$\left(x_1, x_2, x_1^2, x_2^2, x_1 x_2\right)$$

is often needed. Logarithms of the independent variables are used also. (See Draper and Smith 1981, pp. 218-222; Box and Tidwell 1962; Atkinson 1985, pp. 177-180; Cook and Weisberg 1982, pp. 78-86.)

When the responses are described by a nonlinear function of the parameters, a transformation of the model equation often can be selected so that the transformed model is linear in the regression parameters. For example, by taking natural logarithms on both sides of the equation, the exponential model

$$y = e^{\beta_0 + \beta_1 x_1} \varepsilon$$

can be transformed to a model that satisfies the linear regression model provided the ε_i's have a log-normal distribution (Draper and Smith, pp. 222-225).

When the responses are nonnormal and their distribution is known, a transformation of the responses can often be selected so that the transformed responses closely satisfy the regression model, assumptions. The square-root transformation for counts with a Poisson distribution and the arc-sine transformation for binomial proportions are common examples (Snedecor and Cochran 1967, pp. 325-330; Draper and Smith, pp. 237-239).

Missing Values

NaN (Not a Number) is the missing value code used by the regression functions. Use field `Double.NaN` to retrieve NaN. Any element of the data matrix that is missing must be set to `Double.NaN`. In fitting regression models, any observation containing `NaN` for the independent, dependent, weight, or frequency variables is omitted from the computation of the regression parameters.

RegressorsForGLM class

```
public class com.imsl.stat.RegressorsForGLM implements Serializable
```

Generates regressors for a general linear model.

Class `RegressorsForGLM` generates regressors for a general linear model from a data matrix. The data matrix can contain classification variables as well as continuous variables. Regressors for effects composed solely of continuous variables are generated as powers and crossproducts. Consider a data matrix containing continuous variables as Columns 3 and 4. The effect indices (3, 3) generate a regressor whose *i*-th value is the square of the *i*-th value in Column 3. The effect indices (3, 4) generates a regressor whose *i*-th value is the product of the *i*-th value in Column 3 with the *i*-th value in Column 4.

Regressors for an effect (source of variation) composed of a single classification variable are generated using indicator variables. Let the classification variable A take on values $a_1, a_2, \ldots, a_n$. From this classification variable, `RegressorsForGLM` creates n indicator variables. For $k = 1, 2, \ldots, n$, we have

$$I_k = \begin{cases} 1 & \text{if } A = a_k \\ 0 & \text{otherwise} \end{cases}$$

For each classification variable, another set of variables is created from the indicator variables. These new variables are called *dummy variables*. Dummy variables are generated from the indicator variables in one of three manners:

1. The dummies are the n indicator variables.
2. The dummies are the first $n-1$ indicator variables.
3. The $n-1$ dummies are defined in terms of the indicator variables so that for balanced data, the usual summation restrictions are imposed on the regression coefficients.

In particular, for dummy method `ALL`, the dummy variables are $A_k = I_k\,(k = 1, 2, \ldots, n)$. For dummy method `LEAVE_OUT_LAST`, the dummy variables are $A_k = I_k\,(k = 1, 2, ..., n-1)$. For dummy method `SUM_TO_ZERO`, the dummy variables are $A_k = I_k - I_n\,(k = 1, 2, \ldots, n-1)$. The regressors generated for an effect composed of a single-classification variable are the associated dummy variables.

Let m_j be the number of dummies generated for the *j*-th classification variable. Suppose there are two classification variables A and B with dummies

$$A_1, A_2, \ldots, A_{m_1}$$

and

$$B_1, B_2, \ldots, B_{m_2}$$

The regressors generated for an effect composed of two classification variables A and B are

$$\begin{aligned} A \otimes B = & \quad (A_1, A_2, \ldots, A_{m_1}) \otimes (B_1, B_2, \ldots, B_{m_2}) \\ = & \quad (A_1B_1, A_1B_2, \ldots, A_1B_{m_2}, A_2, B_1, A_2B_2, \ldots, \\ = & \quad A_2B_{m_2}, \ldots, A_{m_1}, B_1, A_{m_1}, B_2, \ldots, A_{m_1}B_{m_2}) \end{aligned}$$

More generally, the regressors generated for an effect composed of several classification variables and several continuous variables are given by the Kronecker products of variables, where the order of the variables is specified in `setEffects`. Consider a data matrix containing classification variables in Columns 0 and 1 and continuous variables in Columns 2 and 3. Label these four columns A, B, X_1, and X_2. The regressors generated by the effect indices $(0, 1, 2, 2, 3)$ are $A \otimes B \otimes X_1X_1X_2$.

Remarks

Let the data matrix x $= (A, B, X_1)$, where A and B are classification variables and X_1 is a continuous variable. The model containing the effects A, B, AB, X_1, AX_1, BX_1, and ABX_1 is specified by setting nClassVariables=2 in the constructor and calling setEffects(effects), with

```
int effects[][] = { {0}, {1}, {0, 1}, {2}, {0, 2}, {1, 2}, {0, 1, 2} };
```

For this model, suppose that variable A has two levels, A_1 and A_2, and that variable B has three levels, B_1, B_2, and B_3. For each dummy method option, the regressors in their order of appearance in regressors are given below.

dummyMethod	Regressors
ALL	A_1, A_2, B_1, B_2, B_3, A_1B_1,A_1B_2, A_1B_3,A_2B_1,A_2B_2, A_2B_3,$X1$,A_1X_1,A_2X_1, B_1X_1, B_2X_1, B_3X_1, $A_1B_1X_1$, $A_1B_2X_1$, $A_1B_3X_1$, $A_2B_1X_1$, $A_2B_2X_1$, $A_2B_3X_1$
LEAVE_OUT_LAST	A_1, B_1, B_2, A_1B_1, A_1B_2, X_1, A_1X_1, B_1X_1, B_2X_1, $A_1B_1X_1$, $A_1B_2X_1$
SUM_TO_ZERO	A_1-A_2, B_1-B_3, B_2-B_3, $(A_1-A_2)(B_1-B_2)$, $(A_1-A_2)(B_2-B_3)$, X_1, $(A_1-A_2)X_1$, $(B_1-B_3)X_1$, $(B_2-B_3)X_1$, $(A_1-A_2)(B_1-B_2)X_1$, $(A_1-A_2)(B_2-B_3)X_1$

Within a group of regressors corresponding to an interaction effect, the indicator variables composing the regressors vary most rapidly for the last classification variable, next most rapidly for the next to last classification variable, etc.

By default, RegressorsForGLM internally generates values for effects which correspond to a first order model with nEffects = nContinuousVariables + nClassVariables, where nContinuousVariables is the number of continuous variables and nClassVariables is the number of classification variables. The variables then are used to create the regressor variables. The effects are ordered such that the first effect corresponds to the first column of x, the second effect corresponds to the second column of x, etc. A second order model corresponding to the columns (variables) of x is generated if setModelOrder(2) is used.

The effects array for a first or second order model can be obtained by first using setModelOrder followed by getEffects. This array can then be modified and used as the argument to setEffects. This may be an easier way of setting the effects for an almost linear or quadratic model than creating the effects array from scratch.

There are

$$\texttt{nEffects} = \texttt{nClassVariables} + \texttt{nContinuousVariables} + \frac{\texttt{nVar}(\texttt{nVar}-1)}{2}$$

effects, where nVar = nClassVariables+nContinuousVariables. The first nVar effects correspond to the columns of x, such that the first effect corresponds to the first column of x, the second effect corresponds to the second column of x, ..., the nVar-th effect corresponds to the nVar-th column of x (i.e. x[nVar-1]). The next nContinuousVariables effects correspond to squares of the continuous variables. The last $\texttt{nVar}(\texttt{nVar}-1)/2$ effects correspond to the two-variable interactions.

- Let the data matrix $\mathtt{x} = (A, B, X_1)$, where A and B are classification variables and X_1 is a continuous variable. The effects generated and order of appearance is

$$A,\, B,\, X_1,\, X_1^2,\, AB,\, AX_1,\, BX_1$$

- Let the data matrix $\mathtt{x} = (A, X_1, X_2)$, where A is a classification variable and X_1 and X_2 are continuous variables. The effects generated and order of appearance is

$$A,\, X_1,\, X_2,\, X_1^2,\, X_2^2,\, AX_1,\, AX_2,\, X_1X_2$$

- Let the data matrix $\mathtt{x} = (X_1, A, X_2)$, where A is a classification variable and X_1 and X_2 are continuous variables. The effects generated and order of appearance is

$$X_1,\, A,\, X_2,\, X_1^2,\, X_2^2,\, X_1A,\, X_1X_2,\, AX_2$$

Higher-order and more complicated models can be specified using `setEffects`.

Fields

ALL

`static final public int ALL`

The n indicator variables are the dummy variables.

LEAVE_OUT_LAST

`static final public int LEAVE_OUT_LAST`

The dummies are the first *n-1* indicator variables.

SUM_TO_ZERO

`static final public int SUM_TO_ZERO`

The $n-1$ dummies are defined in terms of the indicator variables so that for balanced data, the usual summation restrictions are imposed on the regression coefficients.

Constructors

RegressorsForGLM

`public RegressorsForGLM(double[][] x, int nClassVariables)`

Description

Constructor where the class columns are the first columns.

Parameters

`x` – is an `nObservations` by `nClassVariables+nContinuousVariables` array containing the data, where `nObservations` is the number of observations. The columns must be ordered such that the first `nClassVariables` columns contain the class variables and the next `nContinuousVariables` columns contain the continuous variables.

`nClassVariables` – is number of class variables. The number of continuous variables is assumed to be the number of columns in `x-nClassVariables`.

RegressorsForGLM

```
public RegressorsForGLM(double[][] x, int[] classColumns)
```

Description

Constructor with an explicit set of class column indicies.

Parameters

`x` – is an `nObservations` by `nClassVariables+nContinuousVariables` array containing the data. The columns containing the class variables are specified by `classColumns`.

`classColumns` – is an array containing the columns indices, in `x`, of the class variables.

Methods

getDummyMethod

```
public int getDummyMethod()
```

Description

Returns the dummy method.

Returns

One of `ALL` (the default), `LEAVE_OUT_LAST` or `SUM_TO_ZERO`.

getEffects

```
public int[][] getEffects()
```

Description

Returns the effects.

Returns

a jagged array containing the effects. The number of rows in the matrix is the number of effects. For each row, the values are the 0-based column numbers of `x`.

getEffectsColumns

```
public int[][] getEffectsColumns()
```

Description

Returns a mapping of effects to regressor columns.

Returns

A jagged `int` array. The number of rows is equal to the number of effects. Each row contains the column numbers of the regressor matrix into which the corresponding effect is mapped.

getNumberOfMissingRows

```
public int getNumberOfMissingRows()
```

Description

Returns the number of rows in the regressors matrix containing `NaN` (not a number). A row of the regressors matrix contains `NaN` for a regressor when any of the variables involved in generation of the regressor equals `NaN` or if a value of one of the classification variables in the model is not given by effects.

Returns

The number of rows in the data matrix having missing data.

getNumberOfRegressors

```
public int getNumberOfRegressors()
```

Description

Returns the number regressors.

Returns

The number of regressors. This is the number of columns in the regressor matrix.

getRegressors

```
public double[][] getRegressors()
```

Description

Returns the regressor array.

Returns

An array of size number of observations by number of regresssors.

setDummyMethod

```
public void setDummyMethod(int dummyMethod)
```

Description

Sets the dummy method.

Parameter

`dummyMethod` – must be one of `ALL` (the default), `LEAVE_OUT_LAST` or `SUM_TO_ZERO`.

setEffects

```
public void setEffects(int[][] effects)
```

Description

Set the effects. This overrides any previously set model order.

Parameter

`effects` – is a jagged array. The number of rows in the matrix is the number of effects. For each row, the values are the 0-based column numbers of `x`.

setModelOrder

```
public void setModelOrder(int modelOrder)
```

Description

Sets the order of the model. Model order can be specified as 1 or 2. Use `setEffects` to specify more complicated models. This overrides previously set effects.

Parameter

`modelOrder` – is one or two. The default effects are equivalent to model equal to one.

Example 1

In the following example, there are two classification variables, A and B, with two and three values, respectively. Regressors for a one-way model (the default model order) are generated using the the default dummy method. The five regressors generated are A_1, A_2, B_1, B_2 and B_3.

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

public class RegressorsForGLMEx1 {

    public static void main(String args[]) {
        double x[][] = {
            {10.0, 5.0},
            {20.0, 15.0},
            {20.0, 10.0},
            {10.0, 10.0},
            {10.0, 15.0},
            {20.0, 5.0}
        };

        RegressorsForGLM r = new RegressorsForGLM(x, 2);
        double regressors[][] = r.getRegressors();
        int n = r.getNumberOfRegressors();
        System.out.println("Number of regressors = " + n);
        new PrintMatrix("Regressors").print(regressors);
    }
}
```

Output

```
Number of regressors = 5
    Regressors
   0  1  2  3  4
0  1  0  1  0  0
1  0  1  0  0  1
2  0  1  0  1  0
```

```
3  1  0  0  1  0
4  1  0  0  0  1
5  0  1  1  0  0
```

Example 2

In this example, a two-way analysis of covariance model containing all the interaction terms is fit. First, `RegressorsForGLM` is used to produce a matrix of regressors, `regressors`, from the data `x`. Then, `regressors` is used as the input matrix into `Regression` to produce the final fit. The mapping from effects to columns in the matrix `regressors` is given by the jagged array returned by `getEffectsColumns`. Each row in this matrix gives the column number for the corresponding effect.

The regressors, generated using the dummy method `LEAVE_OUT_LAST`, are the model whose mean function is

$$\mu + \alpha_i + \beta_j + \gamma_{ij} + \delta x_{ij} + \zeta_i x_{ij} + \eta x_{ij} + \theta_i x_{ij} \quad i = 1,2;\ j = 1,2,3$$

where $\alpha_2 = \beta_3 = \gamma_{21} + \gamma_{22} + \gamma 23 + \zeta_2 + \eta_3 + \theta_{21} + \theta_{22} + \theta_{23} = 0$.

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

public class RegressorsForGLMEx2 {

    static public void main(String arg[]) {
        double x[][] = {
            {1.0, 1.0, 1.11},
            {1.0, 1.0, 2.22},
            {1.0, 1.0, 3.33},
            {1.0, 2.0, 1.11},
            {1.0, 2.0, 2.22},
            {1.0, 2.0, 3.33},
            {1.0, 3.0, 1.11},
            {1.0, 3.0, 2.22},
            {1.0, 3.0, 3.33},
            {2.0, 1.0, 1.11},
            {2.0, 1.0, 2.22},
            {2.0, 1.0, 3.33},
            {2.0, 2.0, 1.11},
            {2.0, 2.0, 2.22},
            {2.0, 2.0, 3.33},
            {2.0, 3.0, 1.11},
            {2.0, 3.0, 2.22},
            {2.0, 3.0, 3.33}
        };
        double y[] = {
            1.0, 2.0, 2.0, 4.0, 4.0, 6.0,
            3.0, 3.5, 4.0, 4.5, 5.0, 5.5,
            2.0, 3.0, 4.0, 5.0, 6.0, 7.0
        };
        int effects[][] = {
            {0},
            {1},
```

```
    {0, 1},
    {2},
    {0, 2},
    {1, 2},
    {0, 1, 2}
};

int nClassVariables = 2;

RegressorsForGLM r = new RegressorsForGLM(x, nClassVariables);
r.setDummyMethod(RegressorsForGLM.LEAVE_OUT_LAST);
r.setEffects(effects);

System.out.println("Mapping of Effects to Regressor Columns");
int col[][] = r.getEffectsColumns();
for (int iEffect = 0; iEffect < effects.length; iEffect++) {
    System.out.print("Effect {");
    for (int j = 0; j < effects[iEffect].length; j++) {
        if (j > 0) {
            System.out.print(", ");
        }
        System.out.print(effects[iEffect][j]);
    }
    System.out.print("} is in regressor column(s) {");
    for (int j = 0; j < col[iEffect].length; j++) {
        if (j > 0) {
            System.out.print(", ");
        }
        System.out.print(col[iEffect][j]);
    }
    System.out.println("}");
}
System.out.println();

double regressors[][] = r.getRegressors();

PrintMatrixFormat pmf = new PrintMatrixFormat();
pmf.setColumnLabels(new String[]{"Alpha1", "Beta1", "Beta2",
    "Gamma11", "Gamma12", "Delta", "Zeta1", "Eta1", "Eta2",
    "Theta11", "Theta12"});
new PrintMatrix("Regressors").print(pmf, regressors);

int nRegressors = r.getNumberOfRegressors();
LinearRegression regression = new LinearRegression(nRegressors, true);
regression.update(regressors, y);

System.out.println("        * * * Analysis of Variance * * *");
ANOVA anova = regression.getANOVA();

Object table[][] = new Object[15][2];
table[0][0] = "Degrees of freedom for the model       ";
table[0][1] = anova.getDegreesOfFreedomForModel();
table[1][0] = "Degrees of freedom for the error       ";
table[1][1] = anova.getDegreesOfFreedomForError();
table[2][0] = "Total degrees of freedom               ";
table[2][1] = anova.getTotalDegreesOfFreedom();
```

```
        table[3][0] = "Sum of squares for the model          ";
        table[3][1] = anova.getSumOfSquaresForModel();
        table[4][0] = "Sum of squares for error              ";
        table[4][1] = anova.getSumOfSquaresForError();
        table[5][0] = "Total sum of squares                  ";
        table[5][1] = anova.getTotalSumOfSquares();
        table[6][0] = "Model mean square                     ";
        table[6][1] = anova.getModelMeanSquare();
        table[7][0] = "Error mean square                     ";
        table[7][1] = anova.getErrorMeanSquare();
        table[8][0] = "F-statistic                           ";
        table[8][1] = anova.getF();
        table[9][0] = "p-value                               ";
        table[9][1] = anova.getP();
        table[10][0] = "R-squared                            ";
        table[10][1] = anova.getRSquared();
        table[11][0] = "Adjusted R-squared                   ";
        table[11][1] = anova.getAdjustedRSquared();
        table[12][0] = "Standard deviation for the model error";
        table[12][1] = anova.getModelErrorStdev();
        table[13][0] = "Overall mean of y                    ";
        table[13][1] = anova.getMeanOfY();
        table[14][0] = "Coefficient of variation             ";
        table[14][1] = anova.getCoefficientOfVariation();
        pmf = new PrintMatrixFormat();
        pmf.setNoColumnLabels();
        pmf.setNoRowLabels();
        pmf.setNumberFormat(new java.text.DecimalFormat("0.0000"));
        new PrintMatrix().print(pmf, table);
    }
}
```

Output

```
Mapping of Effects to Regressor Columns
Effect {0} is in regressor column(s) {0}
Effect {1} is in regressor column(s) {1, 2}
Effect {0, 1} is in regressor column(s) {3, 4}
Effect {2} is in regressor column(s) {5}
Effect {0, 2} is in regressor column(s) {6}
Effect {1, 2} is in regressor column(s) {7, 8}
Effect {0, 1, 2} is in regressor column(s) {9, 10}
```

Regressors

	Alpha1	Beta1	Beta2	Gamma11	Gamma12	Delta	Zeta1	Eta1	Eta2	Theta11	Theta12
0	1	1	0	1	0	1.11	1.11	1.11	0	1.11	0
1	1	1	0	1	0	2.22	2.22	2.22	0	2.22	0
2	1	1	0	1	0	3.33	3.33	3.33	0	3.33	0
3	1	0	1	0	1	1.11	1.11	0	1.11	0	1.11
4	1	0	1	0	1	2.22	2.22	0	2.22	0	2.22
5	1	0	1	0	1	3.33	3.33	0	3.33	0	3.33
6	1	0	0	0	0	1.11	1.11	0	0	0	0
7	1	0	0	0	0	2.22	2.22	0	0	0	0
8	1	0	0	0	0	3.33	3.33	0	0	0	0
9	0	1	0	0	0	1.11	0	1.11	0	0	0

```
10    0       1      0       0        0     2.22  0      2.22  0      0        0
11    0       1      0       0        0     3.33  0      3.33  0      0        0
12    0       0      1       0        0     1.11  0      0     1.11   0        0
13    0       0      1       0        0     2.22  0      0     2.22   0        0
14    0       0      1       0        0     3.33  0      0     3.33   0        0
15    0       0      0       0        0     1.11  0      0     0      0        0
16    0       0      0       0        0     2.22  0      0     0      0        0
17    0       0      0       0        0     3.33  0      0     0      0        0

        * * * Analysis of Variance * * *

Degrees of freedom for the model        11.0000
Degrees of freedom for the error         6.0000
Total degrees of freedom                17.0000
Sum of squares for the model            43.9028
Sum of squares for error                 0.8333
Total sum of squares                    44.7361
Model mean square                        3.9912
Error mean square                        0.1389
F-statistic                             28.7364
p-value                                  0.0003
R-squared                               98.1372
Adjusted R-squared                      94.7221
Standard deviation for the model error   0.3727
Overall mean of y                        3.9722
Coefficient of variation                 9.3821
```

LinearRegression class

`public class com.imsl.stat.LinearRegression implements Serializable, Cloneable`

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 σ^2 / w_i. If `hasIntercept` is `false`, β_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

$$\mathrm{SSE} = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i)^2$$

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

$$\mathrm{SST} = \sum_{i=1}^{n} w_i (y_i - \bar{y})^2$$

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

$$\mathrm{SST} = \sum_{i=1}^{n} y_i^2$$

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.

From a general linear model fitted using the w_i's as the weights, inner class `com.imsl.stat.LinearRegression.CaseStatistics` (p. 717) can also compute predicted values, confidence intervals, and diagnostics for detecting outliers and cases that greatly influence the fitted regression. Let x_i be a column vector containing elements of the i-th row of X. Let $W = diag(w_1, w_2, ..., w_n)$. The leverage is defined as

$$h_i = [x_i^T (X^T W X)^{-} x_i] w_i$$

(In the case of linear equality restrictions on β, the leverage is defined in terms of the reduced model.) Put $D = diag(d_1, d_2, ..., d_k)$ with $d_j = 1$ if the j-th diagonal element of R is positive and 0 otherwise. The leverage is computed as $h_i = (a^T D a) w_i$ where a is a solution to $R^T a = x_i$. The estimated variance of

$$\hat{y}_i = x_i^T \hat{\beta}$$

is given by $h_i s^2 / w_i$, where $s^2 = SSE/DFE$. The computation of the remainder of the case statistics follows easily from their definitions.

Let e_i denote the residual

$$y_i - \hat{y}_i$$

for the ith case. The estimated variance of e_i is $(1-h_i)s^2/w_i$ where s^2 is the residual mean square from the fitted regression. The ith standardized residual (also called the internally studentized residual) is by definition

$$r_i = e_i\sqrt{\frac{w_i}{s^2(1-h_i)}}$$

and r_i follows an approximate standard normal distribution in large samples.

The ith jackknife residual or deleted residual involves the difference between y_i and its predicted value based on the data set in which the ith case is deleted. This difference equals $e_i/(1-h_i)$. The jackknife residual is obtained by standardizing this difference. The residual mean square for the regression in which the ith case is deleted is

$$s_i^2 = \frac{(n-r)s^2 - w_i e_i^2/(1-h_i)}{n-r-1}$$

The jackknife residual is defined to be

$$t_i = e_i\sqrt{\frac{w_i}{s_i^2(1-h_i)}}$$

and t_i follows a t distribution with $n-r-1$ degrees of freedom.

Cook's distance for the ith case is a measure of how much an individual case affects the estimated regression coefficients. It is given by

$$D_i = \frac{w_i h_i e_i^2}{rs^2(1-h_i)^2}$$

Weisberg (1985) states that if D_i exceeds the 50-th percentile of the $F(r, n-r)$ distribution, it should be considered large. (This value is about 1. This statistic does not have an F distribution.)

DFFITS, like Cook's distance, is also a measure of influence. For the ith case, DFFITS is computed by the formula

$$DFFITS_i = e_i\sqrt{\frac{w_i h_i}{s_i^2(1-h_i)^2}}$$

Hoaglin and Welsch (1978) suggest that $DFFITS_i$ greater than

$$2\sqrt{r/n}$$

is large.

Often predicted values and confidence intervals are desired for combinations of settings of the effect variables not used in computing the regression fit. This can be accomplished using a single data matrix by including these settings of the variables as part of the data matrix and by setting the response equal to `Double.NaN`. `LinearRegression` will omit the case when performing the fit and a predicted value and confidence interval for the missing response will be computed from the given settings of the effect variables.

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` – `boolean` which indicates whether or not an intercept is in this regression model

Methods

getANOVA

`public ANOVA getANOVA()`

Description

Get an analysis of variance table and related statistics.

Returns

an `ANOVA` table and related statistics

getCaseStatistics

`public LinearRegression.CaseStatistics getCaseStatistics(double[] x, double y)`

Description

Returns the case statistics for an observation.

Parameters

`x` – a `double` array containing the independent (explanatory) variables. Its length must be equal to the number of variables set in the LinearRegression constructor.

`y` – a `double` representing the dependent (response) variable

Returns

the CaseStatistics for the observation.

getCaseStatistics

`public LinearRegression.CaseStatistics getCaseStatistics(double[] x, double y, double w)`

Description

Returns the case statistics for an observation and a 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

Returns

the CaseStatistics for the observation.

getCaseStatistics

`public LinearRegression.CaseStatistics getCaseStatistics(double[] x, double y, int pred)`

Description

Returns the case statistics for an observation and future response count for the desired prediction interval.

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

`pred` – an `int` representing the number of future responses for which the prediction interval is desired on the average of the future responses.

Returns

the CaseStatistics for the observation.

getCaseStatistics

`public LinearRegression.CaseStatistics getCaseStatistics(double[] x, double y, double w, int pred)`

Description

Returns the case statistics for an observation, weight, and future response count for the desired prediction interval.

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

`pred` – an `int` representing the number of future responses for which the prediction interval is desired on the average of the future responses

Returns

the CaseStatistics for the observation.

getCoefficientTTests

`public LinearRegression.CoefficientTTests getCoefficientTTests()`

Description

Returns statistics relating to the regression coefficients.

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. If the model is not full rank, the regression coefficients are not uniquely determined. In this case, a warning is issued and a solution with all linearly dependent regressors set to zero is returned.

---

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 `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 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 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 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

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

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}
```

Example2: Linear Regression

Selected case statistics of a simple linear regression model, with an intercept, are computed.

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

public class LinearRegressionEx2 {

    public static void main(String args[]) {
        LinearRegression r = new LinearRegression(2, true);
        double y[] = {3, 4, 5, 7, 7, 8, 9};
        double x[][] = {
            {1, 1}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {0, 6}, {1, 7}
        };
        double[][] results = new double[7][5];
        double[] confint = new double[2];
        r.update(x, y);
        for (int k = 0; k < 7; k++) {
            LinearRegression.CaseStatistics cs
                    = r.getCaseStatistics(x[k], y[k]);
            results[k][0] = cs.getJackknifeResidual();
            results[k][1] = cs.getCooksDistance();
            results[k][2] = cs.getDFFITS();
            confint = cs.getConfidenceInterval();
            results[k][3] = confint[0];
            results[k][4] = confint[1];
        }
        PrintMatrix p = new PrintMatrix("Selected Case Statistics");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        String labels[] = {
            "Jackknife Residual.", "Cook's D", "DFFITS",
            "[Conf. Interval", "on the Mean]"
        };
        mf.setColumnLabels(labels);
        p.print(mf, results);
    }
}
```

Output

```
                        Selected Case Statistics
```

```
   Jackknife Residual.  Cook's D  DFFITS  [Conf. Interval  on the Mean]
0        -0.343          0.045    -0.324       2.261          3.996
1        -0.327          0.018    -0.207       3.467          4.818
2        -0.338          0.011    -0.161       4.613          5.702
3         ?              0.276     ?           5.648          6.695
4        -0.418          0.024    -0.237       6.563          7.808
5         ?              ?         ?           6.736          9.264
6        -0.742          0.372    -0.996       8.201         10.227
```

LinearRegression.CoefficientTTests class

`public class com.imsl.stat.LinearRegression.CoefficientTTests implements Serializable`

Contains statistics related to the regression coefficients.

Methods

getCoefficient

`public double getCoefficient(int i)`

Description

Returns the estimate for a coefficient.

Parameter

`i` – an `int` which specifies the index of the coefficient whose estimate is to be returned.

Returns

a `double` which contains the estimate for the *i*-th coefficient.

getPValue

`public double getPValue(int i)`

Description

Returns the *p*-value for the two-sided test.

Parameter

`i` – an `int` which specifies the index of the coefficient whose *p*-value is to be returned.

Returns

a `double` which contains 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.

Parameter

`i` – an `int` which specifies the index of the coefficient whose stardard error estimate is to be returned.

Returns

a `double` which contains 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.

Parameter

`i` – an `int` specifying the index of the coefficient whose stardard error estimate is to be returned.

Returns

a `double` which contains the estimated standard error for the *i*-th coefficient estimate.

LinearRegression.CaseStatistics class

```
public class com.imsl.stat.LinearRegression.CaseStatistics
```

Inner Class `CaseStatistics` allows for the computation of predicted values, confidence intervals, and diagnostics for detecting outliers and cases that greatly influence the fitted regression.

Methods

getConfidenceInterval

```
public double[] getConfidenceInterval()
```

Description

Returns the Confidence Interval of the population mean for an observation.

The width of the confidence interval depends on the confidence level set by `com.imsl.stat.LinearRegression.CaseStatistics.setConLevelMean` (p. 719) .

Returns

a `double`[2] array containing the Confidence Interval of the population mean at the given observation.

---

getCooksDistance

`public double getCooksDistance()`

Description

Returns Cook's Distance for an observation.

Returns

a `double` containing Cook's Distance for an observation

---

getDFFITS

`public double getDFFITS()`

Description

Returns DFFITS for an observation.

Returns

a `double` containing the DFFITS value for an observation

---

getJackknifeResidual

`public double getJackknifeResidual()`

Description

Returns the Jackknife Residual for an observation.

Returns

a `double` containing the Jackknife Residual for an observation

---

getLeverage

`public double getLeverage()`

Description

Returns the Leverage for an observation.

Returns

a `double` containing the Leverage for an observation

---

getObservedResponse

`public double getObservedResponse()`

Description

Returns the observed response for an observation.

Returns

a `double` containing the observed response for an observation

getPredictedResponse

```
public double getPredictedResponse()
```

Description

Returns the predicted response for an observation.

Returns

a `double` containing the predicted response for an observation

getPredictionInterval

```
public double[] getPredictionInterval()
```

Description

Returns the Prediction Interval of the predicted response for an observation.

The width of the prediction interval depends on the confidence level set by `com.imsl.stat.LinearRegression.CaseStatistics.setConLevelPred` (p. 720) .

Returns

a `double`[2] array containing the Prediction Interval of the predicted response at the given observation

getResidual

```
public double getResidual()
```

Description

Returns the Residual for an observation.

Returns

a `double` containing the residual for an observation

getStandardizedResidual

```
public double getStandardizedResidual()
```

Description

Returns the Standardized Residual for an observation.

Returns

a `double` containing the Standardized Residual for an observation

setConLevelMean

```
public void setConLevelMean(double conpcm)
```

Description

Sets the confidence level for two-sided confidence intervals of the population mean.

Parameter

`conpcm` – a `double` used to set the confidence level for two-sided confidence intervals of the population mean. Note that `conpcm` must be in the interval `0` to `1` inclusive. That is, it must be expressed as a probability.

Default: `conpcm`=.95.

setConLevelPred

```
public void setConLevelPred(double conpcp)
```

Description

Sets the confidence level for two-sided prediction intervals.

Parameter

`conpcp` – a `double` used to set the confidence level for two-sided prediction intervals of a predicted response. Note that `conpcp` must be in the interval `0` to `1` inclusive. That is, it must be expressed as a probability.

Default: `conpcp`=.95.

NonlinearRegression class

```
public class com.imsl.stat.NonlinearRegression
```

Fits a multivariate nonlinear regression model using least squares. The nonlinear regression model is

$$y_i = f(x_i;\theta) + \varepsilon_i \quad i = 1, 2, \ldots, n$$

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, θ is the vector of p regression parameters, and the ε_i are independently distributed normal errors each with mean zero and variance σ^2. For this model, a least squares estimate of θ is also a maximum likelihood estimate of θ.

The residuals for the model are

$$e_i(\theta) = y_i - f(x_i;\theta) \quad i = 1, 2, \ldots, n$$

A value of θ that minimizes

$$\sum_{i=1}^{n} [e_i(\theta)]^2$$

is the least-squares estimate of θ 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 θ. 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, μ_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 ε_i in the nonlinear regression model is not constant but instead is σ^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 θ 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 (`com.imsl.stat.NonlinearRegression.setMaxStepsize` (p. 726)) 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 `com.imsl.stat.NonlinearRegression.setDigits` (p. 725) to indicate the actual number of good digits in the residuals.
3. The model is discontinuous as a function of θ. 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` (p. 735) . If the derivative is supplied try providing only `com.imsl.stat.NonlinearRegression.Function` (p. 734) .
5. Overflow occurs during the computations. Make sure the user-supplied functions do not overflow at some value of θ.
6. The estimate of θ is going to infinity. A parameterization of the problem in terms of reciprocals may help.
7. Some components of θ 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.

Constructor

NonlinearRegression

`public NonlinearRegression(int nparm)`

Description

Constructs a new nonlinear regression object.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`IllegalArgumentException` is thrown if `ngood` is less than or equal to 0

setFalseConvergenceTolerance

```
public void setFalseConvergenceTolerance(double falseConvergenceTolerance)
```

Description

Sets the false convergence tolerance.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`IllegalArgumentException` is thrown if `gradientTolerance` is less than 0

---

setGuess

```
public void setGuess(double[] thetaGuess)
```

Description

Sets the initial guess of the parameter values

Parameter

`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.

Parameter

`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.

Exception

`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

Parameter

`maxIterations` – An `int` specifying the maximum number of iterations allowed during optimization. The value must be greater than 0. The default value is 100.

Exception

`IllegalArgumentException` is thrown if `maxIterations` is less than or equal to 0

---

setMaxStepsize

```
public void setMaxStepsize(double maxStepsize)
```

Description

Sets the maximum allowable stepsize.

Parameter

`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`.

Exception

`IllegalArgumentException` is thrown if `maxStepsize` is less than or equal to 0

setRelativeTolerance

`public void setRelativeTolerance(double relativeTolerance)`

Description

Sets the relative function tolerance

Parameter

`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.

Exception

`IllegalArgumentException` is thrown if `relativeTolerance` is less than 0

setScale

`public void setScale(double[] scale)`

Description

Sets the scaling array for `theta`.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`IllegalArgumentException` is thrown if `stepTolerance` is less than 0

solve

`public double[] solve(NonlinearRegression.Function F) throws NonlinearRegression.TooManyIterationsException, NonlinearRegression.NegativeFreqException, NonlinearRegression.NegativeWeightException`

Description

Solves the least squares problem and returns the regression coefficients.

Parameter

`F` – A `NonlinearRegression.Function` whose coefficients are to be computed.

Returns

A `double` array containing the regression coefficients.

Exceptions

`TooManyIterationsException` is thrown when the number of allowed iterations is exceeded

`NegativeFreqException` is thrown when the specified frequency is negative

`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 Exception {
        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 {58.60656294699903, -0.039586447308674395}
The computed rank is 2
The degrees of freedom for error are 13.0
The sums of squares for error is 49.459299862471724
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 Exception {
        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 49.459299862472186
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 Exception {
        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];
    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);
    double 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
```

NonlinearRegression.NegativeFreqException class

```
static public class com.imsl.stat.NonlinearRegression.NegativeFreqException
extends com.imsl.IMSLException
```

A negative frequency was encountered.

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.

NonlinearRegression.NegativeWeightException class

```
static public class com.imsl.stat.NonlinearRegression.NegativeWeightException
extends com.imsl.IMSLException
```

A negative weight was encountered.

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.

NonlinearRegression.TooManyIterationsException class

```
static public class
com.imsl.stat.NonlinearRegression.TooManyIterationsException extends
com.imsl.IMSLException
```

The number of iterations has exceeded the maximum allowed.

Constructor

NonlinearRegression.TooManyIterationsException

```
public NonlinearRegression.TooManyIterationsException()
```

Description

Constructs a `TooManyIterationsException`.

NonlinearRegression.Function interface

```
public interface com.imsl.stat.NonlinearRegression.Function
```

Public interface for the user supplied function for `NonlinearRegression`.

Method

f

```
public 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.

NonlinearRegression.Derivative interface

```
public interface com.imsl.stat.NonlinearRegression.Derivative implements
com.imsl.stat.NonlinearRegression.Function
```

Public interface for the user supplied function to compute the derivative for `NonlinearRegression`.

Method

derivative

```
public 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.

UserBasisRegression class

```
public class com.imsl.stat.UserBasisRegression
```

Fits a linear function of the form $y = c_0 + c_1 f_1(x) + c_2 f_2(x) + \cdots + c_k f_k(x) + \varepsilon$, where $f_1(x), f_2(x), \cdots, f_k(x)$ are the user basis functions $f_i(x)$ evaluated at index values $i = 1, 2, \ldots, k, c_0$ is the intercept, $c_1, c_2, \cdots, c_k$ are the coefficients associated with the basis functions, and is the random error associated with y. The coefficients $c_0, c_1, \cdots, c_k$ are determined by least squares.

Description

`UserBasisRegression` generalizes the concept of linear regression to user defined basis functions. The linear regression model is $y = c_0 + c_1 x_1 + \cdots + c_k x_k + \varepsilon$, where are the k independent variables. `UserBasisRegression` generalizes this concept by setting $x_i = f_i(x)$, where $f_i(x)$ is any user defined function of x.

This makes it easier for users to fit complex univariate models. For example, the `LinearRegression` class can be used to fit polynomials such as $y = c_0 + c_1 x + c_2 x^2 \cdots + c_k x^k + \varepsilon$, but this requires an input matrix where the ith column of that array contains the values of x^i.

With `UserBasisRegression`, these columns can be automatically generated. For this polynomial model, the user would define a user basis function $f_i(x) = x^{i+1}$. The `UserBasisRegression` class

automatically inserts the necessary values into the regression equation and then calculates the coefficients and analysis of variance statistics.

Since the user provides a method for calculating the basis function, other more complex user basis functions are possible such as $y = c_1 Sin(x) + c_2 Cos(x) + \varepsilon$. In this example, `nBasis`=2, $f_0(x) = Sin(x)$, and $f_1(x) = Cos(x)$.

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.

Exception

`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);
            }
        }

        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);
        }
        double[] 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
```

Example: Regression with User-supplied Basis Functions

In this example, we fit the polynomial $y = c_0 + c_1 x + c_2 x^2 \cdots + c_4 x^4 + \varepsilon$. For this model, the user basis function is $f_i(x) = x^{i+1}$ with $i = 0, 1, \cdots, nBasis = 4$ and `hasIntercept=true`.

Data are generated using the model $y = 10 + 2x + 5x^3$ with $x = 0, 1, \cdots 9$.

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

public class UserBasisRegressionEx2 {

    public static void main(String args[]) {
        class Basis implements RegressionBasis {

            public double basis(int index, double x) {
                // Notice zero-based indexing requires index be incremented
                return Math.pow(x, index + 1);
            }
        }

        UserBasisRegression ubr
                = new UserBasisRegression(new Basis(), 4, true);

        for (double x = 0; x < 10; x++) {
            double y = 10.0 + x + 3 * Math.pow(x, 3);
            ubr.update(x, y, 1.0);
        }

        // Print the regression coefficients
        double coef[] = ubr.getCoefficients();
        new PrintMatrix("The regression coefficients are:").print(coef);

        // Print the Anova Table
        ANOVA anovaTable = ubr.getANOVA();

        System.out.println("Degrees Of Freedom For Model:          "
                + anovaTable.getDegreesOfFreedomForModel());
        System.out.println("Degrees Of Freedom For Error:          "
                + anovaTable.getDegreesOfFreedomForError());
        System.out.println("Total (Corrected) Degrees Of Freedom:  "
                + anovaTable.getTotalDegreesOfFreedom());
        System.out.println("Sum Of Squares For Model:              "
                + anovaTable.getSumOfSquaresForModel());
        System.out.println("Sum Of Squares For Error:              "
                + anovaTable.getSumOfSquaresForError());
        System.out.println("Total (Corrected) Sum Of Squares:      "
                + anovaTable.getTotalSumOfSquares());
```

```
        System.out.println("Model Mean Square:                      "
                + anovaTable.getModelMeanSquare());
        System.out.println("Error Mean Square:                      "
                + anovaTable.getErrorMeanSquare());
        System.out.println("F statistic:                            "
                + anovaTable.getF());
        System.out.println("P value:                                "
                + anovaTable.getP());
        System.out.println("R Squared (in percent):                 "
                + anovaTable.getRSquared());
        System.out.println("Adjusted R Squared (in percent):        "
                + anovaTable.getAdjustedRSquared());
        System.out.println("Model Error Standard deviation:         "
                + anovaTable.getModelErrorStdev());
        System.out.println("Mean Of Y:                              "
                + anovaTable.getMeanOfY());
        System.out.println("Coefficient Of Variation (in percent): "
                + anovaTable.getCoefficientOfVariation());
    }
}
```

Output

```
The regression coefficients are:
   0
0  10
1   1
2  -0
3   3
4  -0

Degrees Of Freedom For Model:          4.0
Degrees Of Freedom For Error:          5.0
Total (Corrected) Degrees Of Freedom:  9.0
Sum Of Squares For Model:              5152487.999999998
Sum Of Squares For Error:              1.862645149230957E-9
Total (Corrected) Sum Of Squares:      5152488.0
Model Mean Square:                     1288121.9999999995
Error Mean Square:                     3.7252902984619143E-10
F statistic:                           3.4577761645363185E15
P value:                               8.696627855641336E-39
R Squared (in percent):                99.99999999999997
Adjusted R Squared (in percent):       99.99999999999993
Model Error Standard deviation:        1.9301011109426144E-5
Mean Of Y:                             622.0
Coefficient Of Variation (in percent): 3.103056448460795E-6
```

RegressionBasis interface

```
public interface com.imsl.stat.RegressionBasis
```

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

Method

basis

```
public 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

SelectionRegression class

```
public class com.imsl.stat.SelectionRegression implements Serializable,
Cloneable
```

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)

$$R^2 = 100(1 - \frac{\mathrm{SSE}_p}{\mathrm{SST}})$$

- 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

$$C_p = \frac{\text{SSE}_p}{s_{\text{k}}^2} + 2p - n$$

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. SSE_p is the error sum of squares in a model containing p regression parameters including β_0 (or p - 1 of the k candidate variables). Variable

$$S_{\text{k}}^2$$

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 `com.imsl.stat.SelectionRegression.setMaximumGoodSaved` (p. 747) . 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 $A^T A$, 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 $k+1 > -\log_2(\varepsilon)$, where ε 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 $k > 49$) 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 "VariablesDeleted" warning 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.

Fields

ADJUSTED_R_SQUARED_CRITERION

```
static final public int ADJUSTED_R_SQUARED_CRITERION
```

Indicates R_a^2 (adjusted R^2) criterion regression.

MALLOWS_CP_CRITERION

```
static final public int MALLOWS_CP_CRITERION
```

Indicates Mallow's C_p criterion regression.

R_SQUARED_CRITERION

```
static final public int R_SQUARED_CRITERION
```

Indicates R^2 criterion regression.

Constructor

SelectionRegression

`public SelectionRegression(int nCandidate)`

Description

Constructs a new `SelectionRegression` object.

Parameter

`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
SelectionRegression.NoVariablesException,
Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, 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.

Exceptions

`NoVariablesException` if no variables can enter any model

`Covariances.TooManyObsDeletedException` more observations have been deleted than were originally entered

`Covariances.MoreObsDelThanEnteredException` more observations are being deleted from the output covariance matrix than were originally entered

`Covariances.DiffObsDeletedException` different observations are being deleted from return matrix than were originally entered

compute

```
public void compute(double[][] cov, int nObservations) throws
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`.

Exception

`NoVariablesException` if no variables can enter any model

compute

```
public void compute(double[][] x, double[] y, double[] weights) throws
SelectionRegression.NoVariablesException,
Covariances.NonnegativeWeightException, Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, 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.

Exceptions

`NoVariablesException` if no variables can enter any model

`Covariances.NonnegativeWeightException` weights must be nonnegative

`Covariances.TooManyObsDeletedException` more observations have been deleted than were originally entered

`Covariances.MoreObsDelThanEnteredException` more observations are being deleted from the output covariance matrix than were originally entered

`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 SelectionRegression.NoVariablesException,
Covariances.NonnegativeFreqException, Covariances.NonnegativeWeightException,
Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, 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`.

Exceptions

`NoVariablesException` if no variables can enter any model

`Covariances.NonnegativeFreqException` frequencies must be nonnegative

`Covariances.NonnegativeWeightException` weights must be nonnegative

`Covariances.TooManyObsDeletedException` more observations have been deleted than were originally entered

`Covariances.MoreObsDelThanEnteredException` more observations are being deleted from the output covariance matrix than were originally entered

`Covariances.DiffObsDeletedException` different observations are being deleted from return matrix than were originally entered

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 `com.imsl.stat.SelectionRegression.setMaximumSubsetSize` (p. 747) 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.

Parameter

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.

Parameter

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.

Parameter

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.

Parameter

`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). The method `compute` 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.*;

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");

        SelectionRegression sr = new SelectionRegression(4);
        sr.compute(x, y);
        SelectionRegression.Statistics stats
                = sr.getStatistics();
```

```
        criterionOption = "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
     67.45419641316094        4
     66.62682576332938        2
     53.39480238350332        1
     28.58727312298116        3

Regressions with 2 variable(s) (R-squared)
   Criterion               Variables
     97.86783745356314        1   2
     97.24710477169312        1   4
     93.52896406158074        3   4
```

```
     68.00604079500502        2   4
     54.81667488448575        1   3

Regressions with 3 variable(s) (R-squared)
   Criterion               Variables
     98.23354512004263        1   2   4
     98.22846792190859        1   2   3
     98.12810925873434        1   3   4
     97.28199593862728        2   3   4

Regressions with 4 variable(s) (R-squared)
   Criterion               Variables
     98.23756204076797        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.*;

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");

        SelectionRegression sr = new SelectionRegression(4);
        sr.setCriterionOption(SelectionRegression.MALLOWS_CP_CRITERION);
        sr.setMaximumBestFound(3);
        sr.compute(x, y);
        SelectionRegression.Statistics stats = sr.getStatistics();

        criterionOption = "Mallows Cp";
```

```
        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 < 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) (Mallows Cp)
   Criterion               Variables
     138.73083349167362        4
     142.4864069369577        2
     202.54876912344534        1
     315.154284140073        3

Regressions with 2 variable(s) (Mallows Cp)
```

```
   Criterion               Variables
     2.6782415983184045       1   2
     5.495850824758396       1   4
     22.37311196469674       3   4
     138.22591975463834       2   4
     198.09465256956904       1   3

Regressions with 3 variable(s) (Mallows Cp)
   Criterion               Variables
     3.0182334734873084       1   2   4
     3.04127972306423       1   2   3
     3.4968244423483164       1   3   4
     7.337473995655756       2   3   4

Regressions with 4 variable(s) (Mallows Cp)
   Criterion               Variables
     5.0       1   2   3   4


Best Regressions with 2 variable(s) (Mallows Cp)
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) (Mallows Cp)
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) (Mallows Cp)
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
```

SelectionRegression.NoVariablesException class

```
static public class com.imsl.stat.SelectionRegression.NoVariablesException
extends com.imsl.IMSLException
```

No Variables can enter the model.

Constructor

SelectionRegression.NoVariablesException

```
public SelectionRegression.NoVariablesException()
```

Description

Constructs a `NoVariablesException`.

SelectionRegression.Statistics class

```
public class com.imsl.stat.SelectionRegression.Statistics implements
Serializable
```

`Statistics` contains statistics related to the regression coefficients.

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 `com.imsl.stat.SelectionRegression.setMaximumBestFound` (p. 747) 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

Parameter

`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.

Parameter

`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
`com.imsl.stat.SelectionRegression.Statistics.getCriterionValues` (p. 755) .

Parameter

`numVariables` – An `int` which specifies the number of variables considered.

Returns

An `int` matrix containing the identification numbers for the independent variables considered.

StepwiseRegression class

`public class com.imsl.stat.StepwiseRegression implements Serializable, Cloneable`

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 `com.imsl.stat.StepwiseRegression.setLevels` (p. 762) 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 (`com.imsl.stat.StepwiseRegression.setForce` (p. 762)). 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` (p. 741)) 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.

Fields

BACKWARD_REGRESSION

`static final public 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 is greater than `pValueOut`. During initialization, all candidate independent variables enter the model.

FORWARD_REGRESSION

`static final public 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.

STEPWISE_REGRESSION

`static final public 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
Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, 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

Exceptions

`Covariances.TooManyObsDeletedException` is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative

`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.

`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` (p. 607) class.

`nObservations` – an `int` containing the number of observations associated with `cov`.

StepwiseRegression

```
public StepwiseRegression(double[][] x, double[] y, double[] weights) throws
Covariances.NonnegativeWeightException, Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, 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`

Exceptions

`Covariances.NonnegativeWeightException` is thrown if the weights are negative

`Covariances.TooManyObsDeletedException` is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative

`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

`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 Covariances.NonnegativeFreqException,
Covariances.NonnegativeWeightException, Covariances.TooManyObsDeletedException,
Covariances.MoreObsDelThanEnteredException, 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

Exceptions

Covariances.NonnegativeFreqException is thrown if the frequencies are negative

Covariances.NonnegativeWeightException is thrown if the weights are negative

Covariances.TooManyObsDeletedException is thrown if more observations have been deleted than were originally entered, i.e. the sum of frequencies has become negative

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

Covariances.DiffObsDeletedException is thrown if different observations are being deleted than were originally entered

Methods

compute

```
public void compute() throws StepwiseRegression.NoVariablesEnteredException,
StepwiseRegression.CyclingIsOccurringException
```

Description

Builds the multiple linear regression models using forward selection, backward selection, or stepwise selection.

Exceptions

NoVariablesEnteredException is thrown if no variables entered the model. All elements of ANOVA table are set to NaN.

CyclingIsOccurringException is thrown if cycling occurs

getANOVA

```
public ANOVA getANOVA() throws StepwiseRegression.NoVariablesEnteredException,
StepwiseRegression.CyclingIsOccurringException
```

Description

Gets an analysis of variance table and related statistics.

Returns

an ANOVA table and related statistics

getCoefficientTTests

```
public StepwiseRegression.CoefficientTTests getCoefficientTTests() throws
StepwiseRegression.NoVariablesEnteredException,
StepwiseRegression.CyclingIsOccurringException
```

Description

Returns the student-*t* test statistics for the regression coefficients.

Each row corresponding to a variable not in the model contains statistics for a model which includes the variables of the final model and the variable corresponding to the row in question.

Returns

a `StepwiseRegression.CoefficientTTests` object containing statistics relating to the regression coefficients

getCoefficientVIF

```
public double[] getCoefficientVIF() throws
StepwiseRegression.NoVariablesEnteredException,
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
StepwiseRegression.NoVariablesEnteredException,
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
StepwiseRegression.NoVariablesEnteredException,
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 to 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 the model during the k-th step.

getIntercept

```
public double getIntercept() throws
StepwiseRegression.NoVariablesEnteredException,
StepwiseRegression.CyclingIsOccurringException
```

Description

Returns the intercept.

The intercept is computed as follows:

$$\beta_0 = \bar{y} - \sum_{i=1}^{n} \beta_i \bar{x}_{i-1}$$

where $\bar{y}$ is the mean of the dependent variable `y`, β_i are the coefficients, and $\bar{x}_i$ are the mean values for each independent variable x_i in the final model. If the covariance matrix is used for input, use method `setMean()` to specify the means of the variables. If `x` and `y` are used for input, the means are computed internally and do not need to be specified.

Returns

a `double` containing the intercept

getSwept

```
public double[] getSwept() throws
StepwiseRegression.NoVariablesEnteredException,
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(int[])`.

Parameter

`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.

Parameter

`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.

setMeans

```
public void setMeans(double[] means)
```

Description

Sets the means of the variables.

This is required when the covariance array is input and the intercept `com.imsl.stat.StepwiseRegression.getIntercept` (p. 761) is requested. Otherwise, it is not used.

Parameter

`means` – a `double` array of length *nVars*+1, where *nVars* is the number of independent variables. `means[0]` through `means[`*nVars*`-1]` are the means of the independent variables and `means[`*nVars*`]` is the mean of the dependent variable.

setMethod

```
public void setMethod(int method)
```

Description

Specifies the stepwise selection method, forward, backward, or stepwise Regression.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`tolerance` – a `double` containing the tolerance used for detecting linear dependence
Default: `tolerance` = 2.2204460492503e-16

Example: StepwiseRegression

This example uses a data set from Draper and Smith (1981, pp. 629-630). The method `compute()` is invoked to find the best regression subset from the four candidate variables. The `getSwept()` method is used to label the variables "in" or "out" of the final model.

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

public class StepwiseRegressionEx1 {

    private static void print(String[] labels, ANOVA anova){

        double[] values = anova.getArray();

        for (int i=0; i<values.length-2; i++){
            System.out.printf("%41s%s\n", labels[i], customDecimalFormat(values[i]));
        }
        System.out.println();
    }

    private static String customDecimalFormat(double value) {

        java.text.DecimalFormat df1 = new java.text.DecimalFormat("##,###.###");
        String s = df1.format(value);
        String[] toc = s.split("\\.");
        if (toc.length == 1) {
            s = String.format("%6s", s);
        } else {
            toc[0] = String.format("%6s", toc[0]);
            toc[1] = String.format("%-3s", toc[1]);
            s = toc[0] + "." + toc[1];
        }
        return s;
    }

    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[] rowLabels = {
            "degrees of freedom for regression: ",
            "degrees of freedom for error: ",
            "total degrees of freedom: ",
            "sum of squares for regression: ",
            "sum of squares for error: ",
            "total sum of squares: ",
            "regression mean square: ",
            "error mean square: ",
            "F-statistic: ",
            "p-value: ",
            "R-squared (in percent): ",
            "adjusted R-squared (in percent): ",
            "est. standard deviation of within error: "
        };

        StepwiseRegression sr = new StepwiseRegression(x, y);
        sr.compute();

        System.out.printf("%20s%s\n", " ", "*** ANOVA ***");
        System.out.printf("%45s%s\n", " ", "Value");

        print(rowLabels, sr.getANOVA());

        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);
        }

        String[] cLabels = {"Coef", "Std. Err", "T-Stat", "p-Value"};

        PrintMatrix pm = new PrintMatrix();
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setColumnLabels(cLabels);

        pm.setTitle("*** Coef *** ");
        pm.print(pmf, coef);
        pm.setTitle("*** History *** ");
        pm.print(sr.getHistory());
        pm.setTitle("*** VIF *** ");
        pm.print(sr.getCoefficientVIF());
        pm.setTitle("*** CovS *** ");
        pm.print(sr.getCovariancesSwept());
        System.out.println("*** Intercept ***   " + sr.getIntercept());
    }
}
```

Output

```
                    *** ANOVA ***
                                             Value
```

```
      degrees of freedom for regression:      2
           degrees of freedom for error:     10
               total degrees of freedom:     12
         sum of squares for regression:  2,641.001
               sum of squares for error:     74.762
                   total sum of squares:  2,715.763
                 regression mean square:  1,320.5
                      error mean square:      7.476
                            F-statistic:    176.627
                                p-value:      0
                 R-squared (in percent):     97.247
        adjusted R-squared (in percent):     96.697
est. standard deviation of within error:      2.734

             *** Coef ***
    Coef   Std. Err  T-Stat   p-Value
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

*** 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

*** Intercept ***   103.09738163667471
```

StepwiseRegression.CyclingIsOccurringException class

```
static public class
com.imsl.stat.StepwiseRegression.CyclingIsOccurringException extends
com.imsl.IMSLException
```

Cycling is occurring.

Constructor

StepwiseRegression.CyclingIsOccurringException

```
public StepwiseRegression.CyclingIsOccurringException(int nStep)
```

Description

Constructs a `CyclingIsOccurringException`.

Parameter

`nStep` – an `int` which specifies the number of steps taken.

StepwiseRegression.NoVariablesEnteredException class

```
static public class
com.imsl.stat.StepwiseRegression.NoVariablesEnteredException extends
com.imsl.IMSLException
```

No Variables can enter the model.

Constructor

StepwiseRegression.NoVariablesEnteredException

```
public StepwiseRegression.NoVariablesEnteredException()
```

Description

Constructs a `NoVariablesEnteredException`.

StepwiseRegression.CoefficientTTests class

`public class com.imsl.stat.StepwiseRegression.CoefficientTTests implements Serializable`

`CoefficientTTests()` contains statistics related to the student-t test, for each regression coefficient.

Methods

getCoefficient

`public double getCoefficient(int index)`

Description

Returns the estimate for a coefficient of the independent variable.

Parameter

`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. $H_1 : \beta \neq 0$.

Parameter

`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.

Parameter

`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 ($\beta_{index} = 0$).

Parameter

`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.

Chapter 15: Analysis of Variance

Types

class ANOVA 771
class ANOVAFactorial 781
class ANCOVA 791
class MultipleComparisons 803

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.

ANOVA class

```
public class com.imsl.stat.ANOVA implements Serializable, Cloneable
```

Analysis of Variance table and related statistics.

Fields

BONFERRONI

```
static final public int BONFERRONI
```

The Bonferroni method

DUNN_SIDAK

```
static final public int DUNN_SIDAK
```

The Dunn-Sidak method

ONE_AT_A_TIME

```
static final public int ONE_AT_A_TIME
```

The One-at-a-Time (Fisher's LSD) method

SCHEFFE

```
static final public int SCHEFFE
```

The Scheffe method

TUKEY

```
static final public int TUKEY
```

The Tukey method

TUKEY_KRAMER

```
static final public int TUKEY_KRAMER
```

The Tukey-Kramer method

Constructors

ANOVA

```
public ANOVA(double[][] y)
```

Description

Parameter

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)

getConfidenceInterval

```
public double[] getConfidenceInterval(double conLevel, int i, int j, int
compMethod)
```

Description

Computes the confidence interval associated with the difference of means between two groups using a specified method.

`getConfidenceInterval` computes the simultaneous confidence interval on the pairwise comparison of means μ_i and μ_j in the one-way analysis of variance model. Any of several methods can be chosen. A good review of these methods is given by Stoline (1981). Also the methods are discussed in many elementary statistics texts, e.g., Kirk (1982, pages 114-127). Let s^2 be the estimated variance of a single observation. Let ν be the degrees of freedom associated with s^2. Let

$$\alpha = 1 - \frac{conLevel}{100.0}$$

The methods are summarized as follows:

Tukey method: The Tukey method gives the narrowest simultaneous confidence intervals for the pairwise differences of means $\mu_i - \mu_j$ in balanced $(n_1 = n_2 = \ldots = n_k = n)$ one-way designs. The

method is exact and uses the Studentized range distribution. The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm q_{1-\alpha;k,v}\sqrt{\frac{s^2}{n}}$$

where $q_{1-a,k,v}$ is the $(1-\alpha)100$ percentage point of the Studentized range distribution with parameters k and v. If the group sizes are unequal, the Tukey-Kramer method is used instead.

Tukey-Kramer method: The Tukey-Kramer method is an approximate extension of the Tukey method for the unbalanced case. (The method simplifies to the Tukey method for the balanced case.) The method always produces confidence intervals narrower than the Dunn-Sidak and Bonferroni methods. Hayter (1984) proved that the method is conservative, i.e., the method guarantees a confidence coverage of at least $(1-\alpha)\,100\%$. Hayter's proof gave further support to earlier recommendations for its use (Stoline 1981). (Methods that are currently better are restricted to special cases and only offer improvement in severely unbalanced cases, see, e.g., Spurrier and Isham 1985). The formula for the difference $\mu_i - \mu_j$ is given by the following:

$$\bar{y}_i - \bar{y}_j \pm q_{1-\alpha;v,k}\sqrt{\frac{s^2}{2n_i} + \frac{s^2}{2n_j}}$$

Dunn-Sidak method: The Dunn-Sidak method is a conservative method. The method gives wider intervals than the Tukey-Kramer method. (For large v and small α and k, the difference is only slight.) The method is slightly better than the Bonferroni method and is based on an improved Bonferroni (multiplicative) inequality (Miller, pages 101, 254-255). The method uses the *t* distribution. The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm t_{\frac{1}{2}+\frac{1}{2}(1-\alpha)^{1/k^*};v}\sqrt{\frac{s^2}{n_i} + \frac{s^2}{n_j}}$$

where $t_{f;v}$ is the 100*f* percentage point of the *t* distribution with v degrees of freedom.

Bonferroni method: The Bonferroni method is a conservative method based on the Bonferroni (additive) inequality (Miller, page 8). The method uses the *t* distribution. The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm t_{1-\frac{\alpha}{2k^*};v}\sqrt{\frac{s^2}{n_i} + \frac{s^2}{n_j}}$$

Scheffé method: The Scheffé method is an overly conservative method for simultaneous confidence intervals on pairwise difference of means. The method is applicable for simultaneous confidence intervals on all contrasts, i.e., all linear combinations

$$\sum_{i=1}^{k} c_i\mu_i$$

where the following is true:

$$\sum_{i=1}^{k} c_i = 0$$

The method can be recommended here only if a large number of confidence intervals on contrasts in addition to the pairwise differences of means are to be constructed. The method uses the *F* distribution. The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm \sqrt{(k-1)\,F_{1-\alpha;k-1,\nu}\left(\frac{s^2}{n_i} + \frac{s^2}{n_j}\right)}$$

where $F_{1-a;(k-1),\nu}$ is the $(1-\alpha)\,100$ percentage point of the *F* distribution with $k-1$ and ν degrees of freedom.

One-at-a-time *t* method (Fisher's LSD): The one-at-a-time *t* method is the method appropriate for constructing a single confidence interval. The confidence percentage input is appropriate for one interval at a time. The method has been used widely in conjunction with the overall test of the null hypothesis $\mu_1 = \mu_2 = \ldots = \mu_k$ by the use of the F statistic. Fisher's LSD (least significant difference) test is a two-stage test that proceeds to make pairwise comparisons of means only if the overall *F* test is significant. Milliken and Johnson (1984, page 31) recommend LSD comparisons after a significant *F* only if the number of comparisons is small and the comparisons were planned prior to the analysis. If many unplanned comparisons are made, they recommend Scheffe's method. If the *F* test is insignificant, a few planned comparisons for differences in means can still be performed by using either Tukey, Tukey-Kramer, Dunn-Sidak or Bonferroni methods. Because the *F* test is insignificant, Scheffe's method will not yield any significant differences. The formula for the difference $\mu_i - \mu_j$ is given by

$$\bar{y}_i - \bar{y}_j \pm t_{1-\frac{\alpha}{2};\nu}\sqrt{\frac{s^2}{n_i} + \frac{s^2}{n_j}}$$

Parameters

`conLevel` – a `double` specifying the confidence level for simultaneous interval estimation. If the Tukey method for computing the confidence intervals on the pairwise difference of means is to be used, `conLevel` must be in the range [90.0, 99.0]. Otherwise, `conLevel` must be in the range [0.0, 100.0). One normally sets this value to 95.0.

`i` – is an `int` indicating the *i*-th member of the pair difference, $\mu_i - \mu_j$. `i` must be a valid group index.

`j` – is an `int` indicating the *j*-th member of the pair difference, $\mu_i - \mu_j$. `j` must be a valid group index.

`compMethod` – must be one of the following:

compMethod	**Description**
TUKEY	Uses the Tukey method. This method is valid for balanced one-way designs.
TUKEY_KRAMER	Uses the Tukey-Kramer method. This method simplifies to the Tukey method for the balanced case.
DUNN_SIDAK	Uses the Dunn-Sidak method.
BONFERRONI	Uses the Bonferroni method.
SCHEFFE	Uses the Scheffe method.
ONE_AT_A_TIME	Uses the One-at-a-Time (Fisher's LSD) method.

Returns

a `double` array containing the group numbers, difference of means, and lower and upper confidence limits.

Array Element	Description
0	Group number for the *i*-th mean.
1	Group number for the *j*-th mean.
2	Difference of means (*i*-th mean) - (*j*-th mean).
3	Lower confidence limit for the difference.
4	Upper confidence limit for the difference.

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

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= 0.002768882525349784
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
```

ANOVAFactorial class

`public class com.imsl.stat.ANOVAFactorial implements Serializable, Cloneable`

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.

Fields

POOL_INTERACTIONS

`static final public int POOL_INTERACTIONS`

Indicates factor `nSubscripts` is not error.

PURE_ERROR

`static final public int PURE_ERROR`

Indicates factor `nSubscripts` is 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.

Exception

`IllegalArgumentException` is thrown if `nLevels.length`, and `y.length` are not consistent

Methods

compute

`final public 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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` matrix containing statistics relating to the sums of squares for the effects in the model. Here,

$$\text{NEF} = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{\min(n, |\text{model_order}|)}$$

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.

Parameter

`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.

Parameter

`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

$$\sum_{i=1}^{2} \alpha_i = 0; \sum_{j=1}^{3} \beta_j = 0; \sum_{i=1}^{2} \gamma_{ij} = 0 \text{ for } j = 1,\ 2,\ 3;$$

and

$$\sum_{j=1}^{3} \gamma_{ij} = 0 \text{ for } j = 1,\ 2$$

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.*;

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.*;

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.*;
```

```
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
```

ANCOVA class

```
public class com.imsl.stat.ANCOVA implements Serializable, Cloneable
```

Analyzes a one-way classification model with covariates. Class `ANCOVA` performs analysis for models that combine the features of a oneway analysis of variance model with that of a multiple linear regression model. The basic one-way analysis of covariance model is

$$y_{ij} = \beta_{0i} + \beta_1 x_{ij1} + \beta_2 x_{ij2} + \ldots + \beta_m x_{ijm} + \varepsilon_{ij}$$
$$i = 1, 2, \ldots, ngroup$$
$$j = 1, 2, \ldots, n_i$$

where, *ngroup* is the number of treatment groups, the observed value of y_{ij} constitutes the *j*-th response in the *i* -th group, β_{0i} denotes the y intercept for the regression function for the *i*-th group, β_1, β_2, ..., β_m are the regression coefficients for the covariates, and the ε_{ij}'s are independently distributed normal errors with mean zero and variance σ^2. This model allows the regression function for each group to have different intercepts. However, the remaining *m* regression coefficients are the same for each group, i.e., the regression functions are parallel.

In practice, sometimes the regression functions are not parallel. In addition to estimates for the model assuming parallelism (parallel regression planes), `ANCOVA` computes estimates and summary statistics for the separate regressions of each group. These estimates can be examined using the methods `getCoefficientTables` and `getANOVATables`.

Estimates for the β_{0i}'s and β_1, β_2, ..., β_m in the model assuming parallelism are returned using the method `getModelCoefficients`. Summary statistics are also computed for this model and returned by the `compute` method.

The adjusted group means, stored in the last column of *xymean*, are computed using the formula:

$$\hat{\beta}_{0i} + \hat{\beta}_1 \bar{x}_1 + \hat{\beta}_2 \bar{x}_2 + \ldots + \hat{\beta}_{ncov} \bar{x}_{ncov}$$

where *xymean* is the matrix returned by `getMeans` and *ncov* is the number of covariates.

The estimated covariance between the i_1-th and i_2-th adjusted group mean is given by

$$v_{i_1 i_2} + \sum_{r=1}^{m} \sum_{s=1}^{m} \bar{x}_r v_{k+r,k+s} \bar{x}_s + \sum_{r=1}^{m} \bar{x}_r v_{i_1,k+r} + \sum_{r=1}^{m} \bar{x}_r v_{i_2,k+r}$$

where v_{pq} is the entry in `covb[p-1][q-1]`, where `covb` is returned by `getVarCovCoefficients` and is the estimated covariance between the *p*-th and *q*-th estimated coefficients in the regression function.

Constructor

ANCOVA

```
public ANCOVA(double[][] responses, double[][][] covariates)
```

Description

Constructs a one-way classification model with covariates.

Parameters

`responses` – a `double` matrix containing the responses. Each row in `responses` corresponds to a treatment group. Each row of `responses` can contain a different number of observations. There must be at least two groups (*responses.length* > 1).

`covariates` – is a three-dimensional `double` array containing the covariates. The first dimension corresponds to the number of covariates (consider each element an individual covariate matrix or `covariates.length` = number of covariates). Each row in `covariates[i]` corresponds to a treatment group. There must be the same number of rows, for each covariate, as there are in the responses matrix (`covariates[i].length = responses.length`). There must be at least one covariate (*covariate.length* > 1). Each row of `covariates[i]` must contain the same number of elements as the corresponding row in `responses`
(`covariates[i][j].length = responses[j].length`).

Methods

compute

`public double[] compute()`

Description

Performs one-way analysis of covariance assuming parallelism and returns an array containing the parallelism tests for the one-way analysis of covariance.

Returns

a `double` array containing the parallelism tests for the one-way analysis of covariance organized as follows:

index	Description
0	Extra degrees of freedom for model not assuming parallelism.
1	Degrees of freedom for error for model not assuming parallelism.
2	Degrees of freedom for error for model assuming parallelism.
3	Extra sum of squares for model not assuming parallelism.
4	Sum of squares for error for model not assuming parallelism.
5	Sum of squares for error for model assuming parallelism.
6	Mean square for *index* = 0.
7	Mean square for *index* = 1.
8	F statistic
9	p-value

getANCOVA

`public double[] getANCOVA()`

Description

Returns an array containing the one-way analysis of covariance assuming parallelism.

Returns

a `double` array of length 15 containing the following values:

index	ANCOVA Table 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)

getANOVATables

```
public double[][] getANOVATables()
```

Description

Returns a matrix of size *ngroup* by 15 containing the analysis of variance tables for each linear regression model fitted separately to each treatment group.

Returns

a `double` matrix containing the analysis of variance tables for each linear regression model fitted separately to each treatment group. The 15 values in the *i*-th row are for treatment group *i* organized as follows:

index	Description
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)

getAdjustedANOVA

```
public double[] getAdjustedANOVA()
```

Description

Returns the partial sum of squares for the one-way analysis of covariance.

Returns

a `double` array containing the partial sum of squares for the one-way analysis of covariance organized as follows:

index	Description
0	Degrees of freedom for groups after covariates.
1	Degrees of freedom for covariates after groups.
2	Sum of squares for groups after covariates.
3	Sum of squares for model (groups and covariates combined).
4	*F*-statistic for groups.
5	*F*-statistic for covariates.
6	*p*-value for groups.
7	*p*-value for covariates.

getCoefficientTable

```
public double[][] getCoefficientTable(int group)
```

Description

Returns a matrix of size $ncov + 1$ by 4 containing statistics for a linear regression model fitted separately for each of the *ngroup* treatment groups.

Parameter

`group` – an `int` specifying the group that regression statistics will be retreived for.

Returns

a `double` matrix containing statistics for a `group`. Each row corresponds to the model coefficients. For row = 0, the statistics relate to the intercept in the regression model. For row = 1, 2, ..., *ncov*, the statistics relate to the slopes for the covariates. The column dimension corresponds to the row described for `getModelCoefficients` as follows:

Column	Description
0	Coefficient estimate.
1	Estimated standard error of the estimate.
2	*t*-statistic.
3	*p*-value.

getCoefficientTables

`public double[][][] getCoefficientTables()`

Description

Returns an array containing statistics for a linear regression model fitted separately for all *ngroup* treatments.

Returns

a `double[][][]` array containing statistics for a linear regression model fitted separately for each of the *ngroup* treatment groups. The 3 dimensional array organized with *ngroup* rows, *ncov* + 1 columns, and depth of 4. Each row corresponds to one of the *ngroup* treatment groups. Each column corresponds to the model coefficients. For column = 0, the statistics relate to the intercept in the regression model. For column = 1, 2, ..., *ncov*, the statistics relate to the slopes for the covariates. The depth dimension corresponds to the columns described for `getModelCoefficients` as follows:

Column	Description
0	Coefficient estimate.
1	Estimated standard error of the estimate.
2	*t*-statistic.
3	*p*-value.

getMeans

`public double[][] getMeans()`

Description

Returns a matrix containing the unadjusted means for the covariates and the response variate and the means for the response variate adjusted for the covariates.

Returns

a `double` matrix of size *ngroup* + 1 by *ncov* + 3 containing the unadjusted means for the covariates and the response variate and the means for the response variate adjusted for the covariates. Each row for i = 0, 1, ..., *ngroup* - 1 corresponds to a group. Row *ngroup* contains overall statistics. The means are organized in columns as follows:

Column	Description
0	Number of non-missing cases
1 thru *ncov*	Covariate means.
ncov + 1	Response mean.
ncov + 2	Response mean adjusted assuming parallelism.

getModelCoefficients

```
public double[][] getModelCoefficients()
```

Description

Returns a matrix containing statistics for the regression coefficients for the model assuming parallelism.

Returns

a `double` matrix of size *ngroup* + *ncov* by 4 containing statistics for the regression coefficients for the model assuming parallelism. Each row corresponds to a coefficient in the model. For $i = 0, 1, ..., ngroup - 1$, row `i` is for the y intercept for the i -th group. The remaining *ncov* rows are for the covariate coefficients. The statistics in the columns are organized as follows:

Column	Description
0	Coefficient estimate.
1	Estimated standard error of the estimate.
2	t-statistic.
3	p-value.

getNumberOfMissing

```
public int getNumberOfMissing()
```

Description

Returns the number of cases with missing values in `covariates` or `responses`. Cases with any missing values are not used in the analysis.

Returns

an `int` scalar value indicating the number of cases with missing values in `covariates` or `responses`.

getR

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

Description

Returns the R matrix from the QR decomposition. The R matrix is from the regression assuming parallelism.

Returns

a `double` matrix of size *ngroup* + *ncov* by *ngroup* + *ncov* which contains the R from the QR decomposition. The R matrix is from the regression assuming parallelism.

getVarCovAdjustedMeans

```
public double[][] getVarCovAdjustedMeans()
```

Description

Returns a matrix containing the estimated variances and covariances for the adjusted means assuming parallelism.

Returns

a `double` matrix of size *ngroup* by *ngroup* containing the estimated variances and covariances for the adjusted means assuming parallelism.

---

getVarCovCoefficients

```
public double[][] getVarCovCoefficients()
```

Description

Returns a matrix containing the estimated variances and covariances for the coefficients returned using `getModelCoefficients`.

Returns

a `matrix` of size *ngroup* + *ncov* by *ngroup* + *ncov* containing the estimated variances and covariances for the coefficients returned using `getModelCoefficients`.

Example 1: One-way analysis of covariance model

This example fits a one-way analysis of covariance model assuming parallelism using data discussed by Snedecor and Cochran (Table 14.6.1, pages 432-436). The responses are concentrations of cholesterol (in mg/100 ml) in the blood of two groups of women: women from Iowa and women from Nebraska. The age of a woman is the single covariate. The cholesterol concentrations and ages of the women according to state are shown in the following table. (There are 11 Iowa women and 19 Nebraska women in the study. Only the first 5 women from each state are shown here.)

Iowa		Nebraska	
Age	Cholesterol	Age	Cholesterol
46	181	18	137
52	228	44	173
39	182	33	177
65	249	78	241
54	259	51	225

There is no evidence from the data to indicate that the regression lines for cholesterol concentration as a function of age are not parallel for Iowa and Nebraska women (p-value is 0.5425). The parallel line model suggests that Nebraska women may have higher cholesterol concentrations than Iowa women. The cholesterol concentrations (adjusted for age) are 195.5 for Iowa women versus 224.2 for Nebraska women. The difference is 28.7 with an estimated standard error of

$$\sqrt{170.4+97.4-2(2.9)} = 16.1$$

```
import com.imsl.stat.ANCOVA;
import com.imsl.math.PrintMatrix;
```

```
public class ANCOVAEx1 {

    public static void main(String args[]) {

        double y[][] = {
            {181.0, 228.0, 182.0, 249.0, 259.0, 201.0, 121.0, 339.0, 224.0,
                112.0, 189.0},
            {137.0, 173.0, 177.0, 241.0, 225.0, 223.0, 190.0, 257.0, 337.0,
                189.0, 214.0, 140.0, 196.0, 262.0, 261.0, 356.0, 159.0, 191.0,
                197.0}};
        double x[][][] = new double[1][2][];
        x[0][0] = new double[]{46.0, 52.0, 39.0, 65.0, 54.0, 33.0, 49.0, 76.0,
            71.0, 41.0, 58.0};
        x[0][1] = new double[]{18.0, 44.0, 33.0, 78.0, 51.0, 43.0, 44.0, 58.0,
            63.0, 19.0, 42.0, 30.0, 47.0, 58.0, 70.0, 67.0, 31.0, 21.0,
            56.0};
        ANCOVA awc = new ANCOVA(y, x);

        double testpl[] = awc.compute();
        double aov[] = awc.getANCOVA();

        System.out.println("             * * * Analysis of Variance * * *\n");
        System.out.println("                  Sum of         "
                + "Mean                Prob of");
        System.out.println("Source   DF      Squares        Square    "
                + "Overall F  Larger F");
        System.out.printf("Model  %3.0f     %10.2f      %9.2f  %2.2f      "
                + "%8.6f\n", aov[0], aov[3], aov[6], aov[8], aov[9]);
        System.out.printf("Error   %3.0f    %10.2f     %9.2f \n", aov[1],
                aov[4], aov[7]);
        System.out.printf("Total   %3.0f     %10.2f \n\n\n", aov[2], aov[5]);
        System.out.println("           * * * Test for Parallelism  * * *\n");
        System.out.println("                       Sum of     Mean     F     "
                + " Prob of");
        System.out.println("Source           DF    Squares   Square   Test   "
                + " Larger F");
        System.out.println("Extra due to");
        System.out.printf("Nonparallelism %3.0f %10.2f   %7.2f   %7.5f %5.4f\n",
                testpl[0], testpl[3], testpl[6], testpl[8], testpl[9]);
        System.out.println("Extra Assuming");
        System.out.printf("Nonparallelism  %3.0f  %10.2f  %7.2f \n", testpl[1],
                testpl[4], testpl[7]);
        System.out.println("Error Assuming");
        System.out.printf("Parallelism     %3.0f  %10.2f \n\n\n", testpl[2],
                testpl[5]);
        new PrintMatrix("XY Mean Matrix\n").print(awc.getMeans());
        new PrintMatrix("\n\nVar./Covar. Matrix of Adjusted Group Means"
                + "\n").print(awc.getVarCovAdjustedMeans());
    }
}
```

Output

```
             * * * Analysis of Variance * * *
```

```
                 Sum of         Mean                Prob of
Source   DF      Squares        Square    Overall F  Larger F
Model    2       54432.75       27216.38  14.97      0.000042
Error    27      49103.91       1818.66
Total    29      103536.67


           * * * Test for Parallelism  * * *

                       Sum of     Mean     F        Prob of
Source          DF    Squares   Square   Test     Larger F
Extra due to
Nonparallelism   1     709.05    709.05   0.38093 0.5425
Extra Assuming
Nonparallelism   26    48394.86  1861.34
Error Assuming
Parallelism      27    49103.91


         XY Mean Matrix

   0     1        2        3
0  11  53.091  207.727  195.521
1  19  45.947  217.105  224.172
2  30  48.567  213.667  213.667



Var./Covar. Matrix of Adjusted Group Means

      0       1
0  170.368  -2.915
1   -2.915  97.407
```

Example 2: One-way analysis of covariance model

This example fits a one-way analysis of covariance model and performs a test for parallelism using data discussed by Snedecor and Cochran (1967, Table 14.8.1, pages 438-443). The responses are weight gains (in pounds per day) of 40 pigs for four groups of pigs under varying treatments. Two covariates-initial age (in days) and initial weight (in pounds) are used. For each treatment, there are 10 pigs. Only the first 5 pigs from each treatment are shown here.

Treatment 1			Treatment 2			Treatment 3			Treatment 4		
Age	Wt.	Gain	Age	Wt.	Gain	Age	Wt.	Gain	Age	Wt.	Gain
78	61	1.4	78	74	1.61	78	80	1.67	77	62	1.4
90	59	1.79	99	75	1.31	83	61	1.41	71	55	1.47
94	76	1.72	80	64	1.12	79	62	1.73	78	62	1.37
71	50	1.47	75	48	1.35	70	47	1.23	70	43	1.15
99	61	1.26	94	62	1.29	85	59	1.49	95	57	1.22

For these data, the test for non-parallelism is not statistically significant (p = 0.901). The one-way analysis of covariance test for the treatment means adjusted for the covariates, assuming parallel slopes, is statistically significant at a stated significance level of $\alpha = 0.05$, (p = 0.04931). Multiple comparisons can be done using the least significant difference approach of comparing each pair of treatment groups with the two-sample student-t test. Since the adjusted means in the one-way analysis of covariance are correlated, the standard error for these comparisons must be computed using the variances and covariances in covm. The standard errors for these comparisons are fairly similar ranging from 0.0630 to 0.0638. The Student's t comparisons identify differences between groups 1 and 2, and 1 and 4 as being statistically significant with p-values of 0.01225 and 0.03854 respectively.

```
import com.imsl.stat.ANCOVA;

public class ANCOVAEx2 {

    public static void main(String[] args) {
        int j;
        int ngroup = 4;
        double x1[][] = {
            {78.0, 90.0, 94.0, 71.0, 99.0, 80.0, 83.0, 75.0, 62.0, 67.0},
            {78.0, 99.0, 80.0, 75.0, 94.0, 91.0, 75.0, 63.0, 62.0, 67.0},
            {78.0, 83.0, 79.0, 70.0, 85.0, 83.0, 71.0, 66.0, 67.0, 67.0},
            {77.0, 71.0, 78.0, 70.0, 95.0, 96.0, 71.0, 63.0, 62.0, 67.0}
        };
        double x2[][] = {
            {61.0, 59.0, 76.0, 50.0, 61.0, 54.0, 57.0, 45.0, 41.0, 40.0},
            {74.0, 75.0, 64.0, 48.0, 62.0, 42.0, 52.0, 43.0, 50.0, 40.0},
            {80.0, 61.0, 62.0, 47.0, 59.0, 42.0, 47.0, 42.0, 40.0, 40.0},
            {62.0, 55.0, 62.0, 43.0, 57.0, 51.0, 41.0, 40.0, 45.0, 39.0}
        };
        double y[][] = {
            {1.40, 1.79, 1.72, 1.47, 1.26, 1.28, 1.34, 1.55, 1.57, 1.26},
            {1.61, 1.31, 1.12, 1.35, 1.29, 1.24, 1.29, 1.43, 1.29, 1.26},
            {1.67, 1.41, 1.73, 1.23, 1.49, 1.22, 1.39, 1.39, 1.56, 1.36},
            {1.40, 1.47, 1.37, 1.15, 1.22, 1.48, 1.31, 1.27, 1.22, 1.36}
        };
        double x[][][] = {x1, x2};
        /* setup covariate input matrix */

        ANCOVA awc = new ANCOVA(y, x);

        double testpl[] = awc.compute();
        double aov[] = awc.getANCOVA();
        double adjAov[] = awc.getAdjustedANOVA();
        double xymean[][] = awc.getMeans();
        double covm[][] = awc.getVarCovAdjustedMeans();

        System.out.println("\n           * * * Test for Parallelism  * * *\n");
        System.out.println("                       Sum of     Mean         "
                + "F     Prob of");
        System.out.println("Source           DF    Squares   Square       "
                + "Test   Larger F");
        System.out.println("Extra due to");
        System.out.printf("Nonparallelism %3.0f %10.2f    %7.2f    %7.5f   "
                + "%5.3f\n", testpl[0], testpl[3], testpl[6], testpl[8],
```

```
                testpl[9]);
        System.out.println("Extra Assuming");
        System.out.printf("Nonparallelism %3.0f %10.2f    %7.2f\n",
                testpl[1], testpl[4], testpl[7]);
        System.out.println("Error Assuming");
        System.out.printf("Parallelism    %3.0f %10.2f   \n\n\n", testpl[2],
                testpl[5]);
        System.out.println("             * * * Analysis of Variance * * *\n");
        System.out.println("                  Sum of        "
                + "Mean            Prob of");
        System.out.println("Source   DF     Squares       Square    "
                + "Overall F  Larger F");
        System.out.printf("Model   %3.0f      %f     %f    %f    %5.4f\n",
                aov[0], aov[3], aov[6], aov[8], aov[9]);
        System.out.printf("Error   %3.0f      %f     %f      \n", aov[1],
                aov[4], aov[7]);
        System.out.printf("Total   %3.0f      %f  \n\n\n", aov[2], aov[5]);
        System.out.println("         * * * Adjusted Analysis of Variance  * "
                + "* *\n");
        System.out.println("                                 Sum of       "
                + "F    Prob of");
        System.out.println("Source                    DF    Squares      "
                + "Test   Larger F");
        System.out.printf("Groups after Covariates %3.0f   %10.2f    %5.2f   "
                + " %7.5f\n", adjAov[0], adjAov[2], adjAov[4], adjAov[6]);
        System.out.printf("Covariates after Groups %3.0f   %10.2f    %5.2f   "
                + " %7.5f\n\n\n", adjAov[1], adjAov[3], adjAov[5], adjAov[7]);
        System.out.println("           * * * Group Means * * *\n");
        System.out.println("GROUP  | Unadjusted   |  Adjusted |  Std. Error");
        for (int i = 0; i < ngroup; i++) {
            double se = Math.sqrt(covm[i][i]);
            System.out.printf("  %d    |   %5.4f     |   %5.4f  |   %7.4f\n",
                    i + 1, xymean[i][ngroup - 1], xymean[i][ngroup], se);
        }
        System.out.println("\n\n      * * * Student-t Multiple Comparisons * "
                + "* *\n");
        System.out.println(" Groups  |    Diff   | Std. Error | Student-t | "
                + "P-Value");
        for (int i = 0; i < ngroup - 1; i++) {
            for (j = i + 1; j < ngroup; j++) {
                double delta = xymean[i][ngroup] - xymean[j][ngroup];
                double se = Math.sqrt(covm[i][i] + covm[j][j] - 2.0
                        * covm[i][j]);
                double t = delta / se;
                double df = xymean[i][0] + xymean[j][0] - 2;
                double pvalue = 1.0 - com.imsl.stat.Cdf.studentsT(t, df);
                System.out.printf(" %d vs %d  |  %7.4f  |  %7.4f   | %7.3f   "
                        + "| %7.5f\n", i + 1, j + 1, delta, se, t, pvalue);
            }
        }
    }
}
```

Output

```
          * * * Test for Parallelism  * * *

                       Sum of     Mean         F     Prob of
Source           DF    Squares   Square       Test   Larger F
Extra due to
Nonparallelism   6       0.05       0.01    0.35534   0.901
Extra Assuming
Nonparallelism  28       0.62       0.02
Error Assuming
Parallelism     34       0.67


            * * * Analysis of Variance * * *

                  Sum of        Mean                Prob of
Source   DF      Squares        Square    Overall F  Larger F
Model     5      0.352517     0.070503    3.576390    0.0105
Error    34      0.670261     0.019714
Total    39      1.022777


        * * * Adjusted Analysis of Variance  * * *

                               Sum of       F     Prob of
Source                    DF    Squares      Test   Larger F
Groups after Covariates   3         0.17     2.90    0.04931
Covariates after Groups   2         0.17     4.44    0.01939


          * * * Group Means * * *

GROUP  | Unadjusted    |  Adjusted |  Std. Error
  1    |   1.4640      |   1.4614  |    0.0448
  2    |   1.3190      |   1.3068  |    0.0446
  3    |   1.4450      |   1.4429  |    0.0447
  4    |   1.3250      |   1.3418  |    0.0449


     * * * Student-t Multiple Comparisons * * *

 Groups  |    Diff    | Std. Error | Student-t | P-Value
 1 vs 2  |   0.1546  |   0.0630   |   2.455   | 0.01225
 1 vs 3  |   0.0185  |   0.0637   |   0.290   | 0.38750
 1 vs 4  |   0.1196  |   0.0638   |   1.875   | 0.03854
 2 vs 3  |  -0.1362  |   0.0632   |  -2.153   | 0.97743
 2 vs 4  |  -0.0350  |   0.0638   |  -0.549   | 0.70528
 3 vs 4  |   0.1011  |   0.0631   |   1.602   | 0.06330
```

MultipleComparisons class

```
public class com.imsl.stat.MultipleComparisons implements Serializable,
Cloneable
```

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).

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 σ^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

```
final public 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

Parameter

`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 16: Categorical and Discrete Data Analysis

Types

class ContingencyTable 805
class CategoricalGenLinModel 818

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.

ContingencyTable class

```
public class com.imsl.stat.ContingencyTable implements Serializable, Cloneable
```

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 χ^2 is given as $(x_{ij} - e_{ij})^2 / e_{ij}$. The Pearson chi-squared statistic (denoted χ^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 χ^2 is too large.

The maximum likelihood equivalent of χ^2, G^2 is computed as follows:

$$G^2 = -2 \sum_{i,j} x_{ij} \ln(x_{ij} / np_{ij})$$

G^2 is asymptotically equivalent to χ^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:

$$\text{phi}, \phi = \sqrt{\chi^2 / n}$$

$$\text{contingency coefficient}, P = \sqrt{\chi^2 / (n + \chi^2)}$$

$$\text{Cramer's}V, V = \sqrt{\chi^2 / (n \min(r, c))}$$

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 χ^2 statistic, return value from the `getChiSquared` method.

The distribution of the χ^2 statistic in finite samples approximates a chi-squared distribution. To compute the exact mean and standard deviation of the χ^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, ϕ, 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 τ_b, Stuart's τ_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 τ_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 τ_b cannot be 1.0 (if all marginal totals are positive). For this reason, Stuart suggested a modification to the denominator of τ in which the denominator becomes the largest possible value of the "covariance." This maximizing value is approximately $n^2 m/(m-1)$, where $m = min\ (r,\ c)$. Stuart's τ_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 λ 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 λ 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 λ 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 τ: 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:

$$n/2 - (\sum_i x_{i\bullet}^2)/(2n)$$

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 τ 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 τ 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:

$$U_{r|c} = \frac{\sum_{i,j} x_{ij} \log(x_{i\bullet} x_{\bullet j}/n x_{ij})}{\sum_i x_{i\bullet} \log(x_{i\bullet}/n)}$$

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.

Constructor

ContingencyTable

```
public ContingencyTable(double[][] table)
```

Description

Constructs and performs a chi-squared analysis of a two-way contingency table.

Parameter

`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 τ_b
2	Stuart's τ_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 τ for rows (given columns)
8	Goodman and Kruskal τ for columns (given rows)
9	uncertainty coefficient U (symmetric)
10	uncertainty $U_{r\|c}$ (rows)
11	uncertainty $U_{c\|r}$ (columns)
12	optimal prediction λ (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
```

CategoricalGenLinModel class

`public class com.imsl.stat.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, β 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 γ_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)	$\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$	$f(y) = \binom{S+y-1}{y-1} \theta^S (1-\theta)^y$
MODEL2 (Logarithmic)	$\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$	$f(y) = (1-\theta)^y / (y \ln \theta)$
MODEL3 (Logistic)	$\theta = \frac{e^{w+\eta}}{1+e^{w+\eta}}$	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$
MODEL4 (Probit)	$\theta = \Phi(w+\eta)$	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$
MODEL5 (Log-log)	$\theta = 1 - e^{-e^{w+\eta}}$	$f(y) = \binom{N}{y} \theta^y (1-\theta)^{N-y}$

Here Φ 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 θ, θ 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.

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, θ 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 β. 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 Ψ with respect to γ_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 γ, 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 β_m occurs at

$$\hat{\beta_m} = \infty$$

leading to an infinite estimate of both β_m and η_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 η_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 η_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 η 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 γ_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 γ_i when evaluated at the optimal $\hat{\beta}$ and the derivatives here (and only here) are with respect to γ rather than with respect to β. 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 β.

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.

Fields

MODEL0

```
static final public 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.

MODEL1

```
static final public 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.

MODEL2

```
static final public 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.

MODEL3

```
static final public 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.

MODEL4

```
static final public 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.

MODEL5

```
static final public 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 γ 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^{γ}
Logistic	$e^{\gamma}/(1+e^{\gamma})$
Probit	$\Phi(\gamma)$ (where Φ 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
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.

Exception

`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
CategoricalGenLinModel.ClassificationVariableException
```

Description

Returns the distinct values of the classification variables in ascending order.

Returns

A `double` array of length $\sum_{k=0}^{\text{nclvar}} \text{nclval[k]}$ 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.

Exception

`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
CategoricalGenLinModel.ClassificationVariableException,
CategoricalGenLinModel.ClassificationVariableLimitException,
CategoricalGenLinModel.ClassificationVariableValueException,
CategoricalGenLinModel.DeleteObservationsException,
CategoricalGenLinModel.RankDeficientException, SVD.DidNotConvergeException
```

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.

Exceptions

`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

`ClassificationVariableLimitException` is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, `maxcl`

`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	ρ - value associated with the normal score in column 2.

getProduct

```
public double[] getProduct() throws
CategoricalGenLinModel.ClassificationVariableException,
CategoricalGenLinModel.ClassificationVariableLimitException,
CategoricalGenLinModel.ClassificationVariableValueException,
CategoricalGenLinModel.DeleteObservationsException,
CategoricalGenLinModel.RankDeficientException, SVD.DidNotConvergeException
```

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.

Exceptions

`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

`ClassificationVariableLimitException` is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, `maxcl`

`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.

Parameter

`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.

Exception

`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.

Parameter

`indcl` – An `int` vector which contains the column numbers in `x` that are classification variables. By default this vector is not referenced.

Exception

`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.

Parameter

`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.

Exception

`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 $\sum_{k=0}^{\text{nef}-1} \text{nvef}[k]$ 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.

Exception

`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.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`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.

Exception

`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.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`IllegalArgumentException` is thrown if `maxIterations` is less than or equal to 0

setModelIntercept

`public void setModelIntercept(int intcep)`

Description

Sets the intercept option.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`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.

Parameter

`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.

Exception

`IllegalArgumentException` is thrown when `ipar` is less than 0 or greater than or equal to the number of columns of `x`

setTolerance

```
public void setTolerance(double tol)
```

Description

Initializes the tolerance used in determining linear dependence.

Parameter

`tol` – An `double` value used in determining linear dependence. When linear dependence is detected, a RankDeficientException is thrown and no results are computed. Computations for a rank deficient model can be forced to continue by specifying a negative tolerance. If `tol` is negative, the absolute value of `tol` will be used to determine linear dependence, but computations will proceed with warning RankDeficientWarning. In this case the results should be carefully inspected and used with caution. If this member function is not called, `tol` will be set to .2220446049250313080 8e-14.

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.

Parameter

`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 if method `setClassificationVariableColumn` has not been referenced. Otherwise, the default upper bound is set to `nobs * nclvar` where `nobs` is the number of observations and `nclvar` is the number of classification variables.

Exception

`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.

Parameter

`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.

Exception

`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
CategoricalGenLinModel.ClassificationVariableException,
CategoricalGenLinModel.ClassificationVariableLimitException,
CategoricalGenLinModel.ClassificationVariableValueException,
CategoricalGenLinModel.DeleteObservationsException,
CategoricalGenLinModel.RankDeficientException, SVD.DidNotConvergeException
```

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	ρ - value associated with the normal score in column 2.

Exceptions

`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

`ClassificationVariableLimitException` is thrown when the sum of the number of distinct values taken on by each classification variable exceeds the maximum allowed, `maxcl`

`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 disulphide. Both a logit and a probit fit are produced for linear model $\mu + \beta x$. The data is given as

Covariate(x)	N	y
1.690	59	6
1.724	60	13
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 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 = {
            {1.690, 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 -18.77817904233393
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 -18.23235457438456
    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 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 -3.1146384925784423
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
```

CategoricalGenLinModel.ClassificationVariableException class

```
static public class
com.imsl.stat.CategoricalGenLinModel.ClassificationVariableException extends
com.imsl.IMSLException
```

The ClassificationVariable vector has not been initialized.

Constructor

CategoricalGenLinModel.ClassificationVariableException

```
public CategoricalGenLinModel.ClassificationVariableException()
```

Description

Constructs a `ClassificationVariableException`.

CategoricalGenLinModel.ClassificationVariableLimitException class

```
static public class
com.imsl.stat.CategoricalGenLinModel.ClassificationVariableLimitException
extends com.imsl.IMSLException
```

The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

Constructor

CategoricalGenLinModel.ClassificationVariableLimitException

```
public CategoricalGenLinModel.ClassificationVariableLimitException(int maxcl)
```

Description

Constructs a `ClassificationVariableLimitException`.

Parameter

`maxcl` – An `int` which specifies the upper bound.

CategoricalGenLinModel.ClassificationVariableValueException class

```
static public class
com.imsl.stat.CategoricalGenLinModel.ClassificationVariableValueException
extends com.imsl.IMSLException
```

The number of distinct values for each Classification Variable must be greater than 1.

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.

CategoricalGenLinModel.DeleteObservationsException class

```
static public class
com.imsl.stat.CategoricalGenLinModel.DeleteObservationsException extends
com.imsl.IMSLException
```

The number of observations to be deleted (set by `setObservationMax`) has grown too large.

Constructor

CategoricalGenLinModel.DeleteObservationsException

```
public CategoricalGenLinModel.DeleteObservationsException(int nmax)
```

Description

Constructs a `DeleteObservationsException`.

Parameter

`nmax` – An `int` which specifies the maximum number of observations that can be handled in the linear programming as set by `setObservationMax`.

CategoricalGenLinModel.RankDeficientException class

```
static public class com.imsl.stat.CategoricalGenLinModel.RankDeficientException
extends com.imsl.IMSLException
```

The model has been determined to be rank deficient.

Constructor

CategoricalGenLinModel.RankDeficientException

```
public CategoricalGenLinModel.RankDeficientException(int rank)
```

Description

Constructs a `RankDeficientException`.

Parameter

`rank` – An `int` which specifies the rank of the model.

Chapter 17: Nonparametric Statistics

Types

class SignTest 845
class WilcoxonRankSum 849

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.

SignTest class

```
public class com.imsl.stat.SignTest implements Serializable, Cloneable
```

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:

- $H_0 : Pr(x \le q) \ge p$ (the p-th quantile is at least q)
  $H_1 : Pr(x \le q) < p$
  Reject H_0 if *probability* is less than or equal to the significance level
- $H_0 : Pr(x \le q) \le p$ (the p-th quantile is at least q)
  $H_1 : Pr(x \le q) > p$
  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)
  $H_1 : Pr((x \le q) < p)$ or $Pr((x \le q) > p)$
  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 $q = c$.

Constructor

SignTest

`public SignTest(double[] x)`

Description

Constructor for `SignTest`.

Parameter

`x` – A `double` array containing the data.

Methods

compute

```
final public 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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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.

Parameter

`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.

Parameter

`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
```

WilcoxonRankSum class

```
public class com.imsl.stat.WilcoxonRankSum implements Serializable, Cloneable
```

Performs a Wilcoxon rank sum test.

Class `WilcoxonRankSum` performs the Wilcoxon rank sum test for identical population distribution functions. The Wilcoxon test and the Mann-Whitney *U* test are equivalent. 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 uses the average rank of the tied observations for handling tied observations between samples. 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.

The p-value returned in `stat[9]` is the two-sided p-value calculated using the normal approximation with the normal score returned in `stat[8]`.

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. Another output statistic should be used, (`stat[0]` or `stat[3]`, where `stat` is the array containing the statistics returned from the `getStatistics` method), if another method for handling ties is desired.

Test	Null Hypothesis	Alternative Hypothesis	Action
1	$H_0 : \Pr(x < y) = 0.5$ $H_0 : E(x) = E(y)$	$H_1 : \Pr(x < y) \neq 0.5$ $H_1 : E(x) \neq E(y)$	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	$H_0 : \Pr(x < y) \leq 0.5$ $H_0 : E(x) \geq E(y)$	$H_1 : \Pr(x < y) \neq 0.5$ $H_1 : E(x) < E(y)$	Reject if `stat[6]` is too small.
3	$H_0 : \Pr(x < y) \geq 0.5$ $H_0 : E(x) \leq E(y)$	$H_1 : \Pr(x < y) < 0.5$ $H_1 : E(x) > E(y)$	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-values are calculated using either the large-sample normal approximation or the exact probability calculations. The approximate calculation returned by the `compute` method is usually considered adequate when the size of one or both samples is greater than 50. For smaller samples, the exact probability calculations returned by the `computeExactPValues` method are recommended.

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

```
final public double compute()
```

Description

Performs a Wilcoxon rank sum test using an approximate p-value calculation.

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.

computeExactPValues

```
final public double[] computeExactPValues()
```

Description

Performs a Wilcoxon rank sum test using exact p-value calculations.

Returns

A `double` array containing the the exact p-values according to the following table:

Row	p-values
0	The exact left-tailed p-value.
1	The exact right-tailed p-value.
2	The exact two-tailed p-value.

getMannWhitney

```
public double getMannWhitney()
```

Description

Returns the Mann-Whitney test statistic.

Returns

A `double` scalar containing the Mann-Whitney test statistic equivalent to the W statistic with average ranks used in case of ties. Although the test statistics for the Mann-Whitney and Wilcoxon rank sum tests are computed differently, the p-values for these tests are equal since the Wilcoxon test statistic is a linear transformation of the Mann-Whitney test statistic.

getNumberMissingX

```
public int getNumberMissingX()
```

Description

Returns the number of missing observations detected in `x`.

Returns

An `int` scalar containing the number of missing observations in `x`.

getNumberMissingY

```
public int getNumberMissingY()
```

Description

Returns the number of missing observations detected in `y`.

Returns

An `int` scalar containing the number of missing observations in `y`.

getStatistics

```
public double[] getStatistics()
```

Description

Returns the statistics. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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.

Parameter

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. Here, all the statistics are displayed.

```
import java.text.*;
import com.imsl.*;
import com.imsl.stat.*;

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) . ",
            "Approximate Two-sided p-value of W ......... "
        };

        WilcoxonRankSum wilcoxon = new WilcoxonRankSum(x, y);
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMinimumFractionDigits(3);

        // Turn off printing of warning messages.
        Warning.setOut(null);
        double[] exact = wilcoxon.computeExactPValues();
        double[] stat = wilcoxon.getStatistics();

        for (int i = 0; i < 7; i++) {
            System.out.println(labels[i] + " " + nf.format(stat[i]));
        }

        System.out.println("Mann-Whitney statistic ..........."
                + "......... " + nf.format(wilcoxon.getMannWhitney()));

        for (int i = 7; i < 10; i++) {
            System.out.println(labels[i] + " " + nf.format(stat[i]));
        }

        System.out.println("Exact Left-Tailed p-value ..........."
                + "......  " + nf.format(exact[0]));
        System.out.println("Exact Right-Tailed p-value .........."
                + "......  " + nf.format(exact[1]));
        System.out.println("Exact Two-sided p-value ............."
                + "......  " + nf.format(exact[2]));
    }
}
```

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
Mann-Whitney statistic ..................... 19.500
Standard error of W (averaged ranks) ........  4.758
Standard normal score of W (averaged ranks) .  1.471
Approximate Two-sided p-value of W .........  0.141
```

```
Exact Left-Tailed p-value ..................  0.937
Exact Right-Tailed p-value .................  0.079
Exact Two-sided p-value ....................  0.159
```

Chapter 18: Tests of Goodness of Fit

Types

class ChiSquaredTest 857
class NormalityTest 864
class KolmogorovOneSample 869
class KolmogorovTwoSample 872

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.

The Kolmogorov-Smirnov routines in this chapter compute exact probabilities in small to moderate sample sizes. The chi-squared goodness-of-fit test may be used with discrete as well as continuous distributions.

The Kolmogorov-Smirnov and chi-squared goodness-of-fit test routines allow for missing values (NaN, not a number) in the input data. The routines that test for randomness do not allow for missing values.

ChiSquaredTest class

```
public class com.imsl.stat.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.

Constructors

ChiSquaredTest

```
public ChiSquaredTest(CdfFunction cdf, double[] cutpoints, int nParameters)
throws 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. For example, with a binomial distribution `nParameters`=1 if p is estimated from the data and `nParameters`=0 if p is given in advance. The degrees of freedom in χ^2 is:

$$df = n - p - 1$$

where n = number or non-empty cells and p = `nParameters`.

ChiSquaredTest

```
public ChiSquaredTest(CdfFunction cdf, int nCutpoints, int nParameters) throws
ChiSquaredTest.NotCDFException, 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. For example, with a binomial distribution `nParameters`=1 if p is estimated from the data and `nParameters`=0 if p is given in advance. The degrees of freedom in χ^2 is:

$$df = n - p - 1$$

where n = number or non-empty cells and p = `nParameters`.

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 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 ChiSquaredTest.NotCDFException
```

Description

Returns the degrees of freedom in chi-squared. The degrees of freedom (df) in chi-squared is

$$df = n - p - 1$$

where n = number or non-empty cells and p = `nParameters`, the number of estimated parameters.

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 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.

Parameter

`cutpoints` – a `double` array which contains the cutpoints

setRange

```
public void setRange(double lower, double upper) throws
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) throws ChiSquaredTest.NotCDFException
```

Description

Adds a new observation to the test.

Parameter

`x` – a `double`, the new observation to be added to the test. The frequency of this observation is assumed to be 1.0.

update

```
public void update(double[] x) throws ChiSquaredTest.NotCDFException
```

Description

Adds new observations to the test.

Parameter

`x` – a `double` array which contains the new observations to be added to the test. The frequencies of these observations are assumed to be 1.0.

update

```
public void update(double x, double freq) throws 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

update

```
public void update(double[] x, double[] freq) throws
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

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 0.44124295720552553
The Degrees of freedom are 5.0
```

ChiSquaredTest.NotCDFException class

`static public class com.imsl.stat.ChiSquaredTest.NotCDFException extends com.imsl.IMSLRuntimeException`

The function is not a Cumulative Distribution Function (CDF).

Constructor

ChiSquaredTest.NotCDFException

`public ChiSquaredTest.NotCDFException(String key, Object[] arguments)`

Description

Constructs a `NotCDFException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ChiSquaredTest.NoObservationsException class

`static public class com.imsl.stat.ChiSquaredTest.NoObservationsException extends com.imsl.IMSLRuntimeException`

There are no observations.

Constructor

ChiSquaredTest.NoObservationsException

`public ChiSquaredTest.NoObservationsException(String key, Object[] arguments)`

Description

Constructs a `NoObservationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ChiSquaredTest.DidNotConvergeException class

`static public class com.imsl.stat.ChiSquaredTest.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge

Constructors

ChiSquaredTest.DidNotConvergeException

`public ChiSquaredTest.DidNotConvergeException(String message)`

Description

Constructs a `DidNotConvergeException` object.

Parameter

`message` – a `String` containing the error message

ChiSquaredTest.DidNotConvergeException

`public ChiSquaredTest.DidNotConvergeException(String key, Object[] arguments)`

Description

Constructs a `DidNotConvergeException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

NormalityTest class

`public class com.imsl.stat.NormalityTest implements Serializable, Cloneable`

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`.

Constructor

NormalityTest

`public NormalityTest(double[] x)`

Description

Constructor for `NormalityTest`.

Parameter

`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

```
final public double ChiSquaredTest(int n) throws
NormalityTest.NoVariationInputException, InverseCdf.DidNotConvergeException
```

Description

Performs the chi-squared goodness-of-fit test.

Parameter

`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.

Exceptions

`NoVariationInputException` is thrown if there is no variation in the input data.

`DidNotConvergeException` is thrown if the iteration did not converge.

LillieforsTest

```
final public double LillieforsTest() throws
NormalityTest.NoVariationInputException, 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.

Exceptions

`NoVariationInputException` is thrown if there is no variation in the input data.

`DidNotConvergeException` is thrown if the iteration did not converge.

ShapiroWilkWTest

```
final public double ShapiroWilkWTest() throws
NormalityTest.NoVariationInputException, InverseCdf.DidNotConvergeException
```

Description

Performs the Shapiro-Wilk W test.

Returns

A `double` scalar containing the p-value for the Shapiro-Wilk W test.

Exceptions

`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.

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.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
```

NormalityTest.NoVariationInputException class

`static public class com.imsl.stat.NormalityTest.NoVariationInputException extends com.imsl.IMSLException`

There is no variation in the input data.

Constructors

NormalityTest.NoVariationInputException

`public NormalityTest.NoVariationInputException(String message)`

Description

Constructs a `NoVariationInputException` object.

Parameter

`message` – a `String` containing the error message

NormalityTest.NoVariationInputException

`public NormalityTest.NoVariationInputException(String key, Object[] arguments)`

Description

Constructs a `NoVariationInputException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

KolmogorovOneSample class

`public class com.imsl.stat.KolmogorovOneSample implements Serializable`

The class `KolmogorovOneSample` performs a Kolmogorov-Smirnov goodness-of-fit test in one sample.

The hypotheses tested follow:

$$\begin{array}{ll} H_0: F(x) = F^*(x) & H_1: F(x) \neq F^*(x) \\ H_0: F(x) \geq F^*(x) & H_1: F(x) < F^*(x) \\ H_0: F(x) \leq F^*(x) & H_1: F(x) > F^*(x) \end{array}$$

where F is the cumulative distribution function (CDF) of the random variable, and the theoretical cdf, F^*, is specified via the user-supplied function cdf. Let n be the number of observations minus the number of missing observations. The test statistics for both one-sided alternatives D_n^+ and D_n^- and the two-sided D_n alternative are computed as well as an asymptotic z-score and p-values associated with the one-sided and two-sided hypotheses. For $n > 80$, asymptotic p-values are used (see Gibbons 1971). For $n \leq 80$, exact one-sided p-values are computed according to a method given by Conover (1980, page 350). An approximate two-sided test p-value is obtained as twice the one-sided p-value. The approximation is very close for one-sided p-values less than 0.10 and becomes very bad as the one-sided p-values get larger.

The theoretical CDF is assumed to be continuous. If the CDF is not continuous, the statistics D_n^* will not be computed correctly.

Estimation of parameters in the theoretical CDF from the sample data will tend to make the p-values associated with the test statistics too liberal. The empirical CDF will tend to be closer to the theoretical CDF than it should be.

No attempt is made to check that all points in the sample are in the support of the theoretical CDF. If all sample points are not in the support of the CDF, the null hypothesis must be rejected.

Constructor

KolmogorovOneSample

```
public KolmogorovOneSample(CdfFunction cdf, double[] x)
```

Description

Constructs a one sample Kolmogorov-Smirnov goodness-of-fit test.

Parameters

`cdf` – is the cdf function, $F(x)$. If must be non-decreasing and its value must be in [0, 1].

`x` – is a `double` array containing the observations.

Methods

getMaximumDifference

```
public double getMaximumDifference()
```

Description

Returns D^+, the maximum difference between the theoretical and empirical CDF's.

Returns

The value D^+.

getMinimumDifference

```
public double getMinimumDifference()
```

Description

Returns D^-, the minimum difference between the theoretical and empirical CDF's.

Returns

The value D^-.

getNumberMissing

```
public int getNumberMissing()
```

Description

Returns the number of missing values in the data.

Returns

The number of missing values.

getNumberOfTies

```
public int getNumberOfTies()
```

Description

Returns the number of ties in the data.

Returns

the number of ties in the data

getOneSidedPValue

```
public double getOneSidedPValue()
```

Description

Probability of the statistic exceeding D under the null hypothesis of equality and against the one-sided alternative. An exact probability is computed if the number of observation is less than or equal to 80, otherwise an approximate probability is computed.

Returns

the one-sided probability.

getTestStatistic

```
public double getTestStatistic()
```

Description

Returns $D = \max(D^+, D^-)$.

Returns

The value D.

getTwoSidedPValue

```
public double getTwoSidedPValue()
```

Description

Probability of the statistic exceeding D under the null hypothesis of equality and against the two-sided alternative. This probability is twice the probability, p_1, reported by `getOneSidedPValue`, (or 1.0 if $p_1 \geq 1/2$). This approximation is nearly exact when $p_1 < 0.1$.

Returns

the two-sided probability.

getZ

```
public double getZ()
```

Description

Returns the normalized D statistic without the continuity correction applied.

Returns

the value Z

Example

In this example, a random sample of size 100 is generated using class `Random` for the uniform (0, 1) distribution. We want to test the null hypothesis that the cdf is the standard normal distribution with a mean of 0.5 and a variance equal to the uniform (0, 1) variance (1/12).

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

public class KolmogorovOneSampleEx1 {

    static public void main(String arg[]) {
        CdfFunction cdf = new CdfFunction() {
            public double cdf(double x) {
                double mean = 0.5;
                double std = 0.2886751;
                double z = (x - mean) / std;
                return Cdf.normal(z);
            }
        };

        double x[] = new double[100];
        Random random = new Random(123457);
        random.setMultiplier(16807);
        for (int i = 0; i < x.length; i++) {
            x[i] = random.nextDouble();
        }

        KolmogorovOneSample kos = new KolmogorovOneSample(cdf, x);
        System.out.println("D  = " + kos.getTestStatistic());
        System.out.println("D+ = " + kos.getMaximumDifference());
        System.out.println("D- = " + kos.getMinimumDifference());
        System.out.println("Z   = " + kos.getZ());
        System.out.println("Prob greater D one sided = "
                + kos.getOneSidedPValue());
        System.out.println("Prob greater D two sided = "
                + kos.getTwoSidedPValue());
        System.out.println("N missing = " + kos.getNumberMissing());
    }
}
```

Output

```
D  = 0.12191406736055721
D+ = 0.12191406736055721
D- = 0.08694298500243408
Z   = 1.219140673605572
Prob greater D one sided = 0.05116968745900741
Prob greater D two sided = 0.10233937491801481
N missing = 0
```

KolmogorovTwoSample class

`public class com.imsl.stat.KolmogorovTwoSample implements Serializable`

Performs a Kolmogorov-Smirnov two-sample test.

Class `KolmogorovTwoSample` computes Kolmogorov-Smirnov two-sample test statistics for testing that two continuous cumulative distribution functions (CDF's) are identical based upon two random samples. One- or two-sided alternatives are allowed. Exact p-values are computed for the two-sided test when $nm \leq 104$, where n is the number of non-missing X observations and m the number of non-missing Y observation.

Let $F_n(x)$ denote the empirical CDF in the X sample, let $G_m(y)$ denote the empirical CDF in the Y sample and let the corresponding population distribution functions be denoted by $F(x)$ and $G(y)$, respectively. Then, the hypotheses tested by `KolmogorovTwoSample` are as follows:

$$\begin{array}{ll} H_0: \ F(x) = G(x) & H_1: \ F(x) \neq G(x) \\ H_0: \ F(x) \geq G(x) & H_1: \ F(x) < G(x) \\ H_0: \ F(x) \leq G(x) & H_1: \ F(x) > G(x) \end{array}$$

The test statistics are given as follows:

$$\begin{array}{ll} D_{mn} & = \max(D_{mn}^+, D_{mn}^-) \\ D_{mn}^+ & = \max_x(F_n(x) - G_m(x)) \\ D_{mn}^- & = \max_x(G_m(x) - F_n(x)) \end{array}$$

Asymptotically, the distribution of the statistic

$$Z = D_{mn}\sqrt{\frac{mn}{m+n}}$$

converges to a distribution given by Smirnov (1939).

Exact probabilities for the two-sided test are computed when $nm \leq 104$, according to an algorithm given by Kim and Jennrich (1973). When $nm > 104$, the very good approximations given by Kim and Jennrich are used to obtain the two-sided p-values. The one-sided probability is taken as one half the two-sided probability. This is a very good approximation when the p-value is small (say, less than 0.10) and not very good for large p-values.

Constructor

KolmogorovTwoSample

```
public KolmogorovTwoSample(double[] x, double[] y)
```

Description

Constructs a two sample Kolmogorov-Smirnov goodness-of-fit test.

Parameters

`x` – is an array containing the observations from the first sample.

`y` – is an array containing the observations from the second sample.

Methods

getMaximumDifference

```
public double getMaximumDifference()
```

Description

Returns D^+, the maximum difference between the theoretical and empirical CDF's.

Returns

The value D^+.

getMinimumDifference

```
public double getMinimumDifference()
```

Description

Returns D^-, the minimum difference between the theoretical and empirical CDF's.

Returns

The value D^-.

getNumberMissingX

```
public int getNumberMissingX()
```

Description

Returns the number of missing values in the `x` sample.

Returns

The number of missing values in `x`.

getNumberMissingY

```
public int getNumberMissingY()
```

Description

Returns the number of missing values in the `y` sample.

Returns

The number of missing values in `y`.

getOneSidedPValue

```
public double getOneSidedPValue()
```

Description

Probability of the statistic exceeding D under the null hypothesis of equality and against the one-sided alternative. An exact probability is computed if the number of observation is less than or equal to 80, otherwise an approximate probability is computed.

Returns

the one-sided probability.

getTestStatistic

```
public double getTestStatistic()
```

Description

Returns $D = \max(D^+, D^-)$.

Returns

The value D.

getTwoSidedPValue

```
public double getTwoSidedPValue()
```

Description

Probability of the statistic exceeding D under the null hypothesis of equality and against the two-sided alternative. This probability is twice the probability, p_1, reported by `getOneSidedPValue`, (or 1.0 if $p_1 \geq 1/2$. This approximation is nearly exact when $p_1 < 0.1$.

Returns

the two-sided probability.

getZ

```
public double getZ()
```

Description

Returns the normalized D statistic without the continuity correction applied.

Returns

the value Z

Example

The following example illustrates the class `KolmogorovTwoSample` routine with two randomly generated samples from a uniform(0,1) distribution. Since the two theoretical distributions are identical, we would not expect to reject the null hypothesis.

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

public class KolmogorovTwoSampleEx1 {

    static public void main(String arg[]) {
        double x[] = new double[100];
        double y[] = new double[60];
        Random random = new Random(123457);
        random.setMultiplier(16807);
        for (int i = 0; i < x.length; i++) {
            x[i] = random.nextFloat();
        }
```

```
        for (int i = 0; i < y.length; i++) {
            y[i] = random.nextFloat();
        }

        KolmogorovTwoSample k2s = new KolmogorovTwoSample(x, y);
        System.out.println("D  = " + k2s.getTestStatistic());
        System.out.println("D+ = " + k2s.getMaximumDifference());
        System.out.println("D- = " + k2s.getMinimumDifference());
        System.out.println("Z   = " + k2s.getZ());
        System.out.println("Prob greater D one sided = "
                + k2s.getOneSidedPValue());
        System.out.println("Prob greater D two sided = "
                + k2s.getTwoSidedPValue());
        System.out.println("Missing X = " + k2s.getNumberMissingX());
        System.out.println("Missing Y = " + k2s.getNumberMissingY());
    }
}
```

Output

```
D  = 0.18
D+ = 0.18
D- = 0.010000000000000009
Z   = 1.1022703842524302
Prob greater D one sided = 0.07201060734868497
Prob greater D two sided = 0.14402121469736995
Missing X = 0
Missing Y = 0
```

Chapter 19: Time Series and Forecasting

Types

class AutoCorrelation 880
class ARAutoUnivariate 890
class ARSeasonalFit 915
class ARMA 928
class ARMAEstimateMissing 954
class ARMAMaxLikelihood 964
class ARMAOutlierIdentification 981
class AutoARIMA 1000
class CrossCorrelation 1024
class Difference 1035
class GARCH 1040
class KalmanFilter 1050
class MultiCrossCorrelation 1062
class LackOfFit 1076
class HoltWintersExponentialSmoothing 1079
class TimeSeries 1088
class TimeSeriesOperations 1096
class VectorAutoregression 1108

Usage Notes

The classes in this chapter allow users to filter and analyze time series data using autoregressive, ARIMA, GARCH and state-space models. Some are designed for automatic model selection, estimation and forecasting using algorithms based upon minimizing AIC (Akaike's Information Criterion).

General Methodology

Model Identification and Data Filtering

All classes in this chapter for time series analysis assume the data are stationary and collected at equally

spaced times with no missing observations in the series. In order to prepare a series for analysis, any missing values must be replaced with estimates. The class `ARMAEstimateMissing` can be used to automatically estimate missing values using one of four, user-selected methods, provided the largest time gap with missing values has no more than 4 missing values.

In addition to estimating missing values, if a series is nonstationary or contains seasonal variation, its values should be filtered before fitting a model and preparing forecasts. Class `ARSeasonalFit` evaluates seasonal adjustments prior to modeling the series.Users specify a range of adjustments.`ARSeasonalFit` evaluates each adjustment to identify the one that minimizes the AIC. If a user already knows the series needs to be filtered using differences between consecutive values, the class `Difference` can be used to filter a series into an equivalent series of differences.

There are several classes available for transforming a time series into its autocorrelation matrix: `AutoCorrelation`, `CrossCorrelation` and `MultiCrossCorrelation`.Class `AutoCorrelation` computes the autocorrelation and partial autocorrelation matrices from a time series or its filtered values. Box and Jenkins (1976) describe how to identify the structure of an ARIMA model using the autocorrelation and partial autocorrelation matrices.

Classes `CrossCorrelation` and `MultiCrossCorrelation` are used to calculate the cross-correlation matrix for two univariate or multivariate time series, respectively.

Model Estimation and Forecasting

There are several classes useful for modeling stationary, equally spaced time series - `ARMA`, `ARMAMaxLikelihood`, `GARCH`, and `KalmanFilter`. Class `ARMA` can be used to fit autoregressive, moving-average and ARIMA (autoregressive, integrated moving average) models.Rather than passing the original series to this class, users are required to construct this class using the autocorrlelation matrix of the series.Class `AutoCorrelation` is useful for computing this matrix prior to constructing the `ARMA` class.

The other classes require users to construct the class using the original series or its filtered values.Class `ARMAMaxLikelihood` estimates the maximum likelihood estimates of the ARMA parameters and prepares forecasts from these values. Class `ARAutoUnivariate` selects the best autoregressive model for an equally spaced, stationary time series using AIC.Class `GARCH` estimates the parameters of a GARCH model and the `KalmanFilter` class estimates the parameters for a state-space model using Kalman filtering and calculates one-step ahead forecasts

Classes `ARMA`, `ARMAMaxLikelihood` and `ARAutoUnivariate` are the only classes with methods for obtaining forecasts from an ARIMA model. Class `KalmanFilter` can be used to obtain one-step ahead forecasts for a state-space model.

ARIMA Model (Autoregressive Integrated Moving Average)

A small, yet comprehensive, class of stationary time-series models consists of the nonseasonal ARMA processes defined by

$$\phi(B)(W_t - \mu) = \theta(B)A_t, \quad t \in Z$$

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 θ_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, $d > 0$.

There are two classes for estimating the parameters in an ARMA model and preparing forecasts: `ARMA` and `ARMAMaxLikelihood`.Class`ARMA` estimates model parameters using either method of moments or least squares. The method of momentsis selected bydefault or by setting property `Method` to `METHOD_Of_MOMENTS`. Method of momentestimatesare goodinitial valuesfor obtainingthe least-squaresestimates by setting property `Method` to `LEAST_SQUARES` . 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`ARMAMaxLikelihood` estimates model parameters using maximum likelihood. Users can tailor the maximum likelihood algorithm by setting tolerances and providing initial estimates for the parameters. By default, `ARMAMaxLikelihood` uses method of moment estimates to initialize the parameters. However, this can be overridden with the `setAR` method in `ARMAMaxLikelihood`.

Exponential Smoothing Methods

Exponential smoothing approximates the value of a time series at time t with a weighted average of previous values, with weights defined in such a way that they decay exponentially over time. The weights can be determined by a smoothing parameter α and the relation,

$$y_t = \alpha y_{t-1} + \hat{y}_t$$

$$\Rightarrow$$

$$y_t = \sum_{j=0}^{t-1} \alpha^{t-j} (1-\alpha)^j y_j = \sum_{j=0}^{t-1} w_j y_j$$

The parameter α is on the interval (0,1) and controls the rate of decay. For values close to 1, the effect decays rapidly, whereas for values close to 0, the influence of a past value persists for some time.

Exponential smoothing as a forecasting procedure is widely used and largely effective for short term, mean level forecasting. With the addition of a term for linear trend and terms or factors for seasonal patterns, exponential smoothing is an intuitive procedure for many series that arise in business applications. Class `HoltWintersExponentialSmoothing` performs the Holt-Winters method, otherwise known as *triple exponential smoothing*, and allows for either an additive or a multiplicative seasonal effect.

AutoCorrelation class

```
public class com.imsl.stat.AutoCorrelation implements Serializable, Cloneable
```

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 μ of the time series $\{X_t\}$ where

$$\hat{\mu} = \begin{cases} \mu & \text{for } \mu \text{ known} \\ pa \ \frac{1}{n}\sum_{t=1}^{n} X_t & \text{for } \mu \text{ unknown} \end{cases}$$

The autocovariance function $\sigma(k)$ is estimated by

$$\hat{\sigma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu})(X_{t+k} - \hat{\mu}), \qquad \text{k=0,1,...,K}$$

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

$$\text{var}\{\hat{\rho}(k)\} = \frac{1}{n}\sum_{i=-\infty}^{\infty}\left[\rho^2(i) + \rho(i-k)\rho(i+k) - 4\rho(i)\rho(k)\rho(i-k) + 2\rho^2(i)\rho^2(k)\right]$$

where $\hat{\rho}(k)$ assumes μ is unknown. For computational purposes, the autocorrelations $\rho(k)$ are replaced by their estimates $\hat{\rho}(k)$ for $|k| \le K$, 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 μ 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 ϕ_{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

$$\hat{\rho}(j) = \hat{\phi}_{k1}\hat{\rho}(j-1) + \hat{\phi}_{k2}\hat{\rho}(j-2) + \cdots + \hat{\phi}_{kk}\hat{\rho}(j-k), \qquad \text{j = 1,2,}\ldots\text{,k}$$

a recursive relationship for k=1, ..., K was developed by Durbin (1960). The equations are given by

$$\hat{\phi}_{kk} = \begin{cases} \hat{\rho}(1) & \text{for k} = 1 \\ \dfrac{\hat{\rho}(k) - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(k-j)}{1 - \sum_{j=1}^{k-1} \hat{\phi}_{k-1,j}\hat{\rho}(j)} & \text{for k} = 2, \ldots, \text{K} \end{cases}$$

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

$$\text{var}\{\hat{\phi}_{kk}\} \simeq \frac{1}{n} \quad \text{k} \ \ge \ \text{p} + 1$$

See Box and Jenkins (1976, pages 82-84) for more information concerning the partial autocorrelation function.

Fields

BARTLETTS_FORMULA

```
static final public int BARTLETTS_FORMULA
```

Indicates standard error computation using Bartlett's formula.

MORANS_FORMULA

```
static final public 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, ..., maximum_lag.

getAutoCovariances

```
public double[] getAutoCovariances() throws
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 $k = 1, ...,$ maximum_lag.

Exception

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

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the number of java.lang.Thread instances used for parallel processing.

Returns

an int containing the number of java.lang.Thread instances used for parallel processing.

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.

Parameter

`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`.

Parameter

`mean` – a `double` containing the estimate mean of the time series `x`.

setNumberOfThreads

`public void setNumberOfThreads(int numberOfThreads)`

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing.

Default: `numberOfThreads` = 1.

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 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(AutoCorrelation.BARTLETTS_FORMULA));
        System.out.println();
        new PrintMatrix("Standard Error using Moran are:  ").print(
                ac.getStandardErrors(AutoCorrelation.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(AutoCorrelation.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
```

AutoCorrelation.NonPosVariancesException class

`static public class com.imsl.stat.AutoCorrelation.NonPosVariancesException extends com.imsl.IMSLException`

The problem is ill-conditioned.

Constructors

AutoCorrelation.NonPosVariancesException

`public AutoCorrelation.NonPosVariancesException(String message)`

Description

Constructs an `NonPosVariancesException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

AutoCorrelation.NonPosVariancesException

`public AutoCorrelation.NonPosVariancesException(String key, 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

ARAutoUnivariate class

`public class com.imsl.stat.ARAutoUnivariate implements Serializable`

Automatically determines the best autoregressive time series model using Akaike's Information Criterion.

`ARAutoUnivariate` automatically selects the order of the AR model that best fits the data and then computes the AR coefficients. The algorithm used in `ARAutoUnivariate` is derived from the work of Akaike, H., et. al (1979) and Kitagawa and Akaike (1978). This code was adapted from the UNIMAR procedure published as part of the TIMSAC-78 Library.

The best fit AR model is determined by successively fitting AR models with $0, 1, 2, \ldots, \mathrm{maxlag}$ autoregressive coefficients. For each model, Akaike's Information Criterion (AIC) is calculated based on the formula

$$\mathrm{AIC} = -2\ln(likelihood) + 2p$$

Class `ARAutoUnivariate` uses the approximation to this formula developed by Ozaki and Oda (1979),

$$\mathrm{AIC} = (n - \mathrm{maxlag})\ln(\hat{\sigma}^2) + 2(p+1) + (n - \mathrm{maxlag})(\ln(2\pi) + 1)$$

where $\hat{\sigma}^2$ is an estimate of the residual variance of the series, commonly known in time series analysis as the innovation variance and n is the number of observations in the time series `z`, n=`z.length`. By dropping the constant

$$(n - \mathrm{maxlag})(\ln(2\pi) + 1),$$

the calculation is simplified to

$$\mathrm{AIC} = (n - \mathrm{maxlag})\ln(\hat{\sigma}^2) + 2(p+1),$$

The best fit model is the model with minimum AIC. If the number of parameters in this model selected by `ARAutoUnivariate` is equal to the highest order autoregressive model fitted, i.e., p= `maxlag`, then a model with smaller AIC might exist for larger values of `maxlag`. In this case, increasing `maxlag` to explore AR models with additional autoregressive parameters might be warranted.

Method `setEstimationMethod` can be used to specify the method used to calculate the AR coeficients. If `setEstimationMethod` is set to `METHOD_OF_MOMENTS`, estimates of the autoregressive coefficients for the model with minimum AIC are calculated using method of moments as described in the `ARMA`

class. If LEAST_SQUARES is specified, the coefficients are determined by the method of least squares applied in the form described by Kitagawa and Akaike (1978). If MAX_LIKELIHOOD is specified, the coefficients are estimated using maximum likelihood as described in the ARMAMaxLikelihood class.

The Java Logging API can be used to trace the execution of ARAutoUnivariate. The name of this logger is com.imsl.stat.ARAutoUnivariate. 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 compute.
FINER	All of the messages in FINE, a message entering and exiting the compute method, plus the final computations.
FINEST	All of the messages in FINER, plus the intermediate computations.

Fields

LEAST_SQUARES

```
static final public int LEAST_SQUARES
```

Indicates that least-squares should be used for estimating the coefficients in the time series.

MAX_LIKELIHOOD

```
static final public int MAX_LIKELIHOOD
```

Indicates that maximum likelihood should be used for estimating the coefficients in the time series.

METHOD_OF_MOMENTS

```
static final public int METHOD_OF_MOMENTS
```

Indicates the method of moments should be used for estimating the coefficients in the time series.

Constructor

ARAutoUnivariate

```
public ARAutoUnivariate(int maxlag, double[] z)
```

Description

ARAutoUnivariate constructor.

Parameters

maxlag – an int scalar specifying the maximum number of autoregressive lags to evaluate

z – a double array containing the time series

Methods

compute

```
public void compute() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Determines the autoregressive model with the minimum AIC by fitting autoregressive models from 0 to `maxlag` lags using the method of moments or an estimation method specified by the user through `setEstimationMethod`.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

forecast

```
public double[][] forecast(int nForecast) throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns forecasts and associated confidence interval offsets.

Parameter

`nForecast` – an input `int` representing the number of requested forecasts

Returns

a `double` matrix of dimension `nForecast` by `backwardOrigin + 1` containing the forecasts. The forecasts are for lead times $l = 1, 2, \ldots, \text{nForecast}$ at origins `z.length-backwardOrigin-1+j` where $j = 1, \ldots, \text{backwardOrigin} + 1$.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getAIC

```
public double getAIC() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final estimate for Akaike's Information Criterion (AIC) at the optimum.

Returns

a `double` scalar value which is an approximation to $\text{AIC} = -2\ln(L) + 2p$, where L is the value of the maximum likelihood function evaluated at the parameter estimates. The approximation uses the calculation

$$\text{AIC} \approx (\text{n} - \text{maxlag})\ln(\hat{\sigma}^2) + 2(\text{p}+1) + (\text{n} - \text{maxlag})(\ln(2\pi)+1),$$

where $\hat{\sigma}^2$ is an estimate of the residual variance of the series, commonly known in time series analysis as the innovation variance, and `n` is the number of observations in the time series (`n=z.length`).

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getAR

```
public double[] getAR() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final auto regressive parameter estimates at the optimum AIC using the estimation method specified in `setEstimationMethod` .

Returns

a `double` array containing the estimates for the autoregressive parameters.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getBackwardOrigin

```
public int getBackwardOrigin()
```

Description

Returns the maximum backward origin.

Returns

an `int` scalar specifying the maximum backward origin.

getConfidence

```
public double getConfidence()
```

Description

Returns the confidence level for calculating confidence limit deviations returned from `getDeviations`.

Returns

a `double` scalar value representing the confidence level used in computing forecast confidence intervals.

getConstant

```
public double getConstant() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the estimate for the constant parameter in the ARMA series.

Returns

a `double` scalar equal to the estimate for the constant parameter in the ARMA series.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

ARMAMaxLikelihood.InitialMAException is thrown if the initial values provided for the moving average terms using setMA are noninvertible. In this case, ARMAMaxLikelihood terminates and does not compute the time series estimates.

getConvergenceTolerance

```
public double getConvergenceTolerance()
```

Description

Returns the tolerance level used to determine convergence of the nonlinearleast-squares and maximum likelihood algorithms.

Returns

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.

getDeviations

```
public double[] getDeviations()
```

Description

Returns the deviations used for calculating the forecast confidence limits.

Returns

a double array of length backwardOrigin+nForecast containing the deviations for calculating forecast confidence intervals. The confidence level is specified in confidence. By default, confidence = 0.95.

getEstimationMethod

```
public int getEstimationMethod()
```

Description

Returns the estimation method used for estimating the autoregressive coefficients.

Returns

an int equal to 0, 1, or 2, representing the autoregressive coefficient estimation method, which implies METHOD_OF_MOMENTS , LEAST_SQUARES , or MAX_LIKELIHOOD respectively.

getForecast

```
public double[] getForecast(int nForecast) throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns forecasts

Parameter

`nForecast` – an input `int` representing the number of requested forecasts beyond the last value in the series.

Returns

a `double` array containing the `nForecast+backwardOrigin` forecasts. The first `backwardOrigin` forecasts are one-step ahead forecasts for the last `backwardOrigin` values in the series. The next `nForecast` values in the returned series are forecasts for the next values beyond the series.

Exceptions

`NonStationary` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`NonInvertible` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getInnovationVariance

```
public double getInnovationVariance() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final estimate for the innovation variance.

Returns

a `double` scalar value equal to the estimate for the innovation variance.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getLikelihood

```
public double getLikelihood() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final estimate for $L = e^{-(\text{AIC}-2p)/2}$, where p is the AR order, AIC is Akaike's Information Criterion, and L is the likelihood function evaluated for the optimum autoregressive model. If `MAX_LIKELIHOOD` is set in the `setEstimationMethod`, the exact likelihood evaluated for the optimum autoregressive model will be returned. Otherwise it is calculated using the approximation to the AIC.

Returns

a `double` scalar equal to the maximum of the likelihood function, $L = e^{-(\text{AIC}-2p)/2}$.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getLogger

```
public Logger getLogger()
```

Description

Returns the logger object.

Returns

the logger object, if present, or null.

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the value currently being used as the maximum number of iterations allowed in the nonlinear equation solver used in both the method of moments and least-squares algorithms.

Returns

an `int` containing the maximum number of iterations allowed.

getMaxlag

```
public int getMaxlag()
```

Description

Returns the current value used to represent the maximum number of autoregressive lags to achieve the minimum AIC.

Returns

an `int` containing the maximum number of autoregressive lags evaluated.

getMean

```
public double getMean() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the mean used to center the time series `z`.

Returns

a `double` containing the value used to center the series before searching for the optimum number of AR lags. This is equal to the mean of the time series `z` unless the mean was set in the `setMean` method.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getOrder

```
public int getOrder() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the order of the AR model selected with the minimum AIC.

Returns

an `int` containing the optimum AR order selected by `ARAutoUnivariate`'s minimum AIC selection.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

ARMAMaxLikelihood.InitialMAException is thrown if the initial values provided for the moving average terms using setMA are noninvertible. In this case, ARMAMaxLikelihood terminates and does not compute the time series estimates.

getResiduals

```
public double[] getResiduals()
```

Description

Returns the current values of the vector of residuals.

Returns

a double array of length backwardOrigin containing the residuals for the last backwardOrigin values in the time series. This array will be null unless compute and either the forecast or getForecast methods are called previously.

getTimeSeries

```
public double[] getTimeSeries()
```

Description

Returns the time series used for estimating the minimum AIC and the autoregressive coefficients.

Returns

a double array containing the time series values set in the constructor.

getTimsacAR

```
public double[] getTimsacAR() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final auto regressive parameter estimates at the optimum AIC estimated by the original TIMSAC routine (UNIMAR). These estimates vary depending upon the value of maxlag since its value changes the length of the series used to estimate these parameters. Use the getAR method to obtain the best estimates from the estimation method specified by setEstimationMethod calculated using the maximum number of available observations.

Returns

a double array of length p containing the estimates for the autoregressive parameters.

Exceptions

ARMA.MatrixSingularException is thrown if the input matrix is singular.

ARMA.TooManyCallsException is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getTimsacConstant

```
public double getTimsacConstant() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the estimate for the constant parameter in the ARMA series.

Returns

a `double` scalar equal to the estimate for the constant parameter in the ARMA series.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getTimsacVariance

```
public double getTimsacVariance() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final estimate for the innovation variance calculated by the TIMSAC automatic AR modeling routine (UNIMAR). This variance depends upon the value of `maxlag` since the series length in this computation depends on its value. Use `getInnovationVariance` to return the innovation variance calculated using the maximum series length.

Returns

a `double` scalar value equal to the estimate for the innovation variance.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

setBackwardOrigin

`public void setBackwardOrigin(int backwardOrigin)`

Description

Sets the maximum backward origin used in calculating the forecasts.

Parameter

`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 - p`, where `p` is the optimum number of AR coeficients determined by `ARAutoUnivariate`. The forecasted values returned will be `z.length - backwardOrigin` through `z.length`. Default: `backwardOrigin = 0`.

setConfidence

`public void setConfidence(double confidence)`

Description

Sets the confidence level for calculating confidence limit deviations returned from `getDeviations`.

Parameter

`confidence` – a `double` scalar specifying the confidence level used in computing forecast confidence intervals. 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 and maximum likelihood algorithms.

Parameter

`convergenceTolerance` – a `double` scalar containing the tolerance level used to determine convergence of the nonlinear least-squares and maximum likelihood algorithms. `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. By default, convergenceTolerance = 10^{-10}.

setEstimationMethod

`public void setEstimationMethod(int method)`

Description

Sets the estimation method used for estimating the final estimates for the autoregressive coefficients.

Parameter

`method` – an `int` scalar specifying the estimation method. Specify `METHOD_OF_MOMENTS` for the method of moments, `LEAST_SQUARES` for the least squares method, or `MAX_LIKELIHOOD` for the maximum likelihood estimates. Default: `method` = `LEAST_SQUARES`.

setMaxIterations

```
public void setMaxIterations(int iterations)
```

Description

Sets the maximum number of iterations used for estimating the autoregressive coefficients.

Parameter

`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: `iterations` = 300.

Exception

`IllegalArgumentException` is thrown if `iterations` is less than or equal to 0.

setMean

```
public void setMean(double mean)
```

Description

Sets the estimate of the mean used for centering the time series `z` .

Parameter

`mean` – a `double` containing the estimate of the mean for the time series `z`. By default, the time series `z` is centered about its sample mean.

Example 1: ARAutoUnivariate

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. In this example, `ARAutoUnivariate` found the minimum AIC fit is an autoregressive model with 3 lags:

$$\tilde{W}_t = \phi_1 \tilde{W}_{t-1} + \phi_2 \tilde{W}_{t-2} + \phi_3 \tilde{W}_{t-3} + a_t,$$

where

$$\tilde{W}_t := W_t - \mu$$

μ is the sample mean of the time series $\{W_t\}$. Defining the overall constant θ_0 by $\theta_0 := \mu(1 - \sum_{i=1}^{3} \phi_i)$, we obtain the following equivalent representation:

$$W_t = \theta_0 + \phi_1 W_{t-1} + \phi_2 W_{t-2} + \phi_3 W_{t-3} + a_t.$$

The example computes estimates for $\theta_0, \phi_1, \phi_2, \phi_3$ for each of the three parameter estimation methods available.

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

public class ARAutoUnivariateEx1 {

    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
        };

        ARAutoUnivariate autoAR = new ARAutoUnivariate(20, z);
        System.out.println("");
        System.out.println("    Method of Moments ");
        System.out.println("");

        autoAR.setEstimationMethod(ARAutoUnivariate.METHOD_OF_MOMENTS);
        autoAR.compute();

        System.out.println("Order Selected: " + autoAR.getOrder());
        System.out.println("AIC = " + autoAR.getAIC() + "       Variance = "
                + autoAR.getInnovationVariance());
        System.out.println("Constant = " + autoAR.getConstant());
        System.out.println("");
        new PrintMatrix("AR estimates are:  ").print(autoAR.getAR());

        /* try another method */
        System.out.println("");
        System.out.println("");
        System.out.println("    Least Squares ");
        System.out.println("");
        autoAR.setEstimationMethod(ARAutoUnivariate.LEAST_SQUARES);
        autoAR.compute();

        System.out.println("Order Selected: " + autoAR.getOrder());
        System.out.println("AIC = " + autoAR.getAIC() + "       Variance = "
                + autoAR.getInnovationVariance());
        System.out.println("Constant = " + autoAR.getConstant());
        System.out.println();
        new PrintMatrix("AR estimates are:  ").print(autoAR.getAR());

        System.out.println("");
        System.out.println("");
        System.out.println("    Maximum Likelihood ");
        System.out.println("");
        autoAR.setEstimationMethod(ARAutoUnivariate.MAX_LIKELIHOOD);
        autoAR.compute();
```

```
        System.out.println("Order Selected: " + autoAR.getOrder());
        System.out.println("AIC = " + autoAR.getAIC() + "       Variance = "
                + autoAR.getInnovationVariance());
        System.out.println("Constant = " + autoAR.getConstant());
        System.out.println();
        new PrintMatrix("AR estimates are:  ").print(autoAR.getAR());
    }
}
```

Output

```
    Method of Moments

Order Selected: 3
AIC = 405.9812419650225       Variance = 287.2699077443474
Constant = 13.70978940109534

AR estimates are:
     0
0   1.368
1  -0.738
2   0.078



    Least Squares

Order Selected: 3
AIC = 405.9812419650225       Variance = 221.99519660517407
Constant = 11.610494624981452

AR estimates are:
     0
0   1.55
1  -1.004
2   0.205



    Maximum Likelihood

Order Selected: 3
AIC = 405.9812419650225       Variance = 218.84837588472615
Constant = 11.417850982501161

AR estimates are:
     0
0   1.551
1  -0.999
2   0.204
```

Example 2 (Logging and Forecasting): ARAutoUnivariate

Using the Canadian Lynx data included in TIMSAC-78, `ARAutoUnivariate` is used to find the minimum AIC autoregressive model using a maximum number of lags of `maxlag`=20.

This example compares the three different methods for estimating the autoregressive coefficients, and it illustrates the relationship between these estimates and those calculated within the TIMSAC UNIMAR code. As illustrated, the UNIMAR code estimates the coefficients and innovation variance using only the last N-maxlag values in the time series. The other estimation methods use all *N-k* values, where *k* is the number of lags with minimum AIC selected by `ARAutoUnivariate`.

This example also illustrates how to generate forecasts for the observed series values and beyond by setting the backward orgin for the forecasts.

```
import java.text.*;
import com.imsl.stat.*;
import com.imsl.math.*;
import java.util.logging.*;

public class ARAutoUnivariateEx2 {

    public static void main(String args[]) throws Exception {
        /* THE CANDIAN LYNX DATA AS USED IN TIMSAC 1821-1934 */
        double[] y = {
            0.24300e01, 0.25060e01, 0.27670e01, 0.29400e01,
            0.31690e01, 0.34500e01, 0.35940e01, 0.37740e01,
            0.36950e01, 0.34110e01, 0.27180e01, 0.19910e01,
            0.22650e01, 0.24460e01, 0.26120e01, 0.33590e01,
            0.34290e01, 0.35330e01, 0.32610e01, 0.26120e01,
            0.21790e01, 0.16530e01, 0.18320e01, 0.23280e01,
            0.27370e01, 0.30140e01, 0.33280e01, 0.34040e01,
            0.29810e01, 0.25570e01, 0.25760e01, 0.23520e01,
            0.25560e01, 0.28640e01, 0.32140e01, 0.34350e01,
            0.34580e01, 0.33260e01, 0.28350e01, 0.24760e01,
            0.23730e01, 0.23890e01, 0.27420e01, 0.32100e01,
            0.35200e01, 0.38280e01, 0.36280e01, 0.28370e01,
            0.24060e01, 0.26750e01, 0.25540e01, 0.28940e01,
            0.32020e01, 0.32240e01, 0.33520e01, 0.31540e01,
            0.28780e01, 0.24760e01, 0.23030e01, 0.23600e01,
            0.26710e01, 0.28670e01, 0.33100e01, 0.34490e01,
            0.36460e01, 0.34000e01, 0.25900e01, 0.18630e01,
            0.15810e01, 0.16900e01, 0.17710e01, 0.22740e01,
            0.25760e01, 0.31110e01, 0.36050e01, 0.35430e01,
            0.27690e01, 0.20210e01, 0.21850e01, 0.25880e01,
            0.28800e01, 0.31150e01, 0.35400e01, 0.38450e01,
            0.38000e01, 0.35790e01, 0.32640e01, 0.25380e01,
            0.25820e01, 0.29070e01, 0.31420e01, 0.34330e01,
            0.35800e01, 0.34900e01, 0.34750e01, 0.35790e01,
            0.28290e01, 0.19090e01, 0.19030e01, 0.20330e01,
            0.23600e01, 0.26010e01, 0.30540e01, 0.33860e01,
            0.35530e01, 0.34680e01, 0.31870e01, 0.27230e01,
            0.26860e01, 0.28210e01, 0.30000e01, 0.32010e01,
            0.34240e01, 0.35310e01
        };
        double[][] printOutput;
```

```
double timsacAR[], mmAR[], mleAR[], lsAR[];
double forecasts[], residuals[];
double timsacConstant, mmConstant, mleConstant, lsConstant;
double timsacVar, timsacEquivalentVar, mmVar, mleVar, lsVar;
int maxlag = 20;
String[] colLabels = {"TIMSAC", "Method of Moments", "Least Squares",
    "Maximum Likelihood"};
String[] colLabels2 = {"Observed", "Forecast", "Residual"};
PrintMatrixFormat pmf = new PrintMatrixFormat();
PrintMatrix pm = new PrintMatrix();
NumberFormat nf = NumberFormat.getNumberInstance();
pm.setColumnSpacing(4);
nf.setMinimumFractionDigits(4);
pmf.setNumberFormat(nf);
pmf.setColumnLabels(colLabels);
System.out.println("Automatic Selection of Minimum AIC AR Model");
System.out.println("");

ARAutoUnivariate autoAR = new ARAutoUnivariate(maxlag, y);

// logging to console
Logger logger = autoAR.getLogger();
ConsoleHandler ch = new ConsoleHandler();
ch.setLevel(Level.ALL);         // default ConsoleHandler Level is INFO
logger.setLevel(Level.FINE);
logger.addHandler(ch);
ch.setFormatter(new com.imsl.IMSLFormatter());

autoAR.compute();
int orderSelected = autoAR.getOrder();
System.out.println("Minimum AIC Selected=" + autoAR.getAIC()
        + " with an optimum lag of k= " + autoAR.getOrder());
System.out.println("");

timsacAR = autoAR.getTimsacAR();
timsacConstant = autoAR.getTimsacConstant();
timsacVar = autoAR.getTimsacVariance();
lsAR = autoAR.getAR();
lsConstant = autoAR.getConstant();
lsVar = autoAR.getInnovationVariance();

autoAR.setEstimationMethod(ARAutoUnivariate.METHOD_OF_MOMENTS);
autoAR.compute();
mmAR = autoAR.getAR();
mmConstant = autoAR.getConstant();
mmVar = autoAR.getInnovationVariance();

autoAR.setEstimationMethod(ARAutoUnivariate.MAX_LIKELIHOOD);
autoAR.compute();
mleAR = autoAR.getAR();
mleConstant = autoAR.getConstant();
mleVar = autoAR.getInnovationVariance();

printOutput = new double[orderSelected + 1][4];
printOutput[0][0] = timsacConstant;
for (int i = 0; i < orderSelected; i++) {
```

```
    printOutput[i + 1][0] = timsacAR[i];
}
printOutput[0][1] = mmConstant;
for (int i = 0; i < orderSelected; i++) {
    printOutput[i + 1][1] = mmAR[i];
}
printOutput[0][2] = lsConstant;
for (int i = 0; i < orderSelected; i++) {
    printOutput[i + 1][2] = lsAR[i];
}
printOutput[0][3] = mleConstant;
for (int i = 0; i < orderSelected; i++) {
    printOutput[i + 1][3] = mleAR[i];
}
pm.setTitle("Comparison of AR Estimates");
pm.print(pmf, printOutput);

/* calculation of equivalent innovation variance using TIMSAC
 coefficients.  The Timsac innovation variance is calculated using
 only N-maxlag observations in the series.  The following code
 calculates the innovation variance using N-k observations in the
 series with the Timsac coefficient.  This illustrates that the
 least squares Timsac coefficients will not have the least value for
 the sum of squared residuals, which is calculated using all N-k
 observations. */
ARMA armaLS = new ARMA(orderSelected, 0, y);
armaLS.setArmaInfo(autoAR.getTimsacConstant(), autoAR.getTimsacAR(),
        new double[0], autoAR.getTimsacVariance());
armaLS.setBackwardOrigin(y.length - orderSelected);
forecasts = armaLS.getForecast(1);
double sumResiduals = 0.0;
for (int i = 0; i < y.length - orderSelected; i++) {
    sumResiduals += (y[i + orderSelected] - forecasts[i])
            * (y[i + orderSelected] - forecasts[i]);
}
timsacEquivalentVar = sumResiduals / (y.length - orderSelected - 1);
printOutput = new double[1][4];
printOutput[0][0] = timsacEquivalentVar;
/* the method of moments variance */
printOutput[0][1] = mmVar;
/* the least squares variance */
printOutput[0][2] = lsVar;
/* the maximum likelihood estimate of the variance */
printOutput[0][3] = mleVar;
nf.setMinimumFractionDigits(5);
pmf.setNumberFormat(nf);
pm.setTitle("Comparison of Equivalent Innovation Variances");
pm.print(pmf, printOutput);

/* FORECASTING - An example of forecasting using the maximum
 * likelihood estimates for the minimum AIC AR model.  In this example,
 * forecasts are returned for the last 10 values in the series followed
 * by the forecasts for the next 5 values.
 */
autoAR.setBackwardOrigin(10);
forecasts = autoAR.getForecast(15);
```

```
        residuals = autoAR.getResiduals();
        printOutput = new double[15][3];
        for (int i = 0; i < 10; i++) {
            printOutput[i][0] = y[y.length - 10 + i];
            printOutput[i][1] = forecasts[i];
            printOutput[i][2] = residuals[i];
        }
        for (int i = 10; i < 15; i++) {
            printOutput[i][0] = Double.NaN;
            printOutput[i][1] = forecasts[i];
            printOutput[i][2] = Double.NaN;
        }
        nf.setMaximumFractionDigits(3);
        pmf.setFirstRowNumber(105);
        pmf.setNumberFormat(nf);
        pmf.setColumnLabels(colLabels2);
        pm.setTitle("Maximum Likelihood Forecasts of Last 10 Observations & "
                + "the Next 5");
        pm.print(pmf, printOutput);
    }
}
```

Output

```
Automatic Selection of Minimum AIC AR Model

ORDER SELECTED = 11
AIC = -122.36968839314528


Minimum AIC Selected=-296.13013263562374 with an optimum lag of k= 11

ORDER SELECTED = 11
AIC = -122.36968839314528


ORDER SELECTED = 11
AIC = -122.36968839314528


                         Comparison of AR Estimates
       TIMSAC     Method of Moments    Least Squares    Maximum Likelihood
 0     1.0427          1.1679             1.1144              1.1187
 1     1.1813          1.1381             1.1481              1.1664
 2    -0.5516         -0.5061            -0.5331             -0.5420
 3     0.2314          0.2098             0.2757              0.2624
 4    -0.1780         -0.2672            -0.3263             -0.3051
 5     0.0199          0.1112             0.1685              0.1519
 6    -0.0626         -0.1246            -0.1643             -0.1460
 7     0.0286          0.0693             0.0728              0.0581
 8    -0.0507         -0.0419            -0.0305             -0.0309
 9     0.1999          0.1366             0.1509              0.1379
10     0.1618          0.1828             0.1935              0.1994
11    -0.3391         -0.3101            -0.3414             -0.3375
```

```
               Comparison of Equivalent Innovation Variances
      TIMSAC     Method of Moments    Least Squares    Maximum Likelihood
0    0.03769         0.04274            0.03687            0.03618

Maximum Likelihood Forecasts of Last 10 Observations & the Next 5
        Observed    Forecast    Residual
105     3.553       3.439        0.114
106     3.468       3.480       -0.012
107     3.187       2.924        0.263
108     2.723       2.703        0.020
109     2.686       2.556        0.130
110     2.821       2.785        0.036
111     3.000       2.949        0.051
112     3.201       3.186        0.015
113     3.424       3.385        0.039
114     3.531       3.527        0.004
115     ?           3.446       ?
116     ?           3.195       ?
117     ?           2.829       ?
118     ?           2.492       ?
119     ?           2.414       ?
```

ARAutoUnivariate.TriangularMatrixSingularException class

`static public class com.imsl.stat.ARAutoUnivariate.TriangularMatrixSingularException extends com.imsl.IMSLException`

The input triangular matrix is singular.

Constructors

ARAutoUnivariate.TriangularMatrixSingularException

`public ARAutoUnivariate.TriangularMatrixSingularException(String message)`

Description

Constructs a `TriangularMatrixSingularException` object.

Parameter

`message` – a `String` containing the error message

ARAutoUnivariate.TriangularMatrixSingularException

```
public ARAutoUnivariate.TriangularMatrixSingularException(String key, Object[]
arguments)
```

Description

Constructs a `TriangularMatrixSingularException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ARAutoUnivariate.Formatter class

```
static public class com.imsl.stat.ARAutoUnivariate.Formatter extends
java.util.logging.Formatter
```

Simple formatter for ARAutoUnivariate logging.

Constructor

ARAutoUnivariate.Formatter

```
public ARAutoUnivariate.Formatter()
```

Method

format

```
public String format(LogRecord record)
```

ARSeasonalFit class

```
public class com.imsl.stat.ARSeasonalFit implements Serializable
```

Estimates the optimum seasonality parameters for a time series using an autoregressive model, AR(p), to represent the time series.

ARMA time series modeling assumes the time series is stationary. Seasonal trends and cycles violate this assumption, which can lead to inaccurate predictions. However, in many cases the nonstationary series can be transformed into a stationary series by first differencing the series. For example, if the correlation is strong from one period to the next, the series might be differenced by a lag of 1. Instead of fitting a model to the original series Z_t, the model is fitted to the transformed series: $W_t = Z_t - Z_{t-1}$. Higher order lags or differences are warranted if the series has cycles every 4 or 13 weeks. Class `ARSeasonalFit` is designed to help identify the optimum differencing for a series with seasonal trends or cycles.

`ARSeasonalFit` assumes the original series has no missing values, is equally spaced in time and is not centered before computing the optimum differencing. However, by default the transformed series is centered using the mean of that series. Users can change this default using the `setCenter` method. If `setCenter` is set to `NO_CENTER` the series is not centered, if set to `CENTER_MEAN` the series is centered using the mean of the series, and if set to `CENTER_MEDIAN` , the series is centered using the median of the series. If `setCenter` is set to `CENTER_MEAN` or `CENTER_MEDIAN` then the differenced series, W_t is centered before determination of minimum AIC and optimum lag.

For every combination of rows in `sInitial` and `dInitial` , the series Z_t is converted to the seasonally adjusted series using the following computation

$$W_t(s,d) = \Delta_{s_1}^{d_1}\Delta_{s_2}^{d_2}\cdots\Delta_{s_m}^{d_m}Z_t = \prod_{i=1}^{m}\sum_{j=0}^{d_i}\binom{d_i}{j}(-1)^j B^{j\cdot s_i}Z_t$$

where $s := (s_1,\ldots,s_m)$, $d := (d_1,\ldots,d_m)$ represent specific rows of arrays `sInitial` and `dInitial` respectively, and m is the number of differences, or m=`sInitial[0].length`.

This transformation of the series Z_t to $W_t(s,d)$ is computed using the `Difference` class. After this transformation the transformed series

$$W_t(s,d)$$

is centered, unless `NO_CENTER` is specified, and the `ARAutoUnivariate` class is used to automatically determine the optimum lag for an AR(p) representation for $W_t(s,d)$.

This procedure is repeated for every possible combination of rows in `sInitial` and `dInitial`. The series with the minimum AIC is identified as the optimum representation and returned in the methods `getAROrder` , `getOptimumS` , `getOptimumD` , `getAIC` , and `getAR` . The transformed series with the minium AIC can be retrieved from the `getTransformedTimeSeries` method.

Fields

CENTER_MEAN

```
static final public int CENTER_MEAN
```

Indicates the transformed series should be centered using the average of the differenced series.

CENTER_MEDIAN

```
static final public int CENTER_MEDIAN
```

Indicates the transformed series should be centered using the median of the differenced series.

NO_CENTER

```
static final public int NO_CENTER
```

Indicates the transformed series should not be centered.

Constructor

ARSeasonalFit

```
public ARSeasonalFit(int maxlag, int[][] sInitial, double[] z)
```

Description

Constructor for `ARSeasonalFit`.

Parameters

`maxlag` – an `int` scalar specifying the maximum lag allowed when fitting an AR(p) model.

`sInitial` – an `int` matrix where each row represents seasonal differences to evaluate. The number of columns in `sInitial` represent the number of differences to perform. All values of `sInitial` must be greater than zero.

`z` – an input `double` array containing the time series.

Methods

compute

```
public void compute() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException
```

Description

Computes the minimum AIC and optimum values for s and d based upon the candidates provided in `sInitial` and `dInitial`, and computes the values for the transformed series, $W_t(s,d)$.

Warnings are printed if a row in `sInitial` is skipped due to too many observations lost in the differenced series or if problems occurred computing the optimum AIC using the differenced series.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

getAIC

```
public double getAIC()
```

Description

Returns the final estimate for Akaike's Information Criterion (AIC) at the optimum.

Returns

a `double` equal to $\text{AIC} = -2\ln(L) + 2p$, where L is the value of the maximum likelihood function evalutated at the parameter estimates.

getAR

```
public double[] getAR() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException
```

Description

Returns the final autoregressive parameter estimates at the optimum in the transformed series W_t.

Returns

a `double` array containing the estimates for the autoregressive parameters

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

getAROrder

```
public int getAROrder()
```

Description

Returns optimum number of lags, p, for the optimum autoregressive AR(p) model. This is the value of p for the transformed series, W_t.

Returns

an `int` containing the optimum number of lags in the autoregressive model used to fit the transformed series W_t.

getCenter

```
public int getCenter()
```

Description

Returns the current setting for centering the input time series.

Returns

an `int` containing the setting for center equal to 0, 1, or 2, which implies `NO_CENTER` , `CENTER_MEAN` , `CENTER_MEDIAN` respectively.

getDInitial

```
public int[][] getDInitial()
```

Description

Returns the candidate values for *d* to evaluate.

Returns

an `int` matrix containing the candidate values for *d* to evaluate

getExclude

```
public boolean getExclude()
```

Description

Returns the current setting for excluding or replacing the inital values in the transformed series.

If exclude is `true`, then inital values in the transformed series that cannot be computed are set to missing, `NaN`. This ensures that the length of the transformed series W_t is equal to the length of the time series, `z.length` . If exclude is set to `false`, then inital values in the transformed series W_t that cannot be computed are removed. This makes the length of the transformed series W_t equal to `z.length`-*nLost* where *nLost* is the number of lost values obtained from method `getNLost` .

getMaxlag

```
public int getMaxlag()
```

Description

Returns the maximum lag used to fit the AR(p) model.

Returns

an `int` scalar containing the maximum lag allowed when fitting an AR(p) model

getNLost

```
public int getNLost() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException
```

Description

Returns the number of values in the initial part of the series lost to differencing.

Returns

an `int` containing the number of values in the initial part of the series lost to differencing. These lost values will be set to missing or dropped in the transformed series, depending upon the setting for exclude.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

getOptimumD

```
public int[] getOptimumD() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException
```

Description

Returns the optimum values for d selected among the candidates in `dInitial`.

Returns

an `int` array of length `dInitial[0].length` containing the optimum values for d selected among the candidates in `dInitial`

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

getOptimumS

```
public int[] getOptimumS() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException
```

Description

Returns the optimum values for *s* selected among the candidates in `sInitial`.

Returns

an `int` array of length `sInitial[0].length` containing the optimum values for *s* selected among the candidates in `sInitial`

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

getSInitial

```
public int[][] getSInitial()
```

Description

Returns the the candidate values for s to evaluate.

Returns

an `int` matrix containing the candidate values for s to evaluate

getTimeSeries

```
public double[] getTimeSeries()
```

Description

Returns the time series.

Returns

a `double` array containing the time series values

getTransformedTimeSeries

```
public double[] getTransformedTimeSeries() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException
```

Description

Returns the transformed series, $W_t(s,d)$.

$W_t(s,d)$ is an array of length `z.length` or `z.length-nLost` containing the optimum seasonally adjusted, autoregressive series, where `nLost` is the first lost observations in this series that are dropped due to differencing. If the missing values are not dropped the first `nLost` values of W_t will be set (`Double.NaN`). The `getNLost` method can be used to obtain `nLost`.

The seasonal adjustment is done by selecting optimum values for $d_1,\ldots,d_m$, $s_1,\ldots,s_m$ (m is number of differences or `sIinitial[0].length`) and p in the AR model:

$$\phi_p(B)(\Delta_{s_1}^{d_1}\Delta_{s_2}^{d_2}\ldots\Delta_{s_m}^{d_m}Z_t-\mu)=a_t,$$

where $\{Z_t\}$ is the original time series, B is the backward shift operator defined by $B^k Z_t = Z_{t-k}$, with $k \geq 0$, a_t is Gaussian white noise with $E[a_t] = 0$ and $\mathrm{Var}[a_t] = \sigma^2$, $\phi_p(B) = 1 - \phi_1 B - \phi_2 B^2 - \cdots - \phi_p B^p$, $0 \leq p \leq \mathrm{maxlag}$, $\Delta_s^d = (1 - B^s)^d$, with $s > 0, d \geq 0$, and μ is a centering parameter for the differenced series.

Returns

a `double` array of length `z.length` or `z.length-nLost`, depending upon the setting for `setExclude`

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input triangular matrix is singular.

setCenter

```
public void setCenter(int center)
```

Description

Controls centering of the differenced series.

Parameter

`center` – an `int` scalar value equal `NO_CENTER` , `CENTER_MEAN` , or `CENTER_MEDIAN` . By default, `center=CENTER_MEAN`

setDInitial

```
public void setDInitial(int[][] dInitial)
```

Description

Sets the candidate values for selecting the optimum seasonal adjustment prior to calling the compute method.

Parameter

`dInitial` – an `int` array of candidate values for *d* to evaluate. All values must be non-negative. `dInitial` must have the same number of differences (columns) as `sInitial`. By default, `dInitial` is initialized to all ones.

setExclude

```
public void setExclude(boolean exclude)
```

Description

Controls whether to exclude or replace the inital values in the transformed series.

If exclude is `true`, then inital values in the transformed series that cannot be computed are set to missing, `NaN`. This ensures that the length of the transformed series W_t is equal to the length of the time series, `z.length` . If exclude is set to `false`, then inital values in the transformed series W_t that cannot be computed are removed. This makes the length of the transformed series W_t equal to `z.length`-*nLost* where *nLost* is the number of lost values obtained from method `getNLost` .

Parameter

`exclude` – a `boolean` value that controls whether values in W_t that cannot be calculated are dropped or set to missing. Missing values are set to `Double.NaN`. By default, `exclude =true`

Example: ARSeasonalFit

Consider the Airline Data (Box, Jenkins and Reinsel 1994, p. 547) consisting of the monthly total number of international airline passengers from January 1949 through December 1960. Class `ARSeasonalFit` is used to compute the optimum seasonality representation of the adjusted series

$$W_t(s,d) = \Delta_{s_1}^{d_1}\Delta_{s_2}^{d_2}Z_t = (1-B^{s_1})^{d_1}(1-B^{s_2})^{d_2}Z_t,$$

where

$$s = (1,1)$$

or

$$s = (1,12)$$

and

$$d = (1,1).$$

As differenced series with minimum AIC,

$$W_t = \Delta_1^1\Delta_{12}^2 Z_t = (Z_t - Z_{t-12}) - (Z_{t-1} - Z_{t-13}),$$

is identified.

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

public class ARSeasonalFitEx1 {

    public static void main(String args[]) throws Exception {
        double[] z = {
            112, 118, 132, 129, 121, 135, 148, 148, 136, 119, 104, 118,
            115, 126, 141, 135, 125, 149, 170, 170, 158, 133, 114, 140,
            145, 150, 178, 163, 172, 178, 199, 199, 184, 162, 146, 166,
            171, 180, 193, 181, 183, 218, 230, 242, 209, 191, 172, 194,
            196, 196, 236, 235, 229, 243, 264, 272, 237, 211, 180, 201,
            204, 188, 235, 227, 234, 264, 302, 293, 259, 229, 203, 229,
            242, 233, 267, 269, 270, 315, 364, 347, 312, 274, 237, 278,
            284, 277, 317, 313, 318, 374, 413, 405, 355, 306, 271, 306,
            315, 301, 356, 348, 355, 422, 465, 467, 404, 347, 305, 336,
            340, 318, 362, 348, 363, 435, 491, 505, 404, 359, 310, 337,
            360, 342, 406, 396, 420, 472, 548, 559, 463, 407, 362, 405,
            417, 391, 419, 461, 472, 535, 622, 606, 508, 461, 390, 432
        };

        int sInit[][] = {{1, 1}, {1, 12}};

        ARSeasonalFit seasFit = new ARSeasonalFit(10, sInit, z);
        seasFit.compute();
        System.out.println("NLost = " + seasFit.getNLost());
        System.out.println("aic =" + seasFit.getAIC());
        new PrintMatrix("Best Periods (Optimum S)").
                print(seasFit.getOptimumS());
        new PrintMatrix("Best Orders (Optimum D)").
                print(seasFit.getOptimumD());
        System.out.println("Best AR order selected = " + seasFit.getAROrder());
        new PrintMatrix("AR Coefficients").print(seasFit.getAR());

        System.out.println("");
        double w[] = seasFit.getTransformedTimeSeries();
        double pack[][] = new double[30][2];
        for (int i = 0; i < 30; i++) {
            pack[i][0] = z[i];
            pack[i][1] = w[i];
        }
        PrintMatrix pm = new PrintMatrix();
        String str = "First 30 elements of the original time series "
                + "and differenced series";
        pm.setTitle(str);
        PrintMatrixFormat fmt = new PrintMatrixFormat();
        fmt.setColumnLabels(new String[]{"Original", "Differenced"});
        pm.print(fmt, pack);
    }
}
```

Output

```
NLost = 13
aic =606.3972456534914
```

```
Best Periods (Optimum S)
   0
0   1
1  12

Best Orders (Optimum D)
   0
0  1
1  1

Best AR order selected = 1
AR Coefficients
     0
0  -0.31


First 30 elements of the original time series and differenced series
    Original  Differenced
 0    112          ?
 1    118          ?
 2    132          ?
 3    129          ?
 4    121          ?
 5    135          ?
 6    148          ?
 7    148          ?
 8    136          ?
 9    119          ?
10    104          ?
11    118          ?
12    115          ?
13    126          5
14    141          1
15    135         -3
16    125         -2
17    149         10
18    170          8
19    170          0
20    158          0
21    133         -8
22    114         -4
23    140         12
24    145          8
25    150         -6
26    178         13
27    163         -9
28    172         19
29    178        -18
```

ARMA class

`public class com.imsl.stat.ARMA implements Serializable, Cloneable`

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`	`setMean`
`setInitialEstimates`	`getMean`
`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 $\hat{\mu} = \text{zMean}$ be the estimate of the mean μ of the time series $\{Z_t\}$, where $\hat{\mu}$ equals the following:

$$\hat{\mu} = \begin{cases} \mu & \text{for } \mu \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} Z_t & \text{for } \mu \text{ unknown} \end{cases}$$

The autocovariance function is estimated by

$$\hat{\sigma}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(Z_t - \hat{\mu})(Z_{t+k} - \hat{\mu})$$

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

$$\hat{\phi} = (\hat{\phi}_1, \ldots, \hat{\phi}_p)^T$$

$$\hat{\Sigma}_{ij} = \hat{\sigma}(|q + i - j|), \; i, j = 1, \ldots, p$$

$$\hat{\sigma}_i = \hat{\sigma}(q + i), \; i = 1, \ldots, p$$

The overall constant θ_0 is estimated by the following:

$$\hat{\theta}_0 = \begin{cases} \hat{\mu} \text{ for } p = 0 \\ \hat{\mu}\left(1 - \sum_{i=1}^{p}\hat{\phi}_i\right) \text{ for } p > 0 \end{cases}$$

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 ϕ_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:

$$\hat{\sigma}'(k) = \begin{cases} \hat{\sigma}(k) \text{ for } p = 0 \\ \sum_{i=0}^{p}\sum_{j=0}^{p}\hat{\phi}_i\hat{\phi}_j(\hat{\sigma}(|k + i - j|)) \text{ for } p \geq 1, \hat{\phi}_0 \equiv -1 \end{cases}$$

The iterative procedure for determining the moving average parameters is based on the relation

$$\sigma(k) = \begin{cases} \left(1 + \theta_1^2 + \ldots + \theta_q^2\right)\sigma_A^2 \text{ for } k = 0 \\ \left(-\theta_k + \theta_1\theta_{k+1} + \ldots + \theta_{q-k}\theta_q\right)\sigma_A^2 \text{ for } k \geq 1 \end{cases}$$

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

$$f_j = \sum_{i=0}^{q-j} \tau_i \tau_{i+j} - \hat{\sigma}'(j) \text{ for } j = 0, 1, \ldots, q$$

Then, the value of τ 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 τ by setting

$$\hat{\theta}_j = -\tau_j/\tau_0 \text{ for } j = 1, \ldots, q$$

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,

$$\phi(B)(Z_t - \mu) = \theta(B)A_t \text{ for } t \in \{0, \pm 1, \pm 2, \ldots\}$$

where B is the backward shift operator, μ is the mean of Z_t, and

$$\phi(B) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ldots - \phi_p B^{l_\phi(p)} \quad \text{for } p \geq 0$$

$$\theta(B) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ldots - \theta_q B^{l_\theta(q)} \quad \text{for } q \geq 0$$

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 $\mu, \phi, \text{and}\, \theta$.

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 $t \in \{0, \pm1, \pm2, \ldots\}$, where B is the backward shift operator, θ_0 is the constant, and

$$\phi\left(B\right) = 1 - \phi_1 B^{l_\phi(1)} - \phi_2 B^{l_\phi(2)} - \ \ldots \ - \phi_p B^{l_\phi(p)}$$

$$\theta\left(B\right) = 1 - \theta_1 B^{l_\theta(1)} - \theta_2 B^{l_\theta(2)} - \ \ldots \ - \theta_q B^{l_\theta(q)}$$

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

$$\hat{Z}_t(l) = \theta_0 + \phi_1 \left[Z_{t+l-l_\phi(1)}\right] + \ldots + \phi_p \left[Z_{t+l-l_\phi(p)}\right]$$

$$+ [A_{t+l}] - \theta_1 \left[A_{t+l-l_\theta(1)}\right] - \ldots - \theta_q \left[A_{t+l-l_\theta(q)}\right]$$

where the following is true:

$$[Z_{t+k}] = \begin{cases} Z_{t+k} \text{ for } k = 0,\, -1,\, -2,\, \ldots \\ \hat{Z}_t(k) \text{ for } k = 1,\, 2,\, \ldots \end{cases}$$

$$[A_{t+k}] = \begin{cases} Z_{t+k} - \hat{Z}_{t+k-1}(1) \text{ for } k = 0,\, -1,\, -2,\, ... \\ 0 \text{ for } k = 1,\, 2,\, ... \end{cases}$$

The $100(1-\alpha)$ percent probability limits for Z_{t+l} are given by

$$\hat{Z}_t(l) \pm z_{\alpha/2} \left\{ 1 + \sum_{j=1}^{l-1} \psi_j^2 \right\}^{1/2} \sigma_A$$

where $z_{\alpha/2}$ 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 $L =$ nForecast and $b =$ backwardOrigin.

The Box-Jenkins forecasts minimize the mean-square error

$$E\left[Z_{t+l} - \hat{Z}_t(l)\right]^2$$

Also, the forecasts can be easily updated according to the following equation:

$$\hat{Z}_{t+1}(l) = \hat{Z}_t(l+1) + \psi_l A_{t+1}$$

This approach and others are discussed in Chapter 5 of Box and Jenkins (1976).

Fields

LEAST_SQUARES

```
static final public int LEAST_SQUARES
```

Indicates autoregressive and moving average parameters are estimated by a least-squares procedure.

METHOD_OF_MOMENTS

```
static final public int METHOD_OF_MOMENTS
```

Indicates autoregressive and moving average parameters are estimated by a method of moments 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

Exception

`IllegalArgumentException` is thrown if `p`, `q`, and `z.length` are not consistent.

Methods

compute

```
final public void compute() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException
```

Description

Computes least-square estimates of parameters for an ARMA model.

Exceptions

`MatrixSingularException` is thrown if the input matrix is singular.

`TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`NewInitialGuessException` is thrown if the iteration has not made good progress.

`IllConditionedException` is thrown if the problem is ill-conditioned.

`TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

forecast

`final public double[][] forecast(int nForecast)`

Description

Computes forecasts and their associated probability limits for an ARMA model.

Parameter

`nForecast` – an `int` scalar containing the maximum lead time for forecasts. `nForecast` must be greater than 0.

Returns

a `double` matrix of dimensions of `nForecast` by `backwardOrigin + 1` containing the forecasts. The forecasts are for lead times $l = 1, 2, \ldots, nForecast$ at origins `z.length-backwardOrigin-1+j` where $j = 1, \ldots, backwardOrigin + 1$. Returns `NULL` if the least-square estimates of parameters is not computed.

getAR

`public double[] getAR()`

Description

Returns the final autoregressive parameter estimates. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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`. Note that the `compute` method must be invoked before this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` array containing the autocovariances of lag `k`, where `k = 1, ..., p + q + 1`

getBackwardOrigin

`public int getBackwardOrigin()`

Description

Returns the user-specified backward origin

Returns

an `int` scalar containing the user-specified backward origin

getConstant

`public double getConstant()`

Description

Returns the constant parameter estimate. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `NaN`.

Returns

a `double` scalar containing the constant parameter estimate

getDeviations

`public double[] getDeviations()`

Description

Returns the deviations used for calculating the forecast confidence limits.

Returns

A `double` array of length `nForecast` containing the deviations for calculating forecast confidence intervals. The confidence level is specified in `confidence`. By default, `confidence`= 0.95.

getForecast

`public double[] getForecast(int nForecast)`

Description

Returns forecasts

Parameter

`nForecast` – An input `int` representing the number of requested forecasts beyond the last value in the series.

Returns

A `double` array containing the `nForecast+backwardOrigin` forecasts. The first `backwardOrigin` forecasts are one-step ahead forecasts for the last `backwardOrigin` values in the series. The next `nForecast` values in the returned series are forecasts for the next values beyond the series.

getInnovationVariance

`public double getInnovationVariance()`

Description

Returns the variance of the random shock.

Returns

a `double` scalar equal to the variance of the random shock.

getMA

```
public double[] getMA()
```

Description

Returns the final moving average parameter estimates. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` array of length `q` containing the final moving average parameter estimates

getMaxFunctionEvaluations

```
public int getMaxFunctionEvaluations()
```

Description

Returns the maximum number of function evaluations.

Returns

an `int`, the maximum number of function evaluations allowed in the nonlinear equation solver

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the maximum number of iterations.

Returns

an `int`, the maximum number of iterations allowed in the nonlinear equation solver

getMean

```
public double getMean()
```

Description

Returns an update of the mean of the time series `z`. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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.

getNumberOfBackcasts

```
public int getNumberOfBackcasts()
```

Description

Returns the number of backcasts used to calculate the AR coefficients for the time series `z`. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

Returns

an `int` scalar containing the number of backcasts calculated, this value will be less than or equal to the maximum number of backcasts set in the `setBackcasting` method.

getParamEstimatesCovariance

```
public double[][] getParamEstimatesCovariance()
```

Description

Returns the covariances of parameter estimates. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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. Note that the `forecast` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` array of length `nForecast` containing the psi weights of the infinite order moving average form of the model.

getResidual

```
public double[] getResidual()
```

Description

Returns the residuals. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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, nb=ARMA.getNumberOfBackcasts(). This method is only applicable using least-squares algorithm.

getSSResidual

```
public double getSSResidual()
```

Description

Returns the sum of squares of the random shock. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

Returns

a `double` scalar containing the sum of squares of the random shock, $\text{residual}[0]^2 + \ldots + \text{residual}[\text{na}-1]^2$, 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`. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `NaN`.

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.

Parameter

`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]`

setArmaInfo

`public void setArmaInfo(double constant, double[] ar, double[] ma, double var)`

Description

Sets the ARMA_Info Object to previously determined values

Parameters

`constant` – a `double` scalar equal to the constant term in the ARMA model.

`ar` – a `double` array of length `p` containing estimates of the autoregressive parameters.

`ma` – a `double` array of length `q` containing estimates of the moving average parameters.

`var` – a `double` scalar equal to the innovation variance

setBackcasting

`public void setBackcasting(int maxBackcast, double tolerance)`

Description

Sets backcasting option.

Parameters

`maxBackcast` – an `int` scalar containing the maximum length of backcasting and must be greater than or equal to 0. Default: `maxBackcast = 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.

Parameter

`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.

Parameter

`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 level for calculating confidence limit deviations returned from `getDeviations`.

Parameter

`confidence` – a `double` scalar specifying the confidence level used in computing forecast confidence intervals. 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.

Parameter

`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 $\max(10^{-10}, \mathrm{eps}^{2/3})$, where `eps = 2.2204460492503131e-16`.

setInitialEstimates

`public void setInitialEstimates(double[] ar, double[] ma)`

Description

Sets preliminary estimates for the `LEAST_SQUARES` estimation method. The values of the autoregressive and moving average parameters submitted are used as intial values for least squares estimation. Otherwise they are initialized to values computed using the method of moments. When the estimation method is set to `METHOD_OF_MOMENTS` these initial values are not used.

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.

Parameter

`maLags` – an `int` array of length `q` containing the order of the moving average parameters. The elements of `maLags` must be greater than or equal to 1. Default: `maLags = [1, 2, ..., q]`

setMaxFunctionEvaluations

`public void setMaxFunctionEvaluations(int evaluations)`

Description

Sets the maximum number of function evaluations.

Parameter

`evaluations` – an `int` scalar specifying the maximum number of function evaluations allowed in the nonlinear equation solver used in both the method of moments and least-squares algorithms. Default: `evaluations = 400`.

setMaxIterations

`public void setMaxIterations(int iterations)`

Description

Sets the maximum number of iterations.

Parameter

`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`.

setMean

`public void setMean(double zMean)`

Description

Sets an initial estimate of the mean of the time series `z`.

Parameter

`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 estimation method used for estimating the ARMA parameters.

Parameter

`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.

Parameter

`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 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.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 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.setMean(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
```

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. An ARMA(2,1) model is fitted to these data using the Method of Moments. With `backward_origin` = 3, the `forecast` method is used to obtain forecasts starting from 1866, 1867, 1868, and 1869, respectively. Note that the values in the first row of the matrix returned by this method are the one-step ahead forecasts for 1867, 1868, ..., 1870. The values in the second row are the two-step ahead forecasts for 1868, 1869, ..., 1871, etc.

Method `getForecast` is used to compute the one-step ahead forecasts setting `backward_origin` = 10. This obtains the one-step ahead forecasts for the last 10 observations in the series, i.e. years 1860-1869, plus the next 5 years. The upper 90% confidence limits are computed for these forecasts using the `getDeviations` method.

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

public class ARMAEx3 {

    public static void main(String args[]) throws Exception {
        /* sunspots from 1770 to 1869 */
        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
};
int backwardOrigin = 3;
double[][] printTable = new double[15][4];
double[][] printEstimates = new double[1][4];
double[] forecasts, deviations;
PrintMatrixFormat pmf = new PrintMatrixFormat();
PrintMatrix pm = new PrintMatrix();
NumberFormat nf = NumberFormat.getNumberInstance();
pm.setColumnSpacing(3);

ARMA arma = new ARMA(2, 1, z);
arma.compute();

System.out.println("ARMA ESTIMATES");
double[] ar = arma.getAR();
double[] ma = arma.getMA();
printEstimates[0][0] = arma.getConstant();
printEstimates[0][1] = ar[0];
printEstimates[0][2] = ar[1];
printEstimates[0][3] = ma[0];
String[] estimateLabels = {"Constant", "AR(1)", "AR(2)", "MA(1)"};
pmf.setColumnLabels(estimateLabels);
nf.setMinimumFractionDigits(5);
nf.setMaximumFractionDigits(5);
pmf.setNumberFormat(nf);
pm.setTitle("ARMA ESTIMATES");
pm.print(pmf, printEstimates);
arma.setBackwardOrigin(backwardOrigin);

String[] labels = {"From 1866", "From 1867",
    "From 1868", "From 1869"};
pmf.setColumnLabels(labels);
pmf.setFirstRowNumber(1);
nf.setMinimumFractionDigits(1);
nf.setMaximumFractionDigits(1);
pmf.setNumberFormat(nf);
pm.setTitle("FORECASTS");
pm.print(pmf, arma.forecast(5));

/* FORECASTING - An example of forecasting using the ARMA estimates
 * In this case, forecasts are returned for the last 10 values in the
 * series followed by the forecasts for the next 5 values.
 */
String[] forecastLabels = {
    "Observed", "Forecast", "Residual", "UCL(90%)"
};
pmf.setColumnLabels(forecastLabels);
backwardOrigin = 10;
arma.setBackwardOrigin(backwardOrigin);
int n_forecast = 5;
arma.setConfidence(0.9);
forecasts = arma.getForecast(n_forecast);
deviations = arma.getDeviations();
```

```
        for (int i = 0; i < backwardOrigin; i++) {
            printTable[i][0] = z[z.length - backwardOrigin + i];
            printTable[i][1] = forecasts[i];
            printTable[i][2] = z[z.length - backwardOrigin + i] - forecasts[i];
            printTable[i][3] = forecasts[i] + deviations[0];
        }
        for (int i = backwardOrigin; i < n_forecast + backwardOrigin; i++) {
            printTable[i][0] = Double.NaN;
            printTable[i][1] = forecasts[i];
            printTable[i][2] = Double.NaN;
            printTable[i][3] = forecasts[i] + deviations[i - backwardOrigin];
        }
        pmf.setFirstRowNumber(1869 - backwardOrigin + 1);
        pm.setTitle("ARMA ONE-STEP AHEAD FORECASTS");
        pm.print(pmf, printTable);
    }
}
```

Output

```
ARMA ESTIMATES
                ARMA ESTIMATES
    Constant    AR(1)      AR(2)      MA(1)
0   15.54398   1.24426   -0.57515   -0.12409

                     FORECASTS
    From 1866   From 1867   From 1868   From 1869
1     17.3        14.0        60.6        87.7
2     27.7        28.8        69.6        82.1
3     40.1        43.3        67.2        67.3
4     49.5        52.9        59.2        52.1
5     54.0        56.4        50.5        41.6

           ARMA ONE-STEP AHEAD FORECASTS
       Observed   Forecast   Residual   UCL(90%)
1860     95.7      103.4       -7.7      131.3
1861     77.2       79.7       -2.5      107.6
1862     59.1       56.2        2.9       84.1
1863     44.0       45.0       -1.0       72.9
1864     47.0       36.2       10.8       64.0
1865     30.5       50.1      -19.6       77.9
1866     16.3       24.0       -7.7       51.9
1867      7.3       17.3      -10.0       45.2
1868     37.3       14.0       23.3       41.9
1869     73.9       60.6       13.3       88.5
1870      ?         87.7        ?        115.6
1871      ?         82.1        ?        129.4
1872      ?         67.3        ?        124.1
1873      ?         52.1        ?        111.3
1874      ?         41.6        ?        101.0
```

ARMA.TooManyCallsException class

`static public class com.imsl.stat.ARMA.TooManyCallsException extends com.imsl.IMSLException`

The number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

Constructors

ARMA.TooManyCallsException

`public ARMA.TooManyCallsException(String message)`

Description

Constructs an `TooManyCallsException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.TooManyCallsException

`public ARMA.TooManyCallsException(String key, 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

ARMA.IncreaseErrRelException class

`static public class com.imsl.stat.ARMA.IncreaseErrRelException extends com.imsl.IMSLException`

The bound for the relative error is too small.

Constructors

ARMA.IncreaseErrRelException

`public ARMA.IncreaseErrRelException(String message)`

Description

Constructs an `IncreaseErrRelException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.IncreaseErrRelException

`public ARMA.IncreaseErrRelException(String key, 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

ARMA.NewInitialGuessException class

`static public class com.imsl.stat.ARMA.NewInitialGuessException extends com.imsl.IMSLException`

The iteration has not made good progress.

Constructors

ARMA.NewInitialGuessException

`public ARMA.NewInitialGuessException(String message)`

Description

Constructs an `NewInitialGuessException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.NewInitialGuessException

`public ARMA.NewInitialGuessException(String key, 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

ARMA.MatrixSingularException class

`static public class com.imsl.stat.ARMA.MatrixSingularException extends com.imsl.IMSLException`

The input matrix is singular.

Constructors

ARMA.MatrixSingularException

`public ARMA.MatrixSingularException(String message)`

Description

Constructs an `MatrixSingularException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.MatrixSingularException

`public ARMA.MatrixSingularException(String key, 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

ARMA.TooManyITNException class

`static public class com.imsl.stat.ARMA.TooManyITNException extends com.imsl.IMSLException`

Maximum number of iterations exceeded.

Constructors

ARMA.TooManyITNException

`public ARMA.TooManyITNException(String message)`

Description

Constructs an `TooManyITNException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.TooManyITNException

`public ARMA.TooManyITNException(String key, Object[] arguments)`

Description

Constructs an `TooManyITNException` 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

ARMA.TooManyFcnEvalException class

`static public class com.imsl.stat.ARMA.TooManyFcnEvalException extends com.imsl.IMSLException`

Maximum number of function evaluations exceeded.

Constructors

ARMA.TooManyFcnEvalException

`public ARMA.TooManyFcnEvalException(String message)`

Description

Constructs an `TooManyFcnEvalException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.TooManyFcnEvalException

`public ARMA.TooManyFcnEvalException(String key, 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

ARMA.TooManyJacobianEvalException class

`static public class com.imsl.stat.ARMA.TooManyJacobianEvalException extends com.imsl.IMSLException`

Maximum number of Jacobian evaluations exceeded.

Constructors

ARMA.TooManyJacobianEvalException

`public ARMA.TooManyJacobianEvalException(String message)`

Description

Constructs an `TooManyJacobianEvalException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.TooManyJacobianEvalException

```
public ARMA.TooManyJacobianEvalException(String key, 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

ARMA.IllConditionedException class

```
static public class com.imsl.stat.ARMA.IllConditionedException extends
com.imsl.IMSLException
```

The problem is ill-conditioned.

Constructors

ARMA.IllConditionedException

```
public ARMA.IllConditionedException(String message)
```

Description

Constructs an `IllConditionedException` with the specified detail message. A detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message

ARMA.IllConditionedException

```
public ARMA.IllConditionedException(String key, 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

ARMAEstimateMissing class

`public class com.imsl.stat.ARMAEstimateMissing implements Serializable`

Estimates missing values in a time series collected with equal spacing. Missing values can be replaced by these estimates prior to fitting a time series using the `ARMA` class.

Traditional time series analysis as described by Box, Jenkins and Reinsel (1994) requires the observations be made at equidistant time points $t_0, t_1, \ldots, t_n$ where $t_i = t_0 + i$. When observations are missing, `ARMA` requires that they be replaced with suitable estimates. Class `ARMAEstimateMissing` offers 4 methods for estimating missing values: `MEDIAN, CUBIC_SPLINE, AR_1,` and `AR_P`

Method `MEDIAN` estimates the missing observations in a gap by the median of the last four time series values before and the first four values after the gap. If not enough values are available before or after the gap then the number is reduced accordingly. This method is very fast and simple, but its use is limited to stationary ergodic series without outliers and level shifts.

Method `CUBIC_SPLINE` uses a cubic spline interpolation method to estimate missing values. Here the interpolation is again done over the last four time series values before and the first four values after the gap. The missing values are estimated by the resulting interpolant. This method gives smooth transitions across missing values.

Method `AR_1` assumes that the time series before the gap can be approximated using an AR(1) process. If the last observation prior to the gap is made at time point t_m then this method uses values at $t_0, t_1, \ldots, t_m$ to compute the one-step-ahead forecast at origin t_m. This value is used to estimate the missing value at time point $t_m + 1$. If the value at $t_m + 2$ is also missing then the values at time points $t_0, t_1, \ldots, t_m + 1$ are used to recompute the AR(1) model, and then estimate the value at $t_m + 2$ and so on. The coefficient ϕ_1 in the AR(1) model is computed internally by the method of least squares from class ARMA.

Finally, method `AR_P` uses an AR(p) model to estimate missing values using a one-step-ahead forecast similar to method `AR_1`. First, class `ARAutoUnivariate`, is applied to the time series values just prior to the missing values to determine the optimum `p` from the set $\{0, 1, \ldots, \texttt{maxlag}\}$ of possible values and to compute the parameters $\phi_1, \ldots, \phi_p$ of the resulting AR(p) model. The parameters are estimated by the least squares method based on Householder transformations as described in Kitagawa and Akaike (1978). Denoting the mean of the series $y_{t_0}, y_{t_1}, \ldots, y_{t_m}$ by μ the one-step-ahead forecast at origin t_m, $\hat{y_{t_m}}(1)$, can be computed by the formula

$$\hat{y_{t_m}}(1) = \mu(1 - \sum_{j=1}^{p} \phi_j) + \sum_{j=1}^{p} \phi_j y_{t_m+1-j}.$$

This value is used as an estimate for the missing value at t_{m+1}. The procedure starting with `ARAutoUnivariate` is then repeated for every further missing value in the gap. All four estimation methods treat gaps of missing values in increasing time order.

Fields

AR_1

```
static final public int AR_1
```

Indicates that missing values should be estimated using an autoregressive time series with 1 lag.

AR_P

```
static final public int AR_P
```

Indicates that missing values should be estimated using an autoregressive time series with a maximum lag of `maxLag`. By default `maxLag=10`, but this can be changed using the `setMaxlag` method.

CUBIC_SPLINE

```
static final public int CUBIC_SPLINE
```

Indicates that missing values should be estimated using cublic spline interpolation.

LEAST_SQUARES

```
static final public int LEAST_SQUARES
```

Estimate autoregressive coefficients using least squares.

MAX_LIKELIHOOD

```
static final public int MAX_LIKELIHOOD
```

Estimate autoregressive coefficients using maximum likelihood.

MEDIAN

```
static final public int MEDIAN
```

Indicates that missing values should be estimated using the median of the values just before and after the missing value gap.

METHOD_OF_MOMENTS

```
static final public int METHOD_OF_MOMENTS
```

Estimate autoregressive coefficients using method of moments.

Constructor

ARMAEstimateMissing

```
public ARMAEstimateMissing(int[] tpoints, double[] z)
```

Description

Constructor for `ARMAEstimateMissing`.

Parameters

`tpoints` – an `int` array containing the times at which the series values were observed. The values must be strictly increasing. Times for missing values are identified as non-incremental gaps in this series. A gap of missing values in `z` is assumed when the difference between two consecutive values is greater than 1, i.e. $t_{i+1} - t_i > 1$. The difference is the number of missing values in the gap. The series can have multiple gaps with missing values, but any one gap can have no more than 3 missing values.

`z` – a `double` array containing the values for the time series observed at the times given in the vector `tpoints`.

Methods

getCompleteTimeSeries

```
public double[] getCompleteTimeSeries() throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns a `double` precision vector of length `tpoints[tpoints.length-1]-tpoints[0]+1` containing the observed values in the time series `z` plus estimates for missing values in gaps identified in `tpoints`.

Returns

A `double` array of length `tpoints[tpoints.length-1]-tpoints[0]+1` containing the observed values in the time series `z` plus estimates for missing values in gaps identified in `tpoints`.

Exceptions

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input matrix to ARAutoUnivariate is singular. This can only occur with estimation method `AR_P`.

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertable. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

getCompleteTimes

`public int[] getCompleteTimes()`

Description

Returns an `int` array of all time points, including values for times with missing values in `z`.

Returns

An `int` array of all times from `tpoints[0]=1` to `tpoints.length+nMissing`. Where `nMissing` is the number of values removed from the original time series, `nMissing = getNumberMissing()`.

getConvergenceTolerance

`public double getConvergenceTolerance()`

Description

Returns the current value of convergence tolerance used by the `AR_1` and `AR_P` estimation methods.

Returns

a `double` scalar value equal to the convergence tolerance. By default the convergence tolerance is 1.0e-09.

getEstimationMethod

`public int getEstimationMethod()`

Description

Returns the method used for estimating the final autoregressive coefficients for missing value estimation methods `AR_1` and `AR_P`.

Returns

an `int` representing the estimation method used for estimating the autoregressive coefficients. 0 implies METHOD_OF_MOMENTS, 1 implies LEAST_SQUARES, 2 implies MAX_LIKELIHOOD.

getMaxIterations

`public int getMaxIterations()`

Description

Returns the maximum number of estimation iterations used by missing value estimation methods `AR_1` and `AR_P`.

Returns

An `int` scalar equal to the maximum number of iterations for the maximum likelihood missing value estimation method. If this limit is exceeded during the `compute` method, ARMAEstimateMissing stops execution and issues an `ARMAMaxLikelihood.IterationLimitExceededException`.

getMaxlag

```
public int getMaxlag()
```

Description

Returns the current value of autoregressive lags used in the `AR_P` estimation method.

Returns

An `int` scalar value equal to the maximum number of autoregressive lags used with the `AR_P` missing value estimation method.

getMean

```
public double getMean()
```

Description

Returns the mean value used to center the series.

Returns

a `double` scalar used to center the series.

getMissingTimes

```
public int[] getMissingTimes()
```

Description

Returns an `int` array of the times with missing values.

Returns

An `int` array containing the times at which missing values were estimated. If there are no missing values a `null` array is returned.

getMissingValueMethod

```
public int getMissingValueMethod()
```

Description

Returns the current missing value estimation method.

Returns

an `int` representing the estimation method used for estimating the missing values in the time series. 0 implies `MEDIAN`, 1 implies `CUBIC_SPLINE`, 2 implies `AR_1` and 3 implies `AR_P`.

getNumberMissing

```
public int getNumberMissing()
```

Description

Returns the number of missing values in the original series

Returns

An int scalar value containing the number of missing values in the time series.

getRelativeError

```
public double getRelativeError()
```

Description

Returns the relative error used for the METHOD_OF_MOMENTS and LEAST_SQARES estimation methods.

Returns

a double scalar containing the stopping criterion for use in the nonlinear equation solver used in both the method of moments and least-squares algorithm.

setConvergenceTolerance

```
public void setConvergenceTolerance(double convergenceTolerance)
```

Description

Sets the covergence tolerance used by the AR_1 and AR_P missing value estimation methods.

Parameter

convergenceTolerance – A double scalar value. Default: convergenceTolerance = 1.0e-09

setEstimationMethod

```
public void setEstimationMethod(int arEstimationMethod)
```

Description

Sets the method used for estimating the autoregressive coefficients for missing value estimation methods AR_1 and AR_P.

Parameter

arEstimationMethod – An int scalar specifying the method used to estimate the autoregressive coefficients. Valid methods are METHOD_OF_MOMENTS, LEAST_SQUARES, and MAX_LIKELIHOOD. By default, arEstimationMethod=LEAST_SQUARES.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Sets the maximum number of estimation iterations for missing value estimation methods AR_1 and AR_P. If this limit is exceeded ARMAEstimateMissing stops execution during the compute method and issues an IterationLimitExceededException.

Parameter

maxIterations – An int specifying the maximum number of iterations for the maximum likelihood estimation. By default, maxIterations=200.

setMaxlag

```
public void setMaxlag(int maxlag)
```

Description

Sets the maximum number of autoregressive lags when method `AR_P` is selected as the missing value estimation method.

Parameter

`maxlag` – An `int` scalar value equal to the maximum number of autoregressive lags. `maxlag` must be greater than `z.length-5`. By default `maxlag=10`.

setMean

`public void setMean(double mean)`

Description

Sets the mean value used to center the series.

Parameter

`mean` – a `double` scalar used to center the series. By default the median of the series is used for centering.

setMissingValueMethod

`public void setMissingValueMethod(int method)`

Description

Sets the current missing value estimation method to `MEDIAN, CUBIC_SPLINE, AR_1,` or `AR_P`.

Parameter

`method` – An `int` scalar. By default `method=AR_1`.

setRelativeError

`public void setRelativeError(double relativeError)`

Description

Sets the relative error used for the `METHOD_OF_MOMENTS` and `LEAST_SQUARES` estimation methods.

Parameter

`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 algorithm. Default: `relativeError` = 2.22045e-14

Example: ARMAEstimateMissing

The data in this example was artificially generated using an autoregressive time series with a lag of 1, i.e., AR(1). The constant term in the model was set to zero and -0.7 was used for the autoregressive coefficient. The data were generated from a random gaussian distribution with a mean of zero and an innovation variance of 0.51. This series is stationary with Var(Y) = 1.0.

Two hundred values were generated. For this example, six values at times t=130, t=140, t=141, t=160, t=175, and t=176 are removed and designated as missing. `ARMAEstimateMissing` is used to estimate these missing values using each of its estimation methods. The missing value estimates are compared to the actual values generated in the full series.

As expected, the AR(1) method produced the best missing value estimates in this example, closely followed by the AR(p) method.

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

public class ARMAEstimateMissingEx1 {

    public static void main(String args[]) throws Exception {
        int i, j, k;
        int maxlag = 20, n_obs = 194, n_miss = 6;
        int[] missing_index, tpointsMiss, tpointsComplete;
        double missVar = 0;
        double sum = 0;
        double variance = 0;
        double[] y, yMiss;
        double[] yComplete = {
            1.30540, -1.37166, 1.47905, -0.91059, 1.36191, -2.16966, 3.11254,
            -1.99536, 2.29740, -1.82474, -0.25445, 0.33519, -0.25480, -0.50574,
            -0.21429, -0.45932, -0.63813, 0.25646, -0.46243, -0.44104, 0.42733,
            0.61102, -0.82417, 1.48537, -1.57733, -0.09846, 0.46311, 0.49156,
            -1.66090, 2.02808, -1.45768, 1.36115, -0.65973, 1.13332, -0.86285,
            1.23848, -0.57301, -0.28210, 0.20195, 0.06981, 0.28454, 0.19745,
            -0.16490, -1.05019, 0.78652, -0.40447, 0.71514, -0.90003, 1.83604,
            -2.51205, 1.00526, -1.01683, 1.70691, -1.86564, 1.84912, -1.33120,
            2.35105, -0.45579, -0.57773, -0.55226, 0.88371, 0.23138, 0.59984,
            0.31971, 0.59849, 0.41873, -0.46955, 0.53003, -1.17203, 1.52937,
            -0.48017, -0.93830, 1.00651, -1.41493, -0.42188, -0.67010, 0.58079,
            -0.96193, 0.22763, -0.92214, 1.35697, -1.47008, 2.47841, -1.50522,
            0.41650, -0.21669, -0.90297, 0.00274, -1.04863, 0.66192, -0.39143,
            0.40779, -0.68174, -0.04700, -0.84469, 0.30735, -0.68412, 0.25888,
            -1.08642, 0.52928, 0.72168, -0.18199, -0.09499, 0.67610, 0.14636,
            0.46846, -0.13989, 0.50856, -0.22268, 0.92756, 0.73069, 0.78998,
            -1.01650, 1.25637, -2.36179, 1.99616, -1.54326, 1.38220, 0.19674,
            -0.85241, 0.40463, 0.39523, -0.60721, 0.25041, -1.24967, 0.26727,
            1.40042, -0.66963, 1.26049, -0.92074, 0.05909, -0.61926, 1.41550,
            0.25537, -0.13240, -0.07543, 0.10413, 1.42445, -1.37379, 0.44382,
            -1.57210, 2.04702, -2.22450, 1.27698, 0.01073, -0.88459, 0.88194,
            -0.25019, 0.70224, -0.41855, 0.93850, 0.36007, -0.46043, 0.18645,
            0.06337, 0.29414, -0.20054, 0.83078, -1.62530, 2.64925, -1.25355,
            1.59094, -1.00684, 1.03196, -1.58045, 2.04295, -2.38264, 1.65095,
            -0.33273, -1.29092, 0.14020, -0.11434, 0.04392, 0.05293, -0.42277,
            0.59143, -0.03347, -0.58457, 0.87030, 0.19985, -0.73500, 0.73640,
            0.29531, 0.22325, -0.60035, 1.42253, -1.11278, 1.30468, -0.41923,
            -0.38019, 0.50937, 0.23051, 0.46496, 0.02459, -0.68478, 0.25821,
            1.17655, -2.26629, 1.41173, -0.68331
        };
        ARMAEstimateMissing estMiss;
        ARAutoUnivariate arAuto;
        String title = " ";
        String[] colLabels = {"TIME", "ACTUAL", "PREDICTED", "DIFFERENCE"};
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        PrintMatrix pm = new PrintMatrix();
```

```
NumberFormat nf = NumberFormat.getNumberInstance();
nf.setMinimumFractionDigits(3);
nf.setMaximumFractionDigits(3);
pmf.setNumberFormat(nf);
pmf.setColumnLabels(colLabels);
pmf.setFirstRowNumber(1);
pm.setColumnSpacing(3);

/* setup missing data arrays */
tpointsComplete = new int[200];
tpointsMiss = new int[194];
yMiss = new double[194];
for (i = 1; i <= 200; i++) {
    tpointsComplete[i - 1] = i;
}
tpointsMiss[0] = tpointsComplete[0];
yMiss[0] = yComplete[0];
k = 0;
for (i = 1; i < 200; i++) {
    /* Generate series with missing values */
    if (i != 129 && i != 139 && i != 140 && i != 159
            && i != 174 && i != 175) {
        k += 1;
        tpointsMiss[k] = tpointsComplete[i];
        yMiss[k] = yComplete[i];
    }
}
n_obs = k + 1;

for (j = 0; j <= 3; j++) {
    estMiss = new ARMAEstimateMissing(tpointsMiss, yMiss);
    switch (j) {
        case ARMAEstimateMissing.MEDIAN:
            estMiss.setMissingValueMethod(ARMAEstimateMissing.MEDIAN);
            title = "MEDIAN ESTIMATES";
            break;
        case ARMAEstimateMissing.CUBIC_SPLINE:
            estMiss.setMissingValueMethod(
                    ARMAEstimateMissing.CUBIC_SPLINE);
            title = "CUBIC SPLINE ESTIMATES";
            break;
        case ARMAEstimateMissing.AR_1:
            estMiss.setMissingValueMethod(ARMAEstimateMissing.AR_1);
            title = "AR(1) ESTIMATES";
            break;
        case ARMAEstimateMissing.AR_P:
            estMiss.setMaxlag(maxlag);
            estMiss.setMissingValueMethod(ARMAEstimateMissing.AR_P);
            estMiss.setEstimationMethod(
                    ARMAEstimateMissing.METHOD_OF_MOMENTS);
            title = "AR(P) ESTIMATES";
            break;
    }
    WarningObject currentWarningLevel = Warning.getWarning();
    /* For some data it is useful to turn off warnings produced by
     * the ARMA estimation process.  This is only necessary for
```

```
            * the AR_1 and AR_P estimation methods
            */
           Warning.setWarning(null);                // turn off warnings
           y = estMiss.getCompleteTimeSeries();
           Warning.setWarning(currentWarningLevel); // turn on warnings
           missing_index = estMiss.getMissingTimes();
           n_miss = y.length - yMiss.length;
           double[][] printOutput = new double[n_miss][4];
           sum = 0;
           for (i = 0; i < n_miss; i++) {
               k = missing_index[i];
               printOutput[i][0] = tpointsComplete[k];
               printOutput[i][1] = yComplete[k];
               printOutput[i][2] = y[k];
               printOutput[i][3] = Math.abs(yComplete[k] - y[k]);
               sum += Math.pow(printOutput[i][3], 2);
           }
           arAuto = new ARAutoUnivariate(maxlag, y);
           arAuto.compute();
           variance = arAuto.getInnovationVariance();
           pm.setTitle(title);
           pm.print(pmf, printOutput);
           missVar = sum / n_miss;
           System.out.println(
                   "Innovation Variance Analysis - Estimate (percent of actual)");
           System.out.println("         Missing Values(only): " + missVar
                   + " (" + Math.round(100.0 * missVar / 0.51) + "%)");
           System.out.println("         Entire  Series:       " + variance
                   + " (" + Math.round(100.0 * variance / 0.51) + "%)");
           System.out.println(
                   "***********************************************************");
           System.out.println("");
       }
   }
}
```

Output

```
               MEDIAN ESTIMATES
     TIME    ACTUAL   PREDICTED   DIFFERENCE
1   130.000   -0.921     0.261       1.182
2   140.000    0.444     0.057       0.386
3   141.000   -1.572     0.057       1.630
4   160.000    2.649     0.047       2.602
5   175.000   -0.423     0.048       0.471
6   176.000    0.591     0.048       0.543

Innovation Variance Analysis - Estimate (percent of actual)
         Missing Values(only): 1.9152593761916663 (376%)
         Entire  Series:       0.5350398415037567 (105%)
***********************************************************

            CUBIC SPLINE ESTIMATES
     TIME    ACTUAL   PREDICTED   DIFFERENCE
1   130.000   -0.921     1.541       2.462
```

```
2   140.000    0.444    -0.407       0.851
3   141.000   -1.572     2.497       4.069
4   160.000    2.649    -2.947       5.596
5   175.000   -0.423     0.251       0.673
6   176.000    0.591     0.380       0.211

Innovation Variance Analysis - Estimate (percent of actual)
         Missing Values(only): 9.193464593399945 (1803%)
         Entire  Series:       0.7591379904132599 (149%)
***********************************************************

                AR(1) ESTIMATES
      TIME    ACTUAL   PREDICTED   DIFFERENCE
1   130.000   -0.921    -0.930       0.009
2   140.000    0.444     1.028       0.584
3   141.000   -1.572    -0.745       0.827
4   160.000    2.649     1.229       1.420
5   175.000   -0.423     0.010       0.433
6   176.000    0.591     0.037       0.555

Innovation Variance Analysis - Estimate (percent of actual)
         Missing Values(only): 0.5897529211558102 (116%)
         Entire  Series:       0.5013106660252865 (98%)
***********************************************************

                AR(P) ESTIMATES
      TIME    ACTUAL   PREDICTED   DIFFERENCE
1   130.000   -0.921    -0.889       0.032
2   140.000    0.444     1.009       0.565
3   141.000   -1.572    -0.688       0.884
4   160.000    2.649     1.210       1.439
5   175.000   -0.423    -0.002       0.421
6   176.000    0.591     0.038       0.553

Innovation Variance Analysis - Estimate (percent of actual)
         Missing Values(only): 0.6091513658345841 (119%)
         Entire  Series:       0.5017903704689042 (98%)
***********************************************************
```

ARMAMaxLikelihood class

`public class com.imsl.stat.ARMAMaxLikelihood implements Serializable`

Computes maximum likelihood estimates of parameters for an ARMA model with p and q autoregressive and moving average terms respectively.

`ARMAMaxLikelihood` 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`. The class is derived from the

maximum likelihood estimation algorithm described by Akaike, Kitagawa, Arahata and Tada (1979), and the XSARMA routine published in the TIMSAC-78 Library.

The stationary time series W_t with mean μ can be represented by the nonseasonal autoregressive moving average (ARMA) model by the following relationship:

$$\phi(B)(W_t - \mu) = \theta(B)a_t$$

where

$$t \in Z, \;\; Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\},$$

B is the backward shift operator defined by $B^k W_t = W_{t-k}$,

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p, \; p \geq 0.$$

and

$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q, \; q \geq 0.$$

The `ARMAMaxLikelihood` class estimates the AR coefficients $\phi_1, \phi_2, \ldots, \phi_p$ and the MA coefficients $\theta_1, \theta_2, \ldots, \theta_p$ using maximum likelihood estimation.

`ARMAMaxLikelihood` checks the initial estimates for both the autoregressive and moving average coefficients to ensure that they represent a stationary and invertible series respectively.

If

$$\phi_1, \phi_2, \ldots, \phi_p$$

are the initial estimates for a stationary series then all (complex) roots of the following polynomial will fall outside the unit circle:

$$1 - \phi_1 z - \phi_2 z^2 - \ldots - \phi_p z^p.$$

If

$$\theta_1, \theta_2, \ldots, \theta_p$$

are initial estimates for an invertible series then all (complex) roots of the polynomial

$$1 - \theta_1 z - \theta_2 z^2 - \ldots - \theta_q z^q$$

will fall outside the unit circle.

By default, the order of the lags for the autoregressive terms is $1, 2, \ldots, p$ and $1, 2, \ldots, q$ for the moving average terms. However, this cannot be overridden.

Initial values for the `AR` and `MA` coefficients can be supplied via the `setAR` and `setMA` methods. Otherwise, initial estimates are computed internally by the method of moments. The class computes the roots of the associated polynomials. If the `AR` estimates represent a nonstationary series, `ARMAMaxLikelihood` issues a warning message and replaces the intial `AR` estimates with initial values that are stationary. If the `MA` estimates represent a noninvertible series, a terminal error is issued and new initial values must be sought.

`ARMAMaxLikelihood` also validates the final estimates of the `AR` coefficients to ensure that they too represent a stationary series. This is done to guard against the possibility that the internal log-likelihood

optimizer converged to a nonstationary solution. If nonstationary estimates are encountered, a fatal error message is issued.

For model selection, the `ARMA` model with the minimum value for AIC might be preferred, $\text{AIC} = -2\ln(L) + 2(p+q)$, where L is the value of the maximum likelihood function evaluated at the parameter estimates.

`ARMAMaxLikelihood` can also handle white noise processes, i.e. $ARMA(0,0)$ processes.

Constructor

ARMAMaxLikelihood

```
public ARMAMaxLikelihood(int p, int q, double[] z) throws
ARMA.MatrixSingularException, ARMA.TooManyCallsException,
ARMA.IncreaseErrRelException, ARMA.NewInitialGuessException,
ARMA.IllConditionedException, ARMA.TooManyITNException,
ARMA.TooManyFcnEvalException, ARMA.TooManyJacobianEvalException
```

Description

Constructor for `ARMAMaxLikelihood`.

Parameters

`p` – An `int` scalar equal to the number of autoregressive (AR) parameters.

`q` – An `int` scalar equal to the number of moving average (MA) parameters.

`z` – A `double` array containing the time series.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

Methods

compute

```
public void compute() throws ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Computes the exact maximum likelihood estimates for the autoregressive and moving average parameters of an ARMA time series

Exceptions

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

forecast

```
public double[][] forecast(int nForecast) throws
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException, ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException
```

Description

Returns forecasts for lead times $l = 1, 2, \ldots, \mathrm{nForecast}$ at origins `z.length-backwardOrigin-1+j` where $j = 1, \ldots, \mathrm{backwardOrigin} + 1$.

Parameter

`nForecast` – An `int` scalar equal to the number of requested forecasts

Returns

a `double` matrix of dimensions of `nForecast` by `backwardOrigin + 1` containing the forecasts. The forecasts are for lead times $l = 1, 2, \ldots, \mathrm{nForecast}$ at origins `z.length-backwardOrigin-1+j` where $j = 1, \ldots, \mathrm{backwardOrigin} + 1$.

Exceptions

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

getAR

```
public double[] getAR()
```

Description

Returns the final autoregressive parameter estimates.

Returns

A `double` array of length `p` containing the final autoregressive parameter estimates.

getBackwardOrigin

```
public int getBackwardOrigin()
```

Description

Returns the current value for forecasting backward origin.

Returns

A `double` scalar value representing the current value of `backwardOrigin`

getConfidence

```
public double getConfidence()
```

Description

Returns the confidence level used for calculating deviations in `getDeviations`.

Returns

A `double` scalar value of `confidence` used for computing the `(1-confidence)*100%` forecast confidence interval.

getConstant

```
public double getConstant()
```

Description

Returns the estimate for the constant parameter in the ARMA series.

Returns

A `double` scalar equal to the estimate for the constant parameter in the ARMA series.

getDeviations

```
public double[] getDeviations()
```

Description

Returns the deviations from each forecast used for calculating the forecast confidence limits.

Returns

A `double` array of length `backwardOrigin+nForecast` containing the deviations for calculating forecast confidence intervals. The confidence level is specified in `confidence`. By default, `confidence`= 0.95.

getForecast

```
public double[] getForecast(int nForecast) throws
ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException, ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException
```

Description

Returns forecasts

Parameter

`nForecast` – An input `int` representing the number of requested forecasts beyond the last value in the series.

Returns

A `double` array containing the `nForecast+backwardOrigin` forecasts. The first `backwardOrigin` forecasts are one-step ahead forecasts for the last `backwardOrigin` values in the series. The next `nForecast` values in the returned series are forecasts for the next values beyond the series.

Exceptions

`ARMAMaxLikelihood.NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

getGradientTolerance

```
public double getGradientTolerance()
```

Description

Returns the gradient tolerance for the convergence algorithm.

Returns

A `double` scalar specifying the value used for the gradient tolerance.

getGradients

```
public double[] getGradients()
```

Description

Returns the gradients for the final parameter estimates.

Returns

A `double` array of length `p+q` containing the gradients of the final parameter estimates.

getInnovationVariance

```
public double getInnovationVariance()
```

Description

Returns the estimated innovation variance of this series.

Returns

A `double` scalar equal to the estimated innovation variance for the time series.

getLikelihood

```
public double getLikelihood() throws ARMAMaxLikelihood.NonStationaryException,
ARMAMaxLikelihood.NonInvertibleException
```

Description

Returns the final estimate for $-2\ln(L)$, where L is equal to the likelihood function evaluated using the final parameter estimates.

Returns

A `double` scalar equal to the log likelihood function, $-2\ln(L)$.

Exceptions

`NonStationaryException` is thrown if the final maximum likelihood estimates for the time series are nonstationary.

`NonInvertibleException` is thrown if the final maximum likelihood estimates for the time series are noninvertible.

`InitialMAException` is thrown if the initial values provided for the moving average terms using `setMA` are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

---

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.

---

getMaxIterations

```
public int getMaxIterations()
```

Description

Returns the maximum number of iterations.

Returns

An `int` scalar containing the maximum number of iterations allowed in the nonlinear equation solver used in both the method of moments and least-squares algorithms. Default: `maxIterations` = 300.

---

getMean

```
public double getMean()
```

Description

Returns the mean used to center the time series.

Returns

A `double` scalar specifying the value to used for centering the time series.

---

getP

```
public int getP()
```

Description

Returns the number of autoregressive terms in the ARMA model

Returns

An `int` scalar value of `p`, the number of autoregressive terms in the ARMA model.

---

getPsiWeights

```
public double[] getPsiWeights()
```

Description

Returns the psi weights used for calculating forecasts from the infinite order moving average form of the ARMA model.

Returns

A `double` array of length `nForecast` containing the psi weights of the infinite order moving average form of the model.

getQ

```
public int getQ()
```

Description

Returns the number of moving average terms in the ARMA model

Returns

An `int` scalar value of `q`, the number of moving average terms in the ARMA model.

getResiduals

```
public double[] getResiduals()
```

Description

Returns the current values of the vector of residuals.

Returns

A `double` array of length `backwardOrigin` containing the residuals for the last `backwardOrigin` values in the time series. The `compute` and either the `forecast` or `getForecast` methods must be called before calling this method.

getTimeSeries

```
public double[] getTimeSeries()
```

Description

Returns the time series used to construct `ARMAMaxLikelihood`.

Returns

A `double` containing the values of the time series passed to the class constructor.

getTolerance

```
public double getTolerance()
```

Description

Returns the tolerance for the convergence algorithm.

Returns

A `double` scalar containing the value of the tolerance used during maximum likelihood estimation.

isInvertible

```
public boolean isInvertible(double[] ma)
```

Description

Tests whether the coefficients in `ma` are invertible

Parameter

`ma` – A `double` array containing the coefficients for the moving average terms in an ARMA model.

Returns

A `boolean` scalar equal to `true` if the coefficients in `ma` are invertible and `false` otherwise.

isStationary

`public boolean isStationary(double[] ar)`

Description

Tests whether the coefficients in `ar` are stationary.

Parameter

`ar` – A `double` array containing the coefficients for the autoregressive terms in an ARMA model

Returns

A `boolean` scalar equal to `true` if the coefficients in `ar` are stationary and `false` otherwise.

setAR

`public void setAR(double[] ar)`

Description

Sets the initial values for the autoregressive terms to the `p` values in `ar`.

Parameter

`ar` – An input `double` array of length `p` containing the initial values for the autoregressive terms. If this method is not called, initial values are computed by method of moments in the `ARMA` class.

setBackwardOrigin

`public void setBackwardOrigin(int backwardOrigin)`

Description

Sets the maximum backward origin.

Parameter

`backwardOrigin` – An `int` scalar specifying the maximum backward origin used in forecasting. `backwardOrigin` must be greater than or equal to 0 and less than or equal to `z.length - Math.max(p, q)`. Default: `backwardOrigin = 0`.

setConfidence

`public void setConfidence(double confidence)`

Description

Sets the confidence level for calculating confidence limit deviations returned from `getDeviations()`.

Parameter

`confidence` – a `double` scalar specifying the confidence level used in computing forecast confidence intervals. 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`.

setConstant

```
public void setConstant(double constant)
```

Description

Sets the initial value for the constant term in the ARMA model.

Parameter

`constant` – A `double` scalar specifying the initial value for the constant term in the ARMA model. By default, the constant term is initially estimated using ARMA method of moments estimation.

setGradientTolerance

```
public void setGradientTolerance(double gradientTolerance)
```

Description

Sets the tolerance for the convergence algorithm.

Parameter

`gradientTolerance` – A `double` scalar specifying the the tolerance used for numerically estimating the gradient by differences. Default: `gradientTolerance = 1e-04`.

setMA

```
public void setMA(double[] ma)
```

Description

Sets the initial values for the moving average terms to the `q` values in `ma`.

Parameter

`ma` – A `double` array of length `q` containing the initial values for the moving average terms. If this method is not called, initial values are computed by method of moments in the `ARMA` class.

setMaxIterations

```
public void setMaxIterations(int maxIterations) throws IllegalArgumentException
```

Description

Sets the maximum number of iterations.

Parameter

`maxIterations` – 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: `maxIterations` = 300.

setMean

```
public void setMean(double wMean)
```

Description

Sets the mean used for centering the series.

Parameter

`wMean` – A `double` scalar specifying the value to use for centering the time series. By default the series is centered using the mean of the series.

setTolerance

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

Description

Sets the tolerance for the convergence algorithm.

Parameter

`tolerance` – A `double` scalar specifying the value to use for the convergence tolerance. Default: `tolerance = 2.220446049e-016`.

Example: ARMAMaxLikelihood

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 maximum likelihood 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

$$\sigma_A^2$$

The maximum likelihood estimates from `ARMAMaxLikelihood` are compared to the same estimates using the method of moments and least squares from the `ARMA` class. For each method, the coefficients and forecasts for the last ten years, 1860-1869, are compared. The method of moments and maximum likelihood estimates produced similar results, but the least squares estimates were very different from the other two.

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

public class ARMAMaxLikelihoodEx1 {

    public static void main(String args[]) throws Exception {
        int backwardOrigin = 0, n_forecast = 10, n_series = 100;
```

```
double[] sunspots = {
    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[] z;
double arMM[], maMM[], constantMM;
double arLS[], maLS[], constantLS;
double ar[], ma[], constant;
double forecastMM[][], forecastLS[][], forecast[][];
double deviations[];
double likelihood, var, varMM, varLS;
double[] avgDev = {0.0, 0.0, 0.0};
ARMA armaMM, armaLS;
ARMAMaxLikelihood maxArma;
double[][] printOutput, printOutput2;
String[] colLabels = {
    "Method of Moments", "Least Squares", "Maximum Likelihood"
};
String[] colLabels1 = {"Least Squares", "Maximum Likelihood"};
String[] colLabels2 = {
    "Observed Sunspots", "Method of Moments",
    "Least Squares", "Maximum Likelihood"
};
String[] colLabels3 = {
    "Lower Confidence Limit", "Forecast", "Upper Confidence Limit"
};
PrintMatrixFormat pmf = new PrintMatrixFormat();
PrintMatrix pm = new PrintMatrix();
NumberFormat nf = NumberFormat.getNumberInstance();
pm.setColumnSpacing(3);
nf.setMinimumFractionDigits(3);
pmf.setNoRowLabels();
pmf.setNumberFormat(nf);
pmf.setColumnLabels(colLabels);
printOutput = new double[1][3];
printOutput2 = new double[n_forecast][4];
z = new double[n_series];
for (int i = 0; i < n_series; i++) {
    z[i] = sunspots[i];
}
/* Method of Moments ARMA(2,1) Estimation */
armaMM = new ARMA(2, 1, z);
armaMM.setMethod(ARMA.METHOD_OF_MOMENTS);
armaMM.compute();
armaMM.setBackwardOrigin(backwardOrigin);
arMM = armaMM.getAR();
maMM = armaMM.getMA();
```

```
constantMM = armaMM.getConstant();
forecastMM = armaMM.forecast(n_forecast);
varMM = armaMM.getInnovationVariance();

/* Least Squares ARMA(2,1) Estimation */
armaLS = new ARMA(2, 1, z);
armaLS.setMethod(ARMA.LEAST_SQUARES);
armaLS.compute();
armaLS.setBackwardOrigin(backwardOrigin);
arLS = armaLS.getAR();
maLS = armaLS.getMA();
constantLS = armaLS.getConstant();
varLS = armaLS.getInnovationVariance();
forecastLS = armaLS.forecast(n_forecast);

/* Maximum Likelihood ARMA(2,1) Estimation */
maxArma = new ARMAMaxLikelihood(2, 1, z);
maxArma.compute();
maxArma.setBackwardOrigin(backwardOrigin);
ar = maxArma.getAR();
ma = maxArma.getMA();
constant = maxArma.getConstant();
likelihood = maxArma.getLikelihood();
var = maxArma.getInnovationVariance();
maxArma.setConfidence(0.9);
forecast = maxArma.forecast(n_forecast);
deviations = maxArma.getDeviations();

printOutput[0][0] = constantMM;
printOutput[0][1] = constantLS;
printOutput[0][2] = constant;
pm.setTitle("ARMA(2,1) - Constant Term");
pm.print(pmf, printOutput);
printOutput[0][0] = arMM[0];
printOutput[0][1] = arLS[0];
printOutput[0][2] = ar[0];
pm.setTitle("ARMA(2,1) - AR(1) Coefficient");
pm.print(pmf, printOutput);
printOutput[0][0] = arMM[1];
printOutput[0][1] = arLS[1];
printOutput[0][2] = ar[1];
pm.setTitle("ARMA(2,1) - AR(2) Coefficient");
pm.print(pmf, printOutput);
printOutput[0][0] = maMM[0];
printOutput[0][1] = maLS[0];
printOutput[0][2] = ma[0];
pm.setTitle("ARMA(2,1) - MA(1) Coefficient");
pm.print(pmf, printOutput);
System.out.println("INNOVATION VARIANCE:");
System.out.println("Method of Moments  " + varMM);
System.out.println("Least Squares      " + varLS);
System.out.println("Maximum Likelihood " + var);
System.out.println("");

for (int i = 0; i < n_forecast; i++) {
    printOutput2[i][0] = sunspots[90 + i];
```

```
            printOutput2[i][1] = forecastMM[i][backwardOrigin];
            printOutput2[i][2] = forecastLS[i][backwardOrigin];
            printOutput2[i][3] = forecast[i][backwardOrigin];
        }
        pm.setTitle("SUNSPOT FORECASTS FOR 1860-1869");
        nf.setMaximumFractionDigits(0);
        pmf.setNumberFormat(nf);
        pmf.setColumnLabels(colLabels2);
        pmf.setFirstRowNumber(1860);
        pm.print(pmf, printOutput2);
        /* Get Confidence Interval Deviations */
        printOutput2 = new double[n_forecast][3];
        for (int i = 0; i < n_forecast; i++) {
            printOutput2[i][0] = Math.max(0, forecast[i][backwardOrigin]
                    - deviations[i + backwardOrigin]);
            printOutput2[i][1] = forecast[i][backwardOrigin];
            printOutput2[i][2] = forecast[i][backwardOrigin] + deviations[i];
        }
        nf.setMaximumFractionDigits(0);
        pmf.setNumberFormat(nf);
        pmf.setColumnLabels(colLabels3);
        pmf.setFirstRowNumber(1860);
        pm.setTitle("SUNSPOT MAX. LIKELIHOOD 90% CONFIDENCE INTERVALS");
        pm.print(pmf, printOutput2);
    }
}
```

Output

```
                ARMA(2,1) - Constant Term
   Method of Moments   Least Squares   Maximum Likelihood
     15.544            17.932            15.758

              ARMA(2,1) - AR(1) Coefficient
   Method of Moments   Least Squares   Maximum Likelihood
      1.244             1.531            1.225

              ARMA(2,1) - AR(2) Coefficient
   Method of Moments   Least Squares   Maximum Likelihood
     -0.575            -0.894            -0.561

              ARMA(2,1) - MA(1) Coefficient
   Method of Moments   Least Squares   Maximum Likelihood
     -0.124            -0.132            -0.383

INNOVATION VARIANCE:
Method of Moments  287.2424037381216
Least Squares      239.68797223858988
Maximum Likelihood 214.5087884256287

                          SUNSPOT FORECASTS FOR 1860-1869
       Observed Sunspots   Method of Moments   Least Squares   Maximum Likelihood
1860          96                   88                98                88
1861          77                   82               102                82
1862          59                   67                86                67
```

```
1863           44                 52                 59                 52
1864           47                 42                 31                 42
1865           30                 37                 13                 38
1866           16                 38                 10                 39
1867            7                 41                 21                 42
1868           37                 45                 42                 45
1869           74                 48                 63                 48

         SUNSPOT MAX. LIKELIHOOD 90% CONFIDENCE INTERVALS
      Lower Confidence Limit   Forecast   Upper Confidence Limit
1860              64              88              112
1861              36              82              128
1862              10              67              124
1863               0              52              112
1864               0              42              102
1865               0              38               98
1866               0              39              100
1867               0              42              104
1868               0              45              107
1869               0              48              110
```

ARMAMaxLikelihood.NonInvertibleException class

static public class com.imsl.stat.ARMAMaxLikelihood.NonInvertibleException
extends com.imsl.IMSLException

The solution is noninvertible.

Constructors

ARMAMaxLikelihood.NonInvertibleException

public ARMAMaxLikelihood.NonInvertibleException(String message)

Description

Constructs a NonInvertible exception with the specified detail message. A detail message is a String exception that describes this particular exception.

Parameter

message – An input string containing the detail message.

ARMAMaxLikelihood.NonInvertibleException

public ARMAMaxLikelihood.NonInvertibleException(String key, Object[] arguments)

Description

Constructs a `NonInvertibleException` exception 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.

ARMAMaxLikelihood.NonStationaryException class

```
static public class com.imsl.stat.ARMAMaxLikelihood.NonStationaryException
extends com.imsl.IMSLException
```

The solution is nonstationary.

Constructors

ARMAMaxLikelihood.NonStationaryException

```
public ARMAMaxLikelihood.NonStationaryException(String message)
```

Description

Constructs a `NonStationary` exception with the specified detail message. A detail message is a `String` exception that describes this particular exception.

Parameter

`message` – An input string containing the detail message.

ARMAMaxLikelihood.NonStationaryException

```
public ARMAMaxLikelihood.NonStationaryException(String key, Object[] arguments)
```

Description

Constructs a `NonStationary` exception 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.

ARMAOutlierIdentification class

```
public class com.imsl.stat.ARMAOutlierIdentification implements Serializable,
Cloneable
```

Detects and determines outliers and simultaneously estimates the model parameters in a time series whose underlying outlier free series follows a general seasonal or nonseasonal ARMA model. This class also allows computation of forecasts.

Consider a univariate time series $\{Y_t\}$ that can be described by the following multiplicative seasonal ARIMA model of order $(p,0,q)\times(0,d,0)_s$:

$$Y_t - \mu = \frac{\theta(B)}{\Delta_s^d \phi(B)} a_t, t = 1,\ldots,n$$

Here, $\Delta_s^d = (1-B^s)^d$, $\theta(B) = 1-\theta_1 B - \ldots - \theta_q B^q$, $\phi(B) = 1-\phi_1 B - \ldots - \phi_p B^p$. B is the lag operator, $B^k Y_t = Y_{t-k}$, $\{a_t\}$ is a white noise process, and μ denotes the mean of the series $\{Y_t\}$.

Outlier detection and parameter estimation

In general, $\{Y_t\}$ is not directly observable due to the influence of outliers. Chen and Liu (1993) distinguish between four types of outliers: innovational outliers (IO), additive outliers (AO), temporary changes (TC) and level shifts (LS). If an outlier occurs as the last observation of the series, then Chen and Liu's algorithm is unable to determine the outlier's classification. In class `ARMAOutlierIdentification`, such an outlier is called a UI (unable to identify) and is treated as an innovational outlier.

In order to take the effects of multiple outliers occurring at time points $t_1,\ldots,t_m$ into account, Chen and Liu consider the following model:

$$Y_t^* - \mu = \sum_{j=1}^{m} \omega_j L_j(B) I_t(t_j) + \frac{\theta(B)}{\Delta_s^d \phi(B)} a_t$$

Here, $\{Y_t^*\}$ is the observed outlier contaminated series, and ω_j and $L_j(B)$ denote the magnitude and dynamic pattern of outlier j, respectively. $I_t(t_j)$ is an indicator function that determines the temporal course of the outlier effect, $I_{t_j}(t_j) = 1, I_t(t_j) = 0$ otherwise. Note that $L_j(B)$ operates on I_t via $B^k I_t = I_{t-k}, k = 0,1,\ldots$.

The last formula shows that the outlier free series $\{Y_t\}$ can be obtained from the original series $\{Y_t^*\}$ by removing all occuring outlier effects:

$$Y_t = Y_t^* - \sum_{j=1}^{m} \omega_j L_j(B) I_t(t_j)$$

The different types of outliers are charaterized by different values for $L_j(B)$:

1. $L_j(B) = \frac{\theta(B)}{\Delta_s^d \phi(B)}$ for an innovational outlier,

2. $L_j(B) = 1$ for an additive outlier,

3. $L_j(B) = (1-B)^{-1}$ for a level shift outlier and

4. $L_j(B) = (1-\delta B)^{-1}, 0 < \delta < 1$, for a temporary change outlier.

Class `ARMAOutlierIdentification` is an implementation of Chen and Liu's algorithm. It determines the coefficients in $\phi(B)$ and $\theta(B)$ and the outlier effects in the model for the observed series jointly in three stages. The magnitude of the outlier effects is determined by least squares estimates. Outlier detection itself is realized by examination of the maximum value of the standardized statistics of the outlier effects. For a detailed description, see Chen and Liu's original paper (1993).

Intermediate and final estimates for the coefficients in $\phi(B)$ and $\theta(B)$ are computed by the `compute` methods from JMSL classes `ARMA` and `ARMAMaxLikelihood`. If the roots of $\phi(B)$ or $\theta(B)$ lie on or within the unit circle, then the algorithm stops with an appropriate exception. In this case, different values for p and q should be tried.

Forecasting

From the relation between original and outlier free series,

$$Y_t^* = Y_t + \sum_{j=1}^{m} \omega_j L_j(B) I_t(t_j)$$

it follows that the Box-Jenkins forecast at origin t for lead time l, $\hat{Y}_t^*(l)$, can be computed as

$$\hat{Y}_t^*(l) = \hat{Y}_t(l) + \sum_{j=1}^{m} \omega_j L_j(B) I_{t+l}(t_j),\ l = 1, \ldots, \text{nForecast}$$

Therefore, computation of the forecasts for $\{Y_t^*\}$ is done in two steps:

1. Computation of the forecasts for the outlier free series $\{Y_t\}$.

2. Computation of the forecasts for the original series $\{Y_t^*\}$ by adding the multiple outlier effects to the forecasts for $\{Y_t\}$.

Step 1: Computation of the forecasts for the outlier free series $\{Y_t\}$

Since

$$\varphi(B)(Y_t - \mu) = \theta(B) a_t$$

where

$$\varphi(B) := \Delta_s^d \phi(B) = 1 - \varphi_1 B - \ldots - \varphi_{p+sd} B^{p+sd}$$

the Box-Jenkins forecast at origin t for lead time l, $\hat{Y}_t(l)$, can be computed recursively as

$$\hat{Y}_t(l) = (1 - \sum_{j=1}^{p+sd} \varphi_j)\mu + \sum_{j=1}^{p+sd} \varphi_j \hat{Y}_t(l-j) - \sum_{j=1}^{q} \theta_j a_{t+l-j}$$

Here,

$$\hat{Y}_t(l-j) = \begin{cases} Y_{t+l-j} & \text{for} \quad l-j \le 0 \\ \hat{Y}_t(l-j) & \text{for} \quad l-j > 0 \end{cases}$$

and

$$a_k = \begin{cases} 0 & \text{for} \quad k \le \max\{1, p+sd\} \\ Y_k - \hat{Y}_{k-1}(1) & \text{for} \quad k = \max\{1, p+sd\}+1, \ldots, n \end{cases}$$

Step 2: Computation of the forecasts for the original series $\{Y_t{}^*\}$ by adding the multiple outlier effects to the forecasts for $\{Y_t\}$

The formulas for $L_j(B)$ for the different types of outliers are as follows:

Innovational outlier (IO)

$$L_j(B) = \frac{\theta(B)}{\Delta_s^d \phi(B)} := \psi(B) = \sum_{k=0}^{\infty} \psi_k B^k,\ \psi_0 = 1$$

Additive outliers (AO)

$$L_j(B) = 1$$

Level shifts (LS)

$$L_j(B) = \frac{1}{1-B} = \sum_{k=0}^{\infty} B^k$$

Temporary changes (TC)

$$L_j(B) = \frac{1}{1-\delta B} = \sum_{k=0}^{\infty} \delta^k B^k$$

Assuming the outlier occurs at time point t_j, the outlier impact is therefore:

Innovational outliers (IO)

$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t < t_j \\ \omega_j \psi_k & \text{for} \quad t = t_j + k,\ k \ge 0 \end{cases}$$

Additive outliers (AO)

$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t \ne t_j \\ \omega_j & \text{for} \quad t = t_j \end{cases}$$

Level shifts (LS)

$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t < t_j \\ \omega_j & \text{for} \quad t = t_j + k,\ k \ge 0 \end{cases}$$

Temporary changes (TC)

$$\omega_j L_j(B) I_t(t_j) = \begin{cases} 0 & \text{for} \quad t < t_j \\ \omega_j \delta^k & \text{for} \quad t = t_j + k,\ k \ge 0 \end{cases}$$

From these formulas, the forecasts $\hat{Y}_t^*(l)$ can be computed easily. The $100(1-\alpha)$ percent probability limits for Y_{t+l}^* and Y_{t+l} are given by

$$\hat{Y}_t^*(l)\,(\text{or } \hat{Y}_t(l), \text{resp.}) \pm u_{\alpha/2}(1 + \sum_{j=1}^{l-1} \psi_j^2)^{1/2} s_a$$

where $u_{\alpha/2}$ is the $100(1-\alpha/2)$ percentile of the standard normal distribution, s_a^2 is an estimate of the variance σ_a^2 of the random shocks, and the ψ weights $\{\psi_j\}$ are the coefficients in

$$\psi(B) := \sum_{k=0}^{\infty} \psi_k B^k := \frac{\theta(B)}{\Delta_s^d \phi(B)},\ \psi_0 = 1\,.$$

For a detailed explanation of these concepts, see chapter 5:"Forecasting" in Box, Jenkins and Reinsel (1994).

Fields

ADDITIVE

```
static final public int ADDITIVE
```

Indicates detection of an additive outlier.

INNOVATIONAL

```
static final public int INNOVATIONAL
```

Indicates detection of an innovational outlier.

LEVEL_SHIFT

```
static final public int LEVEL_SHIFT
```

Indicates detection of a level shift outlier.

TEMPORARY_CHANGE

```
static final public int TEMPORARY_CHANGE
```

Indicates detection of a temporary change outlier.

UNABLE_TO_IDENTIFY

```
static final public int UNABLE_TO_IDENTIFY
```

Indicates detection of an outlier that cannnot be categorized.

Constructor

ARMAOutlierIdentification

```
public ARMAOutlierIdentification(double[] z)
```

Description

Constructor for `ARMAOutlierIdentification`.

Parameter

`z` – a `double` array containing the observations.

Methods

compute

```
final public void compute(int[] model) throws
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException,
ZeroPolynomial.DidNotConvergeException, ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException, SingularMatrixException,
Cholesky.NotSPDException
```

Description

Detects and determines outliers and simultaneously estimates the model parameters for the given time series.

Parameter

`model` – an `int` array of length 4 containing the numbers p, q, s, d of the ARIMA$(p,0,q) \times (0,d,0)_s$ model the outlier free series is following. It is required that p, q and d are non-negative and s is positive and consistent with `z.length`.

Exceptions

`ARMAMaxLikelihood.NonStationaryException` is thrown if the intermediate or final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the intermediate or final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ZeroPolynomial.DidNotConvergeException` is thrown if the algorithm computing the roots of the AR- or MA- polynomial does not converge.

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`SingularMatrixException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is singular.

`Cholesky.NotSPDException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is not positive definite.

computeForecasts

`final public void computeForecasts(int nForecast)`

Description

Computes forecasts, associated probability limits and ψ weights for an outlier contaminated time series whose underlying outlier free series obeys a general seasonal or non-seasonal ARMA model.

Parameter

`nForecast` – an `int` scalar containing the maximum lead time for forecasts. `nForecast` must be greater than 0. Forecast origin is the time point of the last observed value in the time series, n. Forecasts are computed for lead times $1, 2, \ldots, \mathrm{nForecast}$, i.e. time points $\mathrm{n}+1, \mathrm{n}+2, \ldots, \mathrm{n}+\mathrm{nForecast}$. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

getAIC

`public double getAIC()`

Description

Returns Akaike's information criterion (AIC).

Returns

a `double` scalar containing Akaike's information criterion (AIC) for the outlier free series. The `compute` method must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getAICC

`public double getAICC()`

Description

Returns Akaike's Corrected Information Criterion (AICC).

Returns

a `double` scalar containing Akaike's Corrected Information Criterion (AICC) for the outlier free series. The `compute` method must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getAR

`public double[] getAR()`

Description

Returns the final autoregressive parameter estimates.

Returns

a double array of length p = model[0] containing the final autoregressive parameter estimates. Note that the compute method must be invoked first before invoking this method. Otherwise, the method throws an IllegalStateException exception.

getBIC

```
public double getBIC()
```

Description

Returns the Bayesian Information Criterion (BIC).

Returns

a double scalar containing the Bayesian Information Criterion (BIC) for the outlier free series. The compute method must be called before invoking this method. Otherwise, an IllegalStateException exception is thrown.

getConstant

```
public double getConstant()
```

Description

Returns the constant parameter estimate.

Returns

a double scalar containing the constant parameter estimate. The compute method must be invoked first before invoking this method. Otherwise, the method throws an IllegalStateException exception.

getDeviations

```
public double[] getDeviations()
```

Description

Returns the deviations used for calculating the forecast confidence limits.

Returns

a double array of length nForecast containing the deviations from each forecast for calculating forecast confidence intervals. The confidence level is specified in setConfidence. Method computeForecasts has to be invoked before this method is called. Otherwise, an IllegalStateException exception is thrown.

getForecast

```
public double[] getForecast()
```

Description

Returns forecasts for the original outlier contaminated series.

Returns

a double array of length nForecast containing the forecasts for the original series. Method computeForecasts has to be invoked before this method is called. Otherwise, an IllegalStateException exception is thrown.

getMA

`public double[] getMA()`

Description

Returns the final moving average parameter estimates.

Returns

a `double` array of length q = `model[1]` containing the final moving average parameter estimates. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

getNumberOfOutliers

`public int getNumberOfOutliers()`

Description

Returns the number of outliers detected.

Returns

an `int` scalar containing the number of outliers detected. The `compute` method must be invoked first before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getOmegaWeights

`public double[] getOmegaWeights()`

Description

Returns the ω weights for the detected outliers.

Returns

a `double` array containing the computed ω weights for the detected outliers. If the number of detected outliers equals zero, then an array of length zero is returned. The `compute` method must be invoked before using this method. Otherwise, an `IllegalStateException` exception is thrown.

getOutlierFreeForecast

`public double[] getOutlierFreeForecast()`

Description

Returns forecasts for the outlier free series.

Returns

a `double` array of length `nForecast` containing the forecasts for the outlier free series. Method `computeForecasts` has to be invoked before this method is called. Otherwise, an `IllegalStateException` exception is thrown.

getOutlierFreeSeries

`public double[] getOutlierFreeSeries()`

Description

Returns the outlier free series.

Returns

a `double` array containing the outlier free series. The `compute` method must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getOutlierStatistics

```
public int[][] getOutlierStatistics()
```

Description

Returns the outlier statistics.

Returns

an `int` matrix of length `nOutliers` by 2, where `nOutliers` is the number of detected outliers, containing the outlier statistics. The first column contains the time at which the outlier was observed (time ranging from 1 to `z.length`, the number of observations in the time series) and the second column contains an identifier indicating the type of outlier observed. Outlier types fall into one of five categories:

Identifier	Outlier type
`INNOVATIONAL`	Innovational Outliers (IO)
`ADDITIVE`	Additive Outliers (AO)
`LEVEL_SHIFT`	Level Shift Outliers (LS)
`TEMPORARY_CHANGE`	Temporary Change Outliers (TC)
`UNABLE_TO_IDENTIFY`	Unable to Identify (UI)

If the number of detected outliers equals zero, then an `int` array of size 0 is returned.The `compute` method must be invoked first before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getPsiWeights

```
public double[] getPsiWeights()
```

Description

Returns the ψ weights of the infinite order moving average form of the model.

Returns

a `double` array of length `nForecast` containing the ψ weights of the infinite order moving average form of the model for the outlier free series. Method `computeForecasts` must be invoked before this method is called. Otherwise, an `IllegalStateException` exception is thrown.

getResidual

```
public double[] getResidual()
```

Description

Returns the residuals.

Returns

a `double` array containing the residuals for the outlier free series at the final parameter estimation point. The `compute` method must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

getResidualStandardError

```
public double getResidualStandardError()
```

Description

Returns the residual standard error of the outlier free series.

Returns

a `double` scalar containing the standard error of the outlier free series. Note that the `compute` method must be invoked first before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getTauStatistics

```
public double[] getTauStatistics()
```

Description

Returns the t value for each detected outlier.

Returns

a `double` array containing the t statistics for each detected outlier. If the number of detected outliers equals zero, then a vector of length 0 is returned. The `compute` method must be invoked before using this method. Otherwise, an `IllegalStateException` exception is thrown.

setAccuracyTolerance

```
public void setAccuracyTolerance(double epsilon)
```

Description

Sets the tolerance value controlling the accuracy of the parameter estimates.

Parameter

`epsilon` – a `double` scalar, a positive tolerance value controlling the accuracy of parameter estimates during outlier detection. Default: `epsilon = 0.001`.

setConfidence

```
public void setConfidence(double confidence)
```

Description

Sets the confidence level for calculating confidence limit deviations returned from `getDeviations`.

Parameter

`confidence` – a `double` scalar specifying the confidence level used in computing forecast confidence intervals. 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`.

setCriticalValue

```
public void setCriticalValue(double critical)
```

Description

Sets the critical value used as a threshold during outlier detection.

Parameter

`critical` – a `double` scalar, the critical value used as a threshold for the statistics used in the outlier detection. `critical` must be greater than zero. Default: `critical = 3.0`.

setDelta

`public void setDelta(double delta)`

Description

Sets the dampening effect parameter.

Parameter

`delta` – a `double` scalar, the dampening effect parameter used in the detection of a Temporary Change Outlier (TC). `delta` must be greater than 0 and less than 1. Default: `delta = 0.7`.

setRelativeError

`public void setRelativeError(double relativeError)`

Description

Sets the stopping criterion for use in the nonlinear equation solver.

Parameter

`relativeError` – a `double` positive scalar containing the stopping criterion for use in the nonlinear equation solver used in the least-squares algorithm.
Default: `relativeError= 1.0e-10`.

Example 1: Parameter estimation and outlier detection for a time series from biology

This example is based on estimates of the Canadian lynx population. Class `ARMAOutlierIdentification` is used to fit an ARIMA$(2,2,0)$ model of the form $(1-B)(1-\phi_1 B-\phi_2 B^2)Y_t = a_t, t = 1,2,\ldots,144$, $\{a_t\}$ Gaussian White noise, to the given series. Method `compute` determines parameters $\phi_1 = 0.10682$ and $\phi_2 = -0.19666$ and identifies a LS outlier at time point $t = 16$.

```
import com.imsl.stat.*;
import java.util.*;

public class ARMAOutlierIdentificationEx1 {

    public static void main(String args[]) throws Exception {
        double[] series = {
            0.24300E01, 0.25060E01, 0.27670E01, 0.29400E01, 0.31690E01,
            0.34500E01, 0.35940E01, 0.37740E01, 0.36950E01, 0.34110E01,
            0.27180E01, 0.19910E01, 0.22650E01, 0.24460E01, 0.26120E01,
            0.33590E01, 0.34290E01, 0.35330E01, 0.32610E01, 0.26120E01,
            0.21790E01, 0.16530E01, 0.18320E01, 0.23280E01, 0.27370E01,
```

```
    0.30140E01, 0.33280E01, 0.34040E01, 0.29810E01, 0.25570E01,
    0.25760E01, 0.23520E01, 0.25560E01, 0.28640E01, 0.32140E01,
    0.34350E01, 0.34580E01, 0.33260E01, 0.28350E01, 0.24760E01,
    0.23730E01, 0.23890E01, 0.27420E01, 0.32100E01, 0.35200E01,
    0.38280E01, 0.36280E01, 0.28370E01, 0.24060E01, 0.26750E01,
    0.25540E01, 0.28940E01, 0.32020E01, 0.32240E01, 0.33520E01,
    0.31540E01, 0.28780E01, 0.24760E01, 0.23030E01, 0.23600E01,
    0.26710E01, 0.28670E01, 0.33100E01, 0.34490E01, 0.36460E01,
    0.34000E01, 0.25900E01, 0.18630E01, 0.15810E01, 0.16900E01,
    0.17710E01, 0.22740E01, 0.25760E01, 0.31110E01, 0.36050E01,
    0.35430E01, 0.27690E01, 0.20210E01, 0.21850E01, 0.25880E01,
    0.28800E01, 0.31150E01, 0.35400E01, 0.38450E01, 0.38000E01,
    0.35790E01, 0.32640E01, 0.25380E01, 0.25820E01, 0.29070E01,
    0.31420E01, 0.34330E01, 0.35800E01, 0.34900E01, 0.34750E01,
    0.35790E01, 0.28290E01, 0.19090E01, 0.19030E01, 0.20330E01,
    0.23600E01, 0.26010E01, 0.30540E01, 0.33860E01, 0.35530E01,
    0.34680E01, 0.31870E01, 0.27230E01, 0.26860E01, 0.28210E01,
    0.30000E01, 0.32010E01, 0.34240E01, 0.35310E01
};

int[] model = new int[4];
double[] outlierFreeSeries;
double resStdErr, aic, constant;
double[] ar;
int[][] outlierStatistics;
int numOutliers;

model[0] = 2;
model[1] = 0;
model[2] = 1;
model[3] = 2;

ARMAOutlierIdentification armaOutlier
        = new ARMAOutlierIdentification(series);

armaOutlier.setCriticalValue(3.5);
armaOutlier.compute(model);

outlierFreeSeries = armaOutlier.getOutlierFreeSeries();
numOutliers = armaOutlier.getNumberOfOutliers();
outlierStatistics = armaOutlier.getOutlierStatistics();
constant = armaOutlier.getConstant();
ar = armaOutlier.getAR();
resStdErr = armaOutlier.getResidualStandardError();
aic = armaOutlier.getAIC();

System.out.printf("%n%n   ARMA parameters:%n");
System.out.printf(Locale.ENGLISH, "constant:%9.6f%n", constant);
System.out.printf(Locale.ENGLISH, "ar[0]:%12.6f%n", ar[0]);
System.out.printf(Locale.ENGLISH, "ar[1]:%12.6f%n%n", ar[1]);
System.out.printf("Number of outliers:%3d%n", numOutliers);
System.out.printf("%n   Outlier statistics:%n");
System.out.printf("Time point%6sOutlier type%n", " ");
for (int i = 0; i < numOutliers; i++) {
    System.out.printf("%10d%18d%n", outlierStatistics[i][0],
            outlierStatistics[i][1]);
```

```
        }
        System.out.printf(Locale.ENGLISH, "%nRSE:%9.6f%n", resStdErr);
        System.out.printf(Locale.ENGLISH, "AIC:%11.6f", aic);
        System.out.printf("%n%n   Extract from the series:%n");
        System.out.printf("time   original   outlier free%n");
        for (int i = 0; i < 36; i++) {
            System.out.printf(Locale.ENGLISH, "%4d%11.4f%15.4f%n",
                    i + 1, series[i], outlierFreeSeries[i]);
        }
    }
}
```

Output

```
   ARMA parameters:
constant: 0.000000
ar[0]:    0.106532
ar[1]:   -0.195856

Number of outliers:  1

   Outlier statistics:
Time point      Outlier type
        16                 2

RSE: 0.319542
AIC: 282.918191

   Extract from the series:
time   original   outlier free
   1     2.4300         2.4300
   2     2.5060         2.5060
   3     2.7670         2.7670
   4     2.9400         2.9400
   5     3.1690         3.1690
   6     3.4500         3.4500
   7     3.5940         3.5940
   8     3.7740         3.7740
   9     3.6950         3.6950
  10     3.4110         3.4110
  11     2.7180         2.7180
  12     1.9910         1.9910
  13     2.2650         2.2650
  14     2.4460         2.4460
  15     2.6120         2.6120
  16     3.3590         2.6997
  17     3.4290         2.7697
  18     3.5330         2.8737
  19     3.2610         2.6017
  20     2.6120         1.9527
  21     2.1790         1.5197
  22     1.6530         0.9937
  23     1.8320         1.1727
```

```
 24     2.3280         1.6687
 25     2.7370         2.0777
 26     3.0140         2.3547
 27     3.3280         2.6687
 28     3.4040         2.7447
 29     2.9810         2.3217
 30     2.5570         1.8977
 31     2.5760         1.9167
 32     2.3520         1.6927
 33     2.5560         1.8967
 34     2.8640         2.2047
 35     3.2140         2.5547
 36     3.4350         2.7757
```

Example 2: Parameter estimation and outlier detection for an artificial time series

This example is an artificial realization of an ARMA$(1,1)$ process via formula
$Y_t - 0.8Y_{t-1} = 10.0 + a_t + 0.5a_{t-1}, t = 1, \ldots, 300$, $\{a_t\}$ Gaussian white noise, $E[Y_t] = 50.0$. An additive outlier with $\omega_1 = 4.5$ was added at time point $t = 150$, a temporary change outlier with $\omega_2 = 3.0$ was added at time point $t = 200$.

```
import com.imsl.stat.*;
import java.util.*;

public class ARMAOutlierIdentificationEx2 {

    public static void main(String args[]) throws Exception {
        double resStdErr, aic, constant;
        int[][] outlierStatistics;
        int numOutliers;
        double[] omegaWeights, ar, ma;
        int[] model = new int[4];

        double[] series = {
            50.0000000, 50.2728081, 50.6242599, 51.0373917, 51.9317627,
            50.3494759, 51.6597252, 52.7004929, 53.5499802, 53.1673279,
            50.2373505, 49.3373871, 49.5516472, 48.6692696, 47.6606636,
            46.8774185, 45.7315445, 45.6469727, 45.9882355, 45.5216560,
            46.0479660, 48.1958656, 48.6387749, 49.9055367, 49.8077278,
            47.7858467, 47.9386749, 49.7691956, 48.5425873, 49.1239853,
            49.8518791, 50.3320694, 50.9146347, 51.8772049, 51.8745689,
            52.3394470, 52.7273712, 51.4310036, 50.6727448, 50.8370399,
            51.2843437, 51.8162918, 51.6933670, 49.7038231, 49.0189247,
            49.455703, 50.2718010, 49.9605980, 51.3775749, 50.2285385,
            48.2692299, 47.6495590, 49.2938499, 49.1924858, 49.6449242,
            50.0446815, 51.9972496, 54.2576981, 52.9835434, 50.4193535,
            50.3617897, 51.8276901, 53.1239929, 54.0682144, 54.9238319,
            55.6877632, 54.8896332, 54.0701065, 52.2754097, 52.2522354,
            53.1248703, 51.1287193, 50.5003815, 49.6504173, 47.2453079,
            45.4555626, 45.8449707, 45.9765129, 45.7682228, 45.2343674,
            46.6496811, 47.0894432, 49.3368340, 50.8058052, 49.9132500,
            49.5893288, 48.2470627, 46.9779968, 45.6760864, 45.7070389,
```

```
    46.6158409, 47.5303612, 47.5630417, 47.0389214, 46.0352287,
    45.8161545, 45.7974396, 46.0015373, 45.3796463, 45.3461685,
    47.6444016, 49.3327446, 49.3810692, 50.2027817, 51.4567032,
    52.3986320, 52.5819206, 52.7721825, 52.6919098, 53.3274345,
    55.1345940, 56.8962631, 55.7791634, 55.0616989, 52.3551178,
    51.3264084, 51.0968323, 51.1980476, 52.8001442, 52.0545082,
    50.8742943, 51.5150337, 51.2242050, 50.5033989, 48.7760124,
    47.4179192, 49.7319527, 51.3320541, 52.3918304, 52.4140434,
    51.0845947, 49.6485748, 50.6893463, 52.9840813, 53.3246994,
    52.4568024, 51.9196091, 53.6683121, 53.4555359, 51.7755814,
    49.2915611, 49.8755112, 49.4546776, 48.6171913, 49.9643021,
    49.3766441, 49.2551308, 50.1021881, 51.0769119, 55.8328133,
    52.0212708, 53.4930801, 53.2147255, 52.2356453, 51.9648819,
    52.1816330, 51.9898071, 52.5623627, 51.0717278, 52.2431946,
    53.6943054, 54.3752098, 54.1492615, 53.8523254, 52.1093712,
    52.3982697, 51.2405128, 50.3018112, 51.3819618, 49.5479546,
    47.5024452, 47.4447708, 47.8939056, 48.4070015, 48.2440681,
    48.7389755, 49.7309227, 49.1998024, 49.5798340, 51.1196213,
    50.6288414, 50.3971405, 51.6084099, 52.4564743, 51.6443901,
    52.4080658, 52.4643364, 52.6257210, 53.1604691, 51.9309731,
    51.4137230, 52.1233368, 52.9867249, 53.3180733, 51.9647636,
    50.7947655, 52.3815842, 50.8353729, 49.4136009, 52.8355217,
    52.2234840, 51.1392517, 48.5245132, 46.8700218, 46.1607285,
    45.2324257, 47.4157829, 48.9989090, 49.6230736, 50.4352913,
    51.1652985, 50.2588654, 50.7820129, 51.0448799, 51.2880516,
    49.6898804, 49.0288200, 49.9338837, 48.2214432, 46.2103348,
    46.9550171, 47.5595894, 47.7176018, 48.4502945, 50.9816895,
    51.6950073, 51.6973495, 52.1941261, 51.8988075, 52.5617599,
    52.0218391, 49.5236053, 47.9684906, 48.2445183, 48.8275146,
    49.7176971, 51.5649338, 52.5627213, 52.0182419, 50.9688835,
    51.5846901, 50.9486771, 48.8685837, 48.5600624, 48.4760094,
    48.5348396, 50.4187813, 51.2542381, 50.1872864, 50.4407692,
    50.6222687, 50.4972000, 51.0036087, 51.3367500, 51.7368202,
    53.0463791, 53.6261253, 52.0728683, 48.9740753, 49.3280830,
    49.2733917, 49.8519020, 50.8562126, 49.5594254, 49.6109200,
    48.3785629, 48.0026474, 49.4874268, 50.1596375, 51.8059540,
    53.0288620, 51.3321075, 49.3114815, 48.7999306, 47.7201881,
    46.3433914, 46.5303612, 47.6294632, 48.6012459, 47.8567657,
    48.0604057, 47.1352806, 49.5724792, 50.5566483, 49.4182968,
    50.5578079, 50.6883736, 50.6333389, 51.9766159, 51.0595245,
    49.3751640, 46.9667702, 47.1658173, 47.4411278, 47.5360374,
    48.9914742, 50.4747620, 50.2728043, 51.9117165, 53.7627792
};

model[0] = 1;
model[1] = 1;
model[2] = 1;
model[3] = 0;

ARMAOutlierIdentification armaOutlier
        = new ARMAOutlierIdentification(series);

armaOutlier.setRelativeError(1.0e-5);
armaOutlier.compute(model);

double[] outlierFreeSeries = armaOutlier.getOutlierFreeSeries();
```

```
        numOutliers = armaOutlier.getNumberOfOutliers();
        outlierStatistics = armaOutlier.getOutlierStatistics();
        omegaWeights = armaOutlier.getOmegaWeights();
        constant = armaOutlier.getConstant();
        ar = armaOutlier.getAR();
        ma = armaOutlier.getMA();
        resStdErr = armaOutlier.getResidualStandardError();
        aic = armaOutlier.getAIC();

        System.out.printf("%n%n   ARMA parameters:%n");
        System.out.printf(Locale.ENGLISH, "constant:%11.6f%n", constant);
        System.out.printf(Locale.ENGLISH, "ar[0]:%14.6f%n", ar[0]);
        System.out.printf(Locale.ENGLISH, "ma[0]:%14.6f%n%n", ma[0]);
        System.out.printf("Number of outliers:%3d%n%n", numOutliers);
        System.out.printf("   Outlier statistics:%n");
        System.out.printf("Time point%6sOutlier type%n", " ");
        for (int i = 0; i < numOutliers; i++) {
            System.out.printf("%10d%18d%n", outlierStatistics[i][0],
                    outlierStatistics[i][1]);
        }
        System.out.printf("%n   Omega statistics:%n");
        System.out.printf("Time point%6sOmega%n", " ");
        for (int i = 0; i < numOutliers; i++) {
            System.out.printf(Locale.ENGLISH, "%10d%11.6f%n",
                    outlierStatistics[i][0], omegaWeights[i]);
        }
        System.out.printf(Locale.ENGLISH, "%nRSE:%9.6f%n", resStdErr);
        System.out.printf(Locale.ENGLISH, "AIC:%12.6f", aic);
    }
}
```

Output

```
   ARMA parameters:
constant:  10.834213
ar[0]:      0.785117
ma[0]:     -0.496502

Number of outliers:  2

   Outlier statistics:
Time point      Outlier type
       150                 1
       200                 3

   Omega statistics:
Time point      Omega
       150   4.477876
       200   3.381473

RSE: 1.007223
AIC: 1417.044613
```

Example 3: Forecasting an outlier contaminated time series

This example is a realization of an ARMA$(2,1)$ process described by the model
$Y_t - Y_{t-1} + 0.24Y_{t-2} = 10.0 + a_t + 0.5a_{t-1}$, $\{a_t\}$ a Gaussian White noise process. An additive outlier with $\omega_1 = 4.5$ was added at time point $t = 150$, a temporary change outlier with $\omega_2 = 3.0$ was added at time point $t = 200$.

Outliers were artificially added to the outlier free series $\{Y - t\}_{t=1,\ldots,280}$ at time points $t = 150$ (level shift with $\omega_1 = +2.5$) and $t = 200$ (additive outlier with $\omega_2 = +3.2$), resulting in the outlier contaminated series $\{Z_t\}_{t=1,\ldots,280}$. For both series, forecasts were determined for time points $t = 281,\ldots,290$ and compared with the actual values of the series.

```
import com.imsl.stat.*;
import java.util.*;

public class ARMAOutlierIdentificationEx3 {

    public static void main(String args[]) throws Exception {
        double resStdErr, aic, constant;
        int[][] outlierStatistics;
        int numOutliers, nForecast = 10;
        int[] model = new int[4];
        double[] ar, ma;
        double[] outlierContaminatedForecast, outlierFreeForecast,
                probabilityLimits, psiWeights;

        double[] outlierContaminatedSeries = {
            41.6699982, 41.6699982, 42.0752144, 42.6123962, 43.6161919,
            42.1932831, 43.1055450, 44.3518715, 45.3961258, 45.0790215,
            41.8874397, 40.2159805, 40.2447319, 39.6208458, 38.6873589,
            37.9272423, 36.8718872, 36.8310852, 37.4524879, 37.3440933,
            37.9861374, 40.3810501, 41.3464622, 42.6495285, 42.6096764,
            40.3134537, 39.7971268, 41.5401535, 40.7160759, 41.0363541,
            41.8171883, 42.4190292, 43.0318832, 43.9968109, 44.0419617,
            44.3225212, 44.6082611, 43.2199631, 42.0419197, 41.9679718,
            42.4926224, 43.2091255, 43.2512283, 41.2301674, 40.1057358,
            40.4510574, 41.5329170, 41.5678177, 43.0090141, 42.1592140,
            39.9234505, 38.8394127, 40.4319878, 40.8679352, 41.4551926,
            41.9756317, 43.9878922, 46.5736389, 45.5939293, 42.4487762,
            41.5325394, 42.8830910, 44.5771217, 45.8541985, 46.8249474,
            47.5686378, 46.6700745, 45.4120026, 43.2305107, 42.7635345,
            43.7112923, 42.0768661, 41.1835632, 40.3352280, 37.9761467,
            35.9550056, 36.3212509, 36.9925880, 37.2625008, 37.0040665,
            38.5232544, 39.4119797, 41.8316803, 43.7091446, 42.9381447,
            42.1066780, 40.3771248, 38.6518707, 37.0550499, 36.9447708,
            38.1017685, 39.4727097, 39.8670387, 39.3820763, 38.2180786,
            37.7543488, 37.7265244, 38.0290642, 37.5531158, 37.4685936,
            39.8233147, 42.0480766, 42.4053535, 43.0117416, 44.1289330,
            45.0393829, 45.1114540, 45.0086479, 44.6560631, 45.0278931,
            46.7830849, 48.7649765, 47.7991905, 46.5339661, 43.3679199,
            41.6420822, 41.2694893, 41.5959740, 43.5330009, 43.3643608,
            42.1471291, 42.5552788, 42.4521446, 41.7629128, 39.9476891,
            38.3217010, 40.5318718, 42.8811569, 44.4796944, 44.6887932,
            43.1670265, 41.2226143, 41.8330154, 44.3721924, 45.2697029,
            44.4174194, 43.5068550, 44.9793015, 45.0585403, 43.2746620,
```

```
    40.3317070, 40.3880501, 40.2627106, 39.6230278, 41.0305252,
    40.9262009, 40.8326912, 41.7084885, 42.9038048, 45.8650513,
    46.5231590, 47.9916115, 47.8463135, 46.5921936, 45.8854408,
    45.9130440, 45.7450371, 46.2964249, 44.9394569, 45.8141251,
    47.5284042, 48.5527802, 48.3950577, 47.8753052, 45.8880005,
    45.7086983, 44.6174774, 43.5567932, 44.5891113, 43.1778679,
    40.9405632, 40.6206894, 41.3330421, 42.2759552, 42.4744949,
    43.0719833, 44.2178459, 43.8956337, 44.1033440, 45.6241455,
    45.3724861, 44.9167595, 45.9180603, 46.9077835, 46.1666603,
    46.6013489, 46.6592331, 46.7291603, 47.1908340, 45.9784355,
    45.1215782, 45.6791115, 46.7379875, 47.3036957, 45.9968834,
    44.4669495, 45.7734680, 44.6315041, 42.9911766, 46.3842583,
    43.7214432, 43.5276833, 41.3946495, 39.7013168, 39.1033401,
    38.5292892, 41.0096245, 43.4535828, 44.6525154, 45.5725899,
    46.2815285, 45.2766647, 45.3481712, 45.5039482, 45.6745682,
    44.0144806, 42.9305000, 43.6785469, 42.2500534, 40.0007210,
    40.4477005, 41.4432716, 42.0058670, 42.9357758, 45.6758842,
    46.8809929, 46.8601494, 47.0449791, 46.5420647, 46.8939934,
    46.2963371, 43.5479164, 41.3864059, 41.4046364, 42.3037987,
    43.6223717, 45.8602371, 47.3016396, 46.8632469, 45.4651413,
    45.6275482, 44.9968376, 42.7558670, 42.0218239, 41.9883728,
    42.2571678, 44.3708687, 45.7483635, 44.8832512, 44.7945862,
    44.8922577, 44.7409401, 45.1726494, 45.5686874, 45.9946709,
    47.3151054, 48.0654068, 46.4817467, 42.8618279, 42.4550323,
    42.5791168, 43.4230957, 44.7787971, 43.8317108, 43.6481781,
    42.4183960, 41.8426285, 43.3475227, 44.4749908, 46.3498306,
    47.8599319, 46.2449913, 43.6044006, 42.4563484, 41.2715340,
    39.8492508, 39.9997292, 41.4410820, 42.9388237, 42.5687332
};

// Actual values of the outlier contaminated series for
// t = 181,...,190
double[] exactForecastOutlierContaminatedSeries = {
    42.6384087, 41.7088661, 43.9399033, 45.4284401, 44.4558411,
    45.1761856, 45.3489113, 45.1892662, 46.3754730, 45.6082802
};

// Actual values of the outlier free series for
// t = 181,...,190. This are the values of the outlier contaminated
// series - 2.5
double[] exactForecastOutlierFreeSeries = {
    40.1384087, 39.2088661, 41.4399033, 42.9284401, 41.9558411,
    42.6761856, 42.8489113, 42.6892662, 43.8754730, 43.1082802
};

model[0] = 2;
model[1] = 1;
model[2] = 1;
model[3] = 0;

ARMAOutlierIdentification armaOutlier
        = new ARMAOutlierIdentification(outlierContaminatedSeries);

armaOutlier.setRelativeError(1.0e-5);
armaOutlier.compute(model);
armaOutlier.computeForecasts(nForecast);
```

```
        numOutliers = armaOutlier.getNumberOfOutliers();
        outlierStatistics = armaOutlier.getOutlierStatistics();
        constant = armaOutlier.getConstant();
        ar = armaOutlier.getAR();
        ma = armaOutlier.getMA();
        resStdErr = armaOutlier.getResidualStandardError();
        aic = armaOutlier.getAIC();

        outlierContaminatedForecast = armaOutlier.getForecast();
        outlierFreeForecast = armaOutlier.getOutlierFreeForecast();
        probabilityLimits = armaOutlier.getDeviations();
        psiWeights = armaOutlier.getPsiWeights();

        System.out.printf("%n%n   ARMA parameters:%n");
        System.out.printf(Locale.ENGLISH, "constant: %9.6f%n", constant);
        System.out.printf(Locale.ENGLISH, "ar[0]: %12.6f%n", ar[0]);
        System.out.printf(Locale.ENGLISH, "ar[1]: %12.6f%n", ar[1]);
        System.out.printf(Locale.ENGLISH, "ma[0]: %12.6f%n%n", ma[0]);
        System.out.printf("Number of outliers: %d%n%n", numOutliers);
        System.out.printf("   Outlier statistics:%n");
        System.out.printf("Time point%8sOutlier type%n", " ");
        for (int i = 0; i < numOutliers; i++) {
            System.out.printf("%10d%20d%n", outlierStatistics[i][0],
                    outlierStatistics[i][1]);
        }
        System.out.printf(Locale.ENGLISH, "%nRSE:%9.6f%n", resStdErr);
        System.out.printf(Locale.ENGLISH, "AIC:%12.6f%n%n", aic);
        System.out.printf("%5s* * * Forecast Table for outlier "
                + "contaminated series * * *%n%n", " ");
        System.out.printf("%10sExact%5sforecast%6slimits%9spsi%n",
                " ", " ", " ", " ");
        for (int i = 0; i < nForecast; i++) {
            System.out.printf(Locale.ENGLISH, "%7s%8.4f%13.4f%12.4f%12.4f%n",
                    " ", exactForecastOutlierContaminatedSeries[i],
                    outlierContaminatedForecast[i], probabilityLimits[i],
                    psiWeights[i]);
        }

        System.out.printf("%n%n%5s* * * Forecast Table for outlier "
                + "free series * * *%n%n", " ");
        System.out.printf("%10sExact%5sforecast%6slimits%9spsi%n",
                " ", " ", " ", " ");
        for (int i = 0; i < nForecast; i++) {
            System.out.printf(Locale.ENGLISH, "%7s%8.4f%13.4f%12.4f%12.4f%n",
                    " ", exactForecastOutlierFreeSeries[i],
                    outlierFreeForecast[i], probabilityLimits[i],
                    psiWeights[i]);
        }
    }
}
```

Output

```
   ARMA parameters:
constant:  8.891421
ar[0]:     0.944052
ar[1]:    -0.150404
ma[0]:    -0.558928

Number of outliers: 2

   Outlier statistics:
Time point        Outlier type
       150                   2
       200                   1

RSE: 1.004306
AIC: 1323.617443

     * * * Forecast Table for outlier contaminated series * * *

          Exact     forecast      limits         psi
        42.6384      42.3158      1.9684      1.5030
        41.7089      42.7933      3.5535      1.2685
        43.9399      43.2822      4.3430      0.9715
        45.4284      43.6718      4.7453      0.7263
        44.4558      43.9662      4.9560      0.5396
        45.1762      44.1854      5.0686      0.4002
        45.3489      44.3482      5.1294      0.2966
        45.1893      44.4688      5.1625      0.2198
        46.3755      44.5582      5.1806      0.1629
        45.6083      44.6245      5.1906      0.1207


     * * * Forecast Table for outlier free series * * *

          Exact     forecast      limits         psi
        40.1384      40.5904      1.9684      1.5030
        39.2089      41.0679      3.5535      1.2685
        41.4399      41.5567      4.3430      0.9715
        42.9284      41.9464      4.7453      0.7263
        41.9558      42.2407      4.9560      0.5396
        42.6762      42.4600      5.0686      0.4002
        42.8489      42.6227      5.1294      0.2966
        42.6893      42.7434      5.1625      0.2198
        43.8755      42.8328      5.1806      0.1629
        43.1083      42.8991      5.1906      0.1207
```

AutoARIMA class

public class com.imsl.stat.AutoARIMA implements Serializable, Cloneable

Automatically identifies time series outliers, determines parameters of a multiplicative seasonal ARIMA$(p,0,q)\times(0,d,0)_s$ model and produces forecasts that incorporate the effects of outliers whose effects persist beyond the end of the series.

Class `AutoARIMA` determines the parameters of a multiplicative seasonal ARIMA$(p,0,q)\times(0,d,0)_s$ model, and then uses the fitted model to identify outliers and prepare forecasts. The order of this model can be specified or automatically determined through use of an overloaded `compute` method. Potential missing values in the time series are estimated prior to the parameter and outlier computations.

The ARIMA$(p,0,q)\times(0,d,0)_s$ model handled by class `AutoARIMA` has the following form:

$$\phi(B)\Delta_s^d(Y_t-\mu)=\theta(B)a_t,\ t=1,2,\ldots,n,$$

where

$$\phi(B)=1-\phi_1B-\phi_2B^2-\ldots-\phi_pB^p,\ \theta(B)=1-\theta_1B-\theta_2B^2-\ldots-\theta_qB^q,\ \Delta_s^d=(1-B_s)^d$$

and

$$B^kY_t=Y_{t-k}$$

It is assumed that all roots of $\phi(B)$ and $\theta(B)$ lie outside the unit circle. Clearly, if $s=1$ the model reduces to the traditional ARIMA(p,d,q) model.

Y_t is the unobserved, outlier-free time series with mean μ, and white noise a_t. This model is referred to as the underlying, outlier-free model. Class `AutoARIMA` does not assume that this series is observable. It assumes that the observed values might be contaminated by one or more outliers, whose effects are added to the underlying outlier-free series:

$$Y_t^*=Y_t+\text{outlierEffect}_t$$

Outlier identification uses the algorithm developed by Chen and Liu (1993). Outliers are classified into 1 of 5 types:

1. innovational
2. additive
3. level shift
4. temporary change and
5. unable to identify

Once the model parameters are estimated and the outliers are identified, class `AutoARIMA` estimates Y_t, the outlier-free series representation of the data, by removing the estimated outlier effects. Parameter estimation and outlier detection are based on methods from class `ARMAOutlierIdentification`.

Using the information about the adjusted ARIMA$(p,0,q)\times(0,d,0)_s$ model and the removed outliers, forecasts are then prepared for the outlier-free series. Outlier effects are added to these forecasts to produce a forecast for the observed series, Y_t^*. If there are no outliers, then the forecasts for the outlier-free series and the observed series will be identical.

Model selection techniques

Users have an option of either specifying specific values for p, q, s, d or have class `AutoARIMA` automatically select best fit values. Model selection can be conducted in one of three ways listed below depending upon which `compute` method of class `AutoARIMA` is invoked.

Technique 1: Automatic $\text{ARIMA}(p,0,0)\times(0,d,0)_s$ **Selection**

This technique, chosen by use of method `compute(int maxlag)`, tries to fit a model of the form

$$\phi(B)\Delta_s^d(Y_t-\mu)=a_t$$

to the outlier free series Y_t.

It initially searches for the $\text{AR}(p)$ representation with minimum value of the chosen information criterion (AIC, AICC or BIC) for the noisy data, where $p=0,\ldots,\text{maxlag}$.

If the user calls methods `setPeriods` and `setDifferenceOrders` prior to invoking the `compute` method, then the values in arrays `periods` and `orders` are included in the search to find an optimum $\text{ARIMA}(p,0,0)\times(0,d,0)_s$ representation of the series. Here, every possible combination of values for s, d in `periods` and `orders`, respectively, are examined. The best found model order is then used as input for the parameter and outlier detection routine.

The optimum values for p, q, s and d are returned through method `getOptimumModelOrder`.

Technique 2: Grid Search

This technique, chosen by means of method `compute(int[] arOrders, int[] maOrders)`, conducts a grid search for p and q using all possible combinations of candidate values in `arOrders` and `maOrders`.

If methods `setPeriods` and `setDifferenceOrders` are called prior to invoking the `compute` method, then the grid search is extended to include the candidate values for s and d given in arrays `periods` and `orders`, respectively.

If method `setDifferenceOrders` is not called prior to `compute`, then $d=0$ by default, and therefore no seasonal adjustment is attempted. The grid search is then restricted to searching for optimum values of p and q only.

The optimum values for p, q, s and d are contained in the array returned by method `getOptimumModelOrder`.

Technique 3: Specified $\text{ARIMA}(p,0,q)\times(0,d,0)_s$ **Model**

In the third technique, selectable by means of method `compute(int p, int q, int s, int d)`, specific values for p, q, s and d are given. This technique has essentially the same functionality as class `ARMAOutlierIdentification` but with the additional option of missing value estimation.

Outliers

The algorithm of Chen and Liu (1993) is used to identify outliers. The number of outliers identified is returned via method `getNumberOfOutliers`. Both the time and classification for these outliers are contained in the matrix returned by method `getOutlierStatistics`. Outliers are classified into one of five categories based upon the standardized statistic for each outlier type. The time at which the outlier occurred is given in the first column of the returned matrix. The outlier identifier returned in the second

column is according to the descriptions in the following table:

Outlier Identifier	Name	General Description
INNOVATIONAL	Innovational Outlier (IO)	Innovational outliers persist. That is, there is an initial impact at the time the outlier occurs. This effect continues in a lagged fashion with all future observations. The lag coefficients are determined by the coefficients of the underlying ARIMA$(p,0,q) \times (0,d,0)_s$ model.
ADDITIVE	Additive Outlier (AO)	Additive outliers do not persist. As the name implies, an additive outlier affects only the observation at the time the outlier occurs. Hence additive outliers have no effect on future forecasts.
LEVEL_SHIFT	Level Shift (LS)	Level shift outliers persist. They have the effect of either raising or lowering the mean of the series starting at the time the outlier occurs. This shift in the mean is abrupt and permanent.
TEMPORARY_CHANGE	Temporary Change (TC)	Temporary change outliers persist and are similar to level shift outliers with one major exception. Like level shift outliers, there is an abrupt change in the mean of the series at the time this outlier occurs. However, unlike level shift outliers, this shift is not permanent. The TC outlier gradually decays, eventually bringing the mean of the series back to its original value. The rate of this decay is modeled using method `setDelta`. The default of `delta = 0.7` is the value recommended for general use by Chen and Liu (1993).
UNABLE_TO_IDENTIFY	Unable to Identify (UI)	If an outlier is identified as the last observation, then the algorithm is unable to determine the outlier's classification. For forecasting, a UI outlier is treated as an IO outlier. That is, its effect is lagged into the forecasts.

Except for additive outliers (AO), the effect of an outlier persists to observations following that outlier. Forecasts produced by methods of class `AutoARIMA` take this into account.

For more information on forecasting an outlier contaminated series, see the description of class `ARMAOutlierIdentification`.

Fields

ADDITIVE

```
static final public int ADDITIVE
```

Indicates detection of an additive outlier.

AIC

```
static final public int AIC
```

Indicates that Akaike's information criterion (AIC) is used in the optimum model determination.

AICC

```
static final public int AICC
```

Indicates that Akaike's corrected information criterion (AICC) is used in the optimum model determination.

BIC

```
static final public int BIC
```

Indicates that the Bayesian information criterion (BIC) is used in the optimum model determination.

INNOVATIONAL

```
static final public int INNOVATIONAL
```

Indicates detection of an innovational outlier.

LEVEL_SHIFT

```
static final public int LEVEL_SHIFT
```

Indicates detection of a level shift outlier.

TEMPORARY_CHANGE

```
static final public int TEMPORARY_CHANGE
```

Indicates detection of a temporary change outlier.

UNABLE_TO_IDENTIFY

```
static final public int UNABLE_TO_IDENTIFY
```

Indicates detection of an outlier that cannnot be categorized.

Constructor

AutoARIMA

```
public AutoARIMA(int[] times, double[] x)
```

Description

Constructor for `AutoARIMA`.

Parameters

`times` – an `int` array of length `nObs`, where `nObs` is the number of observed time series values, containing the time points $t_1, \ldots, t_{\text{nObs}}$ at which the time series was observed. It is required that $t_1, \ldots, t_{\text{nObs}}$ are in strictly increasing order. Times for missing values are identified as non-incremental gaps in this series. A gap of missing values in `x` is assumed if the difference between two consecutive values is greater than 1, i.e. $t_{i+1} - t_i > 1$. The difference is the number of missing values in the gap.

`x` – a `double` array containing the observations $Y_1^*, Y_2^*, \ldots, Y_{\text{nObs}}^*$ at the times given in array `times`. This series can contain outliers and missing observations.

Methods

compute

```
final public void compute(int maxlag) throws ARMA.MatrixSingularException,
ARMA.TooManyCallsException, ARMA.IncreaseErrRelException,
ARMA.NewInitialGuessException, ARMA.IllConditionedException,
ARMA.TooManyITNException, ARMA.TooManyFcnEvalException,
ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException,
ZeroPolynomial.DidNotConvergeException, SingularMatrixException,
Cholesky.NotSPDException, AutoARIMA.NoAcceptableModelFoundException
```

Description

Estimates potential missing values, detects and determines outliers and simultaneously fits an optimum model from a set of different $\text{ARIMA}(p,0,0) \times (0,d,0)_s$ models to the outlier free time series.

Parameter

`maxlag` – the maximum value for p allowed when fitting $\text{ARIMA}(p,0,0) \times (0,d,0)_s$ models to the given series, $0 \le p \le \text{maxlag}$. It is required that $1 \le \text{maxlag} \le$ `x.length`. The optimum $\text{ARIMA}(p,0,0) \times (0,d,0)_s$ model is determined according to the model selection criterion chosen by the user, see method `setModelSelectionCriterion`.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input matrix to ARAutoUnivariate is singular.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the intermediate or final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the intermediate or final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ZeroPolynomial.DidNotConvergeException` is thrown if the algorithm computing the roots of the AR- or MA- polynomial does not converge.

`SingularMatrixException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is singular.

`Cholesky.NotSPDException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is not positive definite.

`NoAcceptableModelFoundException` is thrown if no appropriate ARIMA model for the given time series could be found.

compute

```
final public void compute(int[] arOrders, int[] maOrders) throws
ARMA.MatrixSingularException, ARMA.TooManyCallsException,
ARMA.IncreaseErrRelException, ARMA.NewInitialGuessException,
ARMA.IllConditionedException, ARMA.TooManyITNException,
ARMA.TooManyFcnEvalException, ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException,
ZeroPolynomial.DidNotConvergeException, SingularMatrixException,
Cholesky.NotSPDException, AutoARIMA.NoAcceptableModelFoundException
```

Description

Estimates potential missing values, detects and determines outliers and simultaneously fits an optimum model from a set of different ARIMA$(p,0,q) \times (0,d,0)_s$ models to the outlier free time series.

Parameters

`arOrders` – an `int` array containing all possible AR orders to consider in the optimum model search. It is required that all values in `arOrders` are greater than or equal to zero.

`maOrders` – an `int` array containing all possible MA orders to consider in the optimum model search. It is required that all values in `maOrders` are greater than or equal to zero.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input matrix to ARAutoUnivariate is singular.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the intermediate or final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the intermediate or final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ZeroPolynomial.DidNotConvergeException` is thrown if the algorithm computing the roots of the AR- or MA- polynomial does not converge.

`SingularMatrixException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is singular.

`Cholesky.NotSPDException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is not positive definite.

`NoAcceptableModelFoundException` is thrown if no appropriate ARIMA model for the given time series could be found.

compute

```
final public void compute(int p, int q, int s, int d) throws
ARMA.MatrixSingularException, ARMA.TooManyCallsException,
ARMA.IncreaseErrRelException, ARMA.NewInitialGuessException,
ARMA.IllConditionedException, ARMA.TooManyITNException,
ARMA.TooManyFcnEvalException, ARMA.TooManyJacobianEvalException,
ARAutoUnivariate.TriangularMatrixSingularException,
ARMAMaxLikelihood.NonInvertibleException,
ARMAMaxLikelihood.NonStationaryException,
ZeroPolynomial.DidNotConvergeException, SingularMatrixException,
Cholesky.NotSPDException, AutoARIMA.NoAcceptableModelFoundException
```

Description

Estimates potential missing values, detects and determines outliers and simultaneously fits an ARIMA$(p,0,q) \times (0,d,0)_s$ model to the outlier free time series.

Parameters

`p` – a non-negative scalar `int`, the order of the AR part of the model.

`q` – a non-negative scalar `int`, the order of the MA part of the model.

`s` – a positive scalar `int`, the period of the difference used in the model.

`d` – a non-negative scalar `int`, the order of the difference used in the model.

Exceptions

`ARMA.MatrixSingularException` is thrown if the input matrix is singular.

`ARMA.TooManyCallsException` is thrown if the number of calls to the function has exceeded the maximum number of iterations times the number of moving average (MA) parameters + 1.

`ARMA.IncreaseErrRelException` is thrown if the bound for the relative error is too small.

`ARMA.NewInitialGuessException` is thrown if the iteration has not made good progress.

`ARMA.IllConditionedException` is thrown if the problem is ill-conditioned.

`ARMA.TooManyITNException` is thrown if the maximum number of iterations is exceeded.

`ARMA.TooManyFcnEvalException` is thrown if the maximum number of function evaluations is exceeded.

`ARMA.TooManyJacobianEvalException` is thrown if the maximum number of Jacobian evaluations is exceeded.

`ARAutoUnivariate.TriangularMatrixSingularException` is thrown if the input matrix to ARAutoUnivariate is singular.

`ARMAMaxLikelihood.NonInvertibleException` is thrown if the intermediate or final maximum likelihood estimates for the time series are noninvertible.

`ARMAMaxLikelihood.NonStationaryException` is thrown if the intermediate or final maximum likelihood estimates for the time series are nonstationary.

`ARMAMaxLikelihood.InitialMAException` is thrown if the initial values provided for the moving average terms are noninvertible. In this case, `ARMAMaxLikelihood` terminates and does not compute the time series estimates.

`ZeroPolynomial.DidNotConvergeException` is thrown if the algorithm computing the roots of the AR- or MA- polynomial does not converge.

`SingularMatrixException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is singular.

`Cholesky.NotSPDException` is thrown if during the computation of a small perturbation of the matrix product $A^T A$, it is found that A, the matrix used in the determination of the ω weights, is not positive definite.

`NoAcceptableModelFoundException` is thrown if no appropriate ARIMA model for the given time series could be found.

forecast

```
final public void forecast(int nForecast)
```

Description

Computes forecasts, associated probability limits and ψ weights for the given outlier contaminated time series. Note that one of the `compute` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Parameter

`nForecast` – an `int` scalar representing the number of forecasts that will be computed. `nForecast` must be greater than 0. Forecast origin is the time point of the last observed value in the time series, t_{nObs}. Forecasts are computed for lead times $1, 2, \ldots, \text{nForecast}$, i.e. time points $t_{\text{nObs}} + 1, t_{\text{nObs}} + 2, \ldots, t_{\text{nObs}} + \text{nForecast}$.

getAIC

```
public double getAIC()
```

Description

Returns Akaike's information criterion (AIC) for the optimum model.

Returns

a `double` scalar containing Akaike's information criterion (AIC) for the optimum outlier free series. One of the `compute` methods must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getAICC

```
public double getAICC()
```

Description

Returns Akaike's Corrected Information Criterion (AICC) for the optimum model.

Returns

a `double` scalar containing Akaike's Corrected Information Criterion (AICC) for the optimum outlier free series. One of the `compute` methods must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getAR

```
public double[] getAR()
```

Description

Returns the final autoregressive parameter estimates of the optimum model. Note that one of the `compute` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a `double` array containing the final autoregressive parameter estimates. If the optimum model has no AR component then a zero-length array is returned.

getBIC

```
public double getBIC()
```

Description

Returns the Bayesian Information Criterion (BIC) for the optimum model.

Returns

a `double` scalar containing the Bayesian Information Criterion (BIC) for the optimum outlier free series. One of the `compute` methods must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getCompleteTimeSeries

```
public double[] getCompleteTimeSeries()
```

Description

Returns the original series with potentially missing values replaced by estimates.

Returns

a `double` array containing the original time series with missing values replaced by estimates. One of the `compute` methods must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getCompleteTimes

```
public int[] getCompleteTimes()
```

Description

Returns all time points at which the original series was observed, including values for times with missing values in `x`.

Returns

an `int` array of length `times[times.length-1]-times[0]+1` containing the times at which the time series (including missing values) was observed. One of the `compute` methods must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getConstant

```
public double getConstant()
```

Description

Returns the constant parameter estimate for the optimum model. Note that one of the `compute` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a `double` scalar containing the constant parameter estimate for the optimum model.

getDeviations

```
public double[] getDeviations()
```

Description

Returns the deviations used for calculating the forecast confidence limits.

Returns

a `double` array of length `nForecast` containing the deviations from each forecast for calculating forecast confidence intervals. The confidence level is specified in `confidence`. Method `forecast` must be invoked before this method is called. Otherwise, an `IllegalStateException` exception is thrown. By default, `confidence` = 0.95.

getForecast

```
public double[] getForecast()
```

Description

Returns forecasts for the original outlier contaminated series.

Returns

a `double` array of length `nForecast` containing the forecasts for the original series. Forecast origin is the time point of the last observed value in the time series, t_{nObs}. Forecasts are returned for lead times $1, 2, \ldots, \text{nForecast}$, i.e. time points $t_{\text{nObs}} + 1, t_{\text{nObs}} + 2, \ldots, t_{\text{nObs}} + \text{nForecast}$. Method `forecast` must be invoked before this method is called. Otherwise, an `IllegalStateException` exception is thrown.

getMA

```
public double[] getMA()
```

Description

Returns the final moving average parameter estimates of the optimum model. Note that one of the `compute` methods must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a `double` array containing the final moving average parameter estimates. If the optimum model has no MA component then a zero-length array is returned.

getNumberOfOutliers

```
public int getNumberOfOutliers()
```

Description

Returns the number of outliers detected.

Returns

an `int` scalar containing the number of outliers detected. The `compute` method must be invoked first before invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

getOptimumModelOrder

```
public int[] getOptimumModelOrder()
```

Description

Returns the order $(p,0,q) \times (0,d,0)_s$ of the optimum model. Note that one of the `compute` methods must be invoked first before calling this method. Otherwise, an `IllegalStateException` exception is thrown.

Returns

an `int` array of length 4 containing the values p, q, s and d for the optimum model.

getOutlierFreeForecast

```
public double[] getOutlierFreeForecast()
```

Description

Returns forecasts for the outlier free series.

Returns

a `double` array of length `nForecast` containing the forecasts for the outlier free series. Forecast origin is the time point of the last observed value in the time series, t_{nObs}. Forecasts are returned for lead times $1,2,\ldots,$nForecast, i.e. time points $t_{\mathrm{nObs}}+1, t_{\mathrm{nObs}}+2, \ldots, t_{\mathrm{nObs}}+\mathrm{nForecast}$. Method `forecast` must be invoked before this method is called. Otherwise, an `IllegalStateException` exception is thrown.

getOutlierFreeSeries

```
public double[] getOutlierFreeSeries()
```

Description

Returns the outlier free series.

Returns

a `double` array containing the original time series with estimated missing values after removal of any outlier effects. One of the `compute` methods must be called before invoking this method. Otherwise, an `IllegalStateException` exception is thrown.

getOutlierStatistics

```
public int[][] getOutlierStatistics()
```

Description

Returns the outlier statistics.

Returns

a `double` array of length `nOutliers` by 2, where `nOutliers` is the number of detected outliers, containing the outlier statistics. The first column contains the time at which the outlier was observed (time ranging from `times[0]` to `times[times.length - 1]`) and the second column contains an identifier indicating the type of outlier observed. Outlier types fall into one of five categories:

Identifier	Outlier type
INNOVATIONAL	Innovational Outliers (IO)
ADDITIVE	Additive Outliers (AO)
LEVEL_SHIFT	Level Shift Outliers (LS)
TEMPORARY_CHANGE	Temporary Change Outliers (TC)
UNABLE_TO_IDENTIFY	Unable to Identify (UI)

If the number of detected outliers equals zero, then an array of length zero is returned.

getPsiWeights

```
public double[] getPsiWeights()
```

Description

Returns the ψ weights of the infinite order moving average form of the model.

Returns

a `double` array of length `nForecast` containing the ψ weights of the infinite order moving average form of the optimum model for the outlier free series. Method `forecast` must be invoked before this method is called. Otherwise, an `IllegalStateException` exception is thrown.

getResidualStandardError

```
public double getResidualStandardError()
```

Description

Returns the residual standard error of the outlier free series. Note that one of the `compute` methods must be invoked first before calling this method. Otherwise, an `IllegalStateException` exception is thrown.

Returns

a `double` scalar containing the standard error of the outlier free series.

getResiduals

```
public double[] getResiduals()
```

Description

Returns the residuals. Note that one of the `compute` methods must be invoked first before invoking this method. Otherwise, `getResiduals` throws an `IllegalStateException` exception.

Returns

a `double` array containing the residuals for the outlier and gap free series at the final parameter estimation point.

setAccuracyTolerance

```
public void setAccuracyTolerance(double epsilon)
```

Description

Sets the tolerance value controlling the accuracy of the parameter estimates.

Parameter

`epsilon` – a `double` scalar, a positive tolerance value controlling the accuracy of parameter estimates during outlier detection. Default: `epsilon = 0.001`.

setConfidence

```
public void setConfidence(double confidence)
```

Description

Sets the confidence level for calculating confidence limit deviations returned by `getDeviations`.

Parameter

`confidence` – a `double` scalar specifying the confidence level used in computing forecast confidence intervals. 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`.

setCriticalValue

```
public void setCriticalValue(double critical)
```

Description

Sets the critical value used as a threshold during outlier detection.

Parameter

`critical` – a `double` scalar, the critical value used as a threshold for the statistics used in the outlier detection. `critical` must be greater than zero. Default: `critical = 3.0`.

setDelta

```
public void setDelta(double delta)
```

Description

Sets the dampening effect parameter.

Parameter

`delta` – a `double` scalar, the dampening effect parameter used in the detection of a Temporary Change Outlier (TC). `delta` must be greater than 0 and less than 1. Default: `delta = 0.7`.

setDifferenceOrders

```
public void setDifferenceOrders(int[] orders)
```

Description

Defines the orders of the periodic differences used in the determination of the optimum model.

Parameter

`orders` – an `int` array containing all possible orders for each difference given in `periods`. All elements in `orders` must be non-negative. By default, `orders` is a one-element array with `orders[0]` = 0.

setMaximumARLag

```
public void setMaximumARLag(int maxARLag)
```

Description

Defines the maximum AR lag used in the determination of the optimum (s,d) combination of method `compute(int[] arOrders, int[] maOrders)`.

Parameter

`maxARLag` – a scalar `int`, the maximum AR lag used in the computation of the optimum (s,d) combination for method `compute(int[] arOrders, int[] maOrders)`. It is required that `maxARLag` is greater than zero and smaller than the original series after replacement of potential missing values. By default, `maxARLag` = 10.

setModelSelectionCriterion

```
public void setModelSelectionCriterion(int infoCriterion)
```

Description

Sets the model selection criterion.

Parameter

`infoCriterion` – an `int` scalar indicating the model selection criterion used in the search for the optimum model.

infoCriterion	Model selection criterion
AIC	Akaike's information criterion (AIC)
AICC	Akaike's corrected information criterion (AICC)
BIC	Bayesian information criterion (BIC)

By default, AIC is chosen.

setPeriods

```
public void setPeriods(int[] periods)
```

Description

Defines the periods used in the determination of the optimum model.

Parameter

`periods` – an `int` array containing all possible periods that can be applied to the original series after insertion of missing values. All elements of `periods` must be positive. By default, `periods` is a one-element array with `periods[0]` = 1;

setRelativeError

```
public void setRelativeError(double relativeError)
```

Description

Sets the stopping criterion for use in the nonlinear equation solver.

Parameter

`relativeError` – a `double` positive scalar containing the stopping criterion for use in the nonlinear equation solver used in the least-squares algorithm. Default: `relativeError = 2.2204460492503131e-012`.

Example 1: Determination of an optimum AR(p) model

This example uses time series LNU03327709 from the US Department of Labor, Bureau of Labor Statistics. It contains the unadjusted special unemployment rate, taken monthly from January 1994 through September 2005. The values 01/2004 - 03/2005 are used by class autoARIMA for outlier detection and parameter estimation. In this example, method 1, without seasonal adjustment, is chosen to find an appropriate AR(p) model. A forecast is done for the following six months and compared with the actual values 04/2005 - 09/2005.

```
import java.util.*;
import com.imsl.stat.*;

public class AutoARIMAEx1 {

    public static void main(String args[]) throws Exception {
        int nOutliers;
        double aic, RSE, constant;
        int[] optimumModel;
        int[][] outlierStatistics;
        double[] outlierForecast, ar, ma;
        double[] psiWeights, probabilityLimits;
        double[] x = {
            12.8, 12.2, 11.9, 10.9, 10.6, 11.3, 11.1, 10.4, 10.0, 9.7, 9.7, 9.7,
            11.1, 10.5, 10.3, 9.8, 9.8, 10.4, 10.4, 10.0, 9.7, 9.3, 9.6, 9.7,
            10.8, 10.7, 10.3, 9.7, 9.5, 10.0, 10.0, 9.3, 9.0, 8.8, 8.9, 9.2,
            10.4, 10.0, 9.6, 9.0, 8.5, 9.2, 9.0, 8.6, 8.3, 7.9, 8.0, 8.2,
            9.3, 8.9, 8.9, 7.7, 7.6, 8.4, 8.5, 7.8, 7.6, 7.3, 7.2, 7.3,
            8.5, 8.2, 7.9, 7.4, 7.1, 7.9, 7.7, 7.2, 7.0, 6.7, 6.8, 6.9,
            7.8, 7.6, 7.4, 6.6, 6.8, 7.2, 7.2, 7.0, 6.6, 6.3, 6.8, 6.7,
            8.1, 7.9, 7.6, 7.1, 7.2, 8.2, 8.1, 8.1, 8.2, 8.7, 9.0, 9.3,
            10.5, 10.1, 9.9, 9.4, 9.2, 9.8, 9.9, 9.5, 9.0, 9.0, 9.4, 9.6,
            11.0, 10.8, 10.4, 9.8, 9.7, 10.6, 10.5, 10.0, 9.8, 9.5, 9.7, 9.6,
            10.9, 10.3, 10.4, 9.3, 9.3, 9.8, 9.8, 9.3, 8.9, 9.1, 9.1, 9.1,
            10.2, 9.9, 9.4
        };
        double[] exactForecast = {8.7, 8.6, 9.3, 9.1, 8.8, 8.5};
        int[] times = {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
            13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
            25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
            37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
            49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
            61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
            73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
            85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
            97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
            109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
            121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132,
            133, 134, 135
        };

        AutoARIMA autoArima = new AutoARIMA(times, x);
        autoArima.setCriticalValue(3.8);
        autoArima.compute(5);
        autoArima.forecast(6);
```

```
        nOutliers = autoArima.getNumberOfOutliers();
        aic = autoArima.getAIC();
        optimumModel = autoArima.getOptimumModelOrder();
        outlierStatistics = autoArima.getOutlierStatistics();
        RSE = autoArima.getResidualStandardError();
        outlierForecast = autoArima.getForecast();
        psiWeights = autoArima.getPsiWeights();
        probabilityLimits = autoArima.getDeviations();
        constant = autoArima.getConstant();
        ar = autoArima.getAR();
        ma = autoArima.getMA();

        System.out.printf("%nMethod 1: Automatic AR model selection"
                + ", no differencing%n");
        System.out.printf("%nOptimum Model: p=%d, q=%d, s=%d, d=%d%n",
                optimumModel[0], optimumModel[1],
                optimumModel[2], optimumModel[3]);
        System.out.printf("%nNumber of outliers:%3d%n%n", nOutliers);
        System.out.printf("Outlier statistics:%n");
        System.out.printf(" Time%4sType%n", " ");
        for (int i = 0; i < nOutliers; i++) {
            System.out.printf("%5d%8d%n", outlierStatistics[i][0],
                    outlierStatistics[i][1]);
        }
        System.out.printf(Locale.ENGLISH, "%nAIC:%12.6f%n", aic);
        System.out.printf(Locale.ENGLISH, "RSE:%12.6f%n%n", RSE);
        System.out.printf("%6sParameters%n", " ");
        System.out.printf(Locale.ENGLISH, " constant:%12.6f%n", constant);
        for (int i = 0; i < ar.length; i++) {
            System.out.printf(Locale.ENGLISH, " ar[%d]:%15.6f%n", i, ar[i]);
        }
        for (int i = 0; i < ma.length; i++) {
            System.out.printf(Locale.ENGLISH, " ma[%d]:%15.6f%n", i, ma[i]);
        }

        System.out.printf("%n%n%6s* * * Forecast Table * * *%n", " ");
        System.out.printf("%2sExact%3sforecast%5slimits%8spsi%n",
                " ", " ", " ", " ");
        for (int i = 0; i < outlierForecast.length; i++) {
            System.out.printf(Locale.ENGLISH, "%7.4f%11.4f%11.4f%11.4f%n",
                    exactForecast[i], outlierForecast[i],
                    probabilityLimits[i], psiWeights[i]);
        }
    }
}
```

Output

```
Method 1: Automatic AR model selection, no differencing

Optimum Model: p=5, q=0, s=1, d=0

Number of outliers:  7
```

```
Outlier statistics:
 Time    Type
    8       2
   13       0
   37       3
   85       0
   97       0
  109       0
  121       0

AIC:  371.104676
RSE:    0.359632

      Parameters
 constant:    0.097541
 ar[0]:       0.891872
 ar[1]:      -0.123830
 ar[2]:      -0.138262
 ar[3]:       0.135621
 ar[4]:       0.224112


      * * * Forecast Table * * *
  Exact   forecast     limits       psi
 8.7000     9.1076     0.7049     0.8919
 8.6000     9.0993     0.9445     0.6716
 9.3000     9.4032     1.0565     0.3503
 9.1000     9.5806     1.0849     0.2416
 8.8000     9.5506     1.0982     0.4243
 8.5000     9.3932     1.1382     0.5910
```

Example 2: Determination of an optimum ARIMA model via Grid search

This is the same as Example 1, except now class `autoARIMA` uses Method 2 with a possible seasonal adjustment. As a result, the unadjusted model with `p = 3, q = 2, s = 1, d = 0` is chosen as optimum.

```
import java.util.*;
import com.imsl.stat.*;

public class AutoARIMAEx2 {

    public static void main(String args[]) throws Exception {
        int nOutliers;
        double aic, RSE, constant;
        int[] optimumModel;
        int[][] outlierStatistics;
        double[] outlierForecast, ar, ma;
        double[] psiWeights, probabilityLimits;
        int[] arOrders = {0, 1, 2, 3};
        int[] maOrders = {0, 1, 2, 3};
```

```
int[] periods = {1, 2};
int[] orders = {0, 1, 2};
double[] x = {
    12.8, 12.2, 11.9, 10.9, 10.6, 11.3, 11.1, 10.4, 10.0, 9.7, 9.7, 9.7,
    11.1, 10.5, 10.3, 9.8, 9.8, 10.4, 10.4, 10.0, 9.7, 9.3, 9.6, 9.7,
    10.8, 10.7, 10.3, 9.7, 9.5, 10.0, 10.0, 9.3, 9.0, 8.8, 8.9, 9.2,
    10.4, 10.0, 9.6, 9.0, 8.5, 9.2, 9.0, 8.6, 8.3, 7.9, 8.0, 8.2,
    9.3, 8.9, 8.9, 7.7, 7.6, 8.4, 8.5, 7.8, 7.6, 7.3, 7.2, 7.3,
    8.5, 8.2, 7.9, 7.4, 7.1, 7.9, 7.7, 7.2, 7.0, 6.7, 6.8, 6.9,
    7.8, 7.6, 7.4, 6.6, 6.8, 7.2, 7.2, 7.0, 6.6, 6.3, 6.8, 6.7,
    8.1, 7.9, 7.6, 7.1, 7.2, 8.2, 8.1, 8.1, 8.2, 8.7, 9.0, 9.3,
    10.5, 10.1, 9.9, 9.4, 9.2, 9.8, 9.9, 9.5, 9.0, 9.0, 9.4, 9.6,
    11.0, 10.8, 10.4, 9.8, 9.7, 10.6, 10.5, 10.0, 9.8, 9.5, 9.7, 9.6,
    10.9, 10.3, 10.4, 9.3, 9.3, 9.8, 9.8, 9.3, 8.9, 9.1, 9.1, 9.1,
    10.2, 9.9, 9.4
};
double[] exactForecast = {8.7, 8.6, 9.3, 9.1, 8.8, 8.5};
int[] times = {
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
    13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
    25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
    37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
    49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
    61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
    73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
    85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
    97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
    109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
    121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132,
    133, 134, 135
};

AutoARIMA autoArima = new AutoARIMA(times, x);
autoArima.setCriticalValue(3.8);
autoArima.setMaximumARLag(5);
autoArima.setPeriods(periods);
autoArima.setDifferenceOrders(orders);
autoArima.compute(arOrders, maOrders);
autoArima.forecast(6);

nOutliers = autoArima.getNumberOfOutliers();
aic = autoArima.getAIC();
optimumModel = autoArima.getOptimumModelOrder();
outlierStatistics = autoArima.getOutlierStatistics();
RSE = autoArima.getResidualStandardError();
outlierForecast = autoArima.getForecast();
psiWeights = autoArima.getPsiWeights();
probabilityLimits = autoArima.getDeviations();
constant = autoArima.getConstant();
ar = autoArima.getAR();
ma = autoArima.getMA();

System.out.printf("%nMethod 2: Grid search with "
        + "differencing%n");
System.out.printf("%nOptimum Model: p=%d, q=%d, s=%d, d=%d%n",
        optimumModel[0], optimumModel[1],
```

```
                optimumModel[2], optimumModel[3]);
        System.out.printf("%nNumber of outliers:%3d%n%n", nOutliers);
        System.out.printf("Outlier statistics:%n");
        System.out.printf(" Time%4sType%n", " ");
        for (int i = 0; i < nOutliers; i++) {
            System.out.printf("%5d%8d%n", outlierStatistics[i][0],
                    outlierStatistics[i][1]);
        }
        System.out.printf(Locale.ENGLISH, "%nAIC:%12.6f%n", aic);
        System.out.printf(Locale.ENGLISH, "RSE:%12.6f%n%n", RSE);
        System.out.printf("%5sParameters%n", " ");
        System.out.printf(Locale.ENGLISH, " constant:%10.6f%n", constant);
        for (int i = 0; i < ar.length; i++) {
            System.out.printf(" ar[%d]:%13.6f%n", i, ar[i]);
        }
        for (int i = 0; i < ma.length; i++) {
            System.out.printf(" ma[%d]:%13.6f%n", i, ma[i]);
        }
        System.out.printf("%n%n%6s* * * Forecast Table * * *%n", " ");
        System.out.printf("%2sExact%3sforecast%5slimits%8spsi%n",
                " ", " ", " ", " ");
        for (int i = 0; i < outlierForecast.length; i++) {
            System.out.printf(Locale.ENGLISH, "%7.4f%11.4f%11.4f%11.4f%n",
                    exactForecast[i], outlierForecast[i],
                    probabilityLimits[i], psiWeights[i]);
        }
    }
}
```

Output

```
Method 2: Grid search with differencing

Optimum Model: p=3, q=2, s=1, d=0

Number of outliers:  1

Outlier statistics:
 Time    Type
  109       0

AIC:  408.108176
RSE:    0.412456

     Parameters
 constant:  0.554459
 ar[0]:     1.940615
 ar[1]:    -1.898025
 ar[2]:     0.897791
 ma[0]:     1.115803
 ma[1]:    -0.911902


      * * * Forecast Table * * *
```

```
  Exact   forecast     limits        psi
 8.7000     9.1085     0.8084     0.8248
 8.6000     9.1715     1.0479     0.6145
 9.3000     9.5039     1.1597     0.5248
 9.1000     9.7677     1.2349     0.5926
 8.8000     9.7051     1.3245     0.7056
 8.5000     9.3817     1.4421     0.7157
```

Example 3: Specified ARIMA model

This example is the same as example 2 but now method 3 with the optimum model parameters `p = 3, q = 2, s = 1, d = 0` from Example 2 is chosen for outlier detection and forecasting.

```
import java.util.*;
import com.imsl.stat.*;

public class AutoARIMAEx3 {

    public static void main(String args[]) throws Exception {
        int nOutliers;
        double aic, RSE, constant;
        int[] optimumModel;
        int[][] outlierStatistics;
        double[] outlierForecast, ar, ma;
        double[] psiWeights, probabilityLimits;
        double[] x = {
            12.8, 12.2, 11.9, 10.9, 10.6, 11.3, 11.1, 10.4, 10.0, 9.7, 9.7, 9.7,
            11.1, 10.5, 10.3, 9.8, 9.8, 10.4, 10.4, 10.0, 9.7, 9.3, 9.6, 9.7,
            10.8, 10.7, 10.3, 9.7, 9.5, 10.0, 10.0, 9.3, 9.0, 8.8, 8.9, 9.2,
            10.4, 10.0, 9.6, 9.0, 8.5, 9.2, 9.0, 8.6, 8.3, 7.9, 8.0, 8.2,
            9.3, 8.9, 8.9, 7.7, 7.6, 8.4, 8.5, 7.8, 7.6, 7.3, 7.2, 7.3,
            8.5, 8.2, 7.9, 7.4, 7.1, 7.9, 7.7, 7.2, 7.0, 6.7, 6.8, 6.9,
            7.8, 7.6, 7.4, 6.6, 6.8, 7.2, 7.2, 7.0, 6.6, 6.3, 6.8, 6.7,
            8.1, 7.9, 7.6, 7.1, 7.2, 8.2, 8.1, 8.1, 8.2, 8.7, 9.0, 9.3,
            10.5, 10.1, 9.9, 9.4, 9.2, 9.8, 9.9, 9.5, 9.0, 9.0, 9.4, 9.6,
            11.0, 10.8, 10.4, 9.8, 9.7, 10.6, 10.5, 10.0, 9.8, 9.5, 9.7, 9.6,
            10.9, 10.3, 10.4, 9.3, 9.3, 9.8, 9.8, 9.3, 8.9, 9.1, 9.1, 9.1,
            10.2, 9.9, 9.4
        };
        double[] exactForecast = {8.7, 8.6, 9.3, 9.1, 8.8, 8.5};
        int[] times = {
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
            13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
            25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36,
            37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48,
            49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
            61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
            73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
            85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96,
            97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108,
            109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120,
            121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132,
            133, 134, 135
        };
```

```
        AutoARIMA autoArima = new AutoARIMA(times, x);
        autoArima.setCriticalValue(3.8);
        autoArima.compute(3, 2, 1, 0);
        autoArima.forecast(6);

        nOutliers = autoArima.getNumberOfOutliers();
        aic = autoArima.getAIC();
        optimumModel = autoArima.getOptimumModelOrder();
        outlierStatistics = autoArima.getOutlierStatistics();
        RSE = autoArima.getResidualStandardError();
        outlierForecast = autoArima.getForecast();
        psiWeights = autoArima.getPsiWeights();
        probabilityLimits = autoArima.getDeviations();
        constant = autoArima.getConstant();
        ar = autoArima.getAR();
        ma = autoArima.getMA();

        System.out.printf("%nMethod 3: Specified ARIMA model%n");
        System.out.printf("%nOptimum Model: p=%d, q=%d, s=%d, d=%d%n",
                optimumModel[0], optimumModel[1],
                optimumModel[2], optimumModel[3]);
        System.out.printf("%nNumber of outliers:%3d%n%n", nOutliers);
        System.out.printf("Outlier statistics:%n");
        System.out.printf(" Time%4sType%n", " ");
        for (int i = 0; i < nOutliers; i++) {
            System.out.printf("%5d%8d%n", outlierStatistics[i][0],
                    outlierStatistics[i][1]);
        }
        System.out.printf(Locale.ENGLISH, "%nAIC:%12.6f%n", aic);
        System.out.printf(Locale.ENGLISH, "RSE%13.6f%n%n", RSE);
        System.out.printf("%5sParameters%n", " ");
        System.out.printf(Locale.ENGLISH, " constant:%10.6f%n", constant);
        for (int i = 0; i < ar.length; i++) {
            System.out.printf(Locale.ENGLISH, " ar[%d]:%13.6f%n", i, ar[i]);
        }
        for (int i = 0; i < ma.length; i++) {
            System.out.printf(Locale.ENGLISH, " ma[%d]:%13.6f%n", i, ma[i]);
        }
        System.out.printf("%n%n%6s* * * Forecast Table * * *%n", " ");
        System.out.printf("%2sExact%3sforecast%5slimits%8spsi%n",
                " ", " ", " ", " ");
        for (int i = 0; i < outlierForecast.length; i++) {
            System.out.printf(Locale.ENGLISH, "%7.4f%11.4f%11.4f%11.4f%n",
                    exactForecast[i], outlierForecast[i],
                    probabilityLimits[i], psiWeights[i]);
        }
    }
}
```

Output

```
Method 3: Specified ARIMA model
```

```
Optimum Model: p=3, q=2, s=1, d=0

Number of outliers:  1

Outlier statistics:
 Time    Type
  109       0

AIC:  408.108176
RSE     0.412456

     Parameters
 constant:  0.554459
 ar[0]:     1.940615
 ar[1]:    -1.898025
 ar[2]:     0.897791
 ma[0]:     1.115803
 ma[1]:    -0.911902


      * * * Forecast Table * * *
  Exact   forecast     limits       psi
 8.7000     9.1085     0.8084     0.8248
 8.6000     9.1715     1.0479     0.6145
 9.3000     9.5039     1.1597     0.5248
 9.1000     9.7677     1.2349     0.5926
 8.8000     9.7051     1.3245     0.7056
 8.5000     9.3817     1.4421     0.7157
```

AutoARIMA.NoAcceptableModelFoundException class

`static public class com.imsl.stat.AutoARIMA.NoAcceptableModelFoundException`
`extends com.imsl.IMSLException`

No appropriate ARIMA model could be found.

Constructors

AutoARIMA.NoAcceptableModelFoundException

`public AutoARIMA.NoAcceptableModelFoundException(String message)`

Description

Constructs a `NoAcceptableModelFoundException` exception with the specified detail message. A

detail message is a `String` that describes this particular exception.

Parameter

`message` – the detail message.

AutoARIMA.NoAcceptableModelFoundException

```
public AutoARIMA.NoAcceptableModelFoundException(String key, Object[]
arguments)
```

Description

Constructs a `NoAcceptableModelFoundException` exception 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.

CrossCorrelation class

```
public class com.imsl.stat.CrossCorrelation implements Serializable, Cloneable
```

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 μ_X of the time series $\{X_t\}$ where

$$\hat{\mu}_X = \begin{cases} \mu_X & \text{for } \mu_X \text{ known} \\ \frac{1}{n}\sum_{t=1}^{n} X_t & \text{for } \mu_X \text{ unknown} \end{cases}$$

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

$$\hat{\sigma}_{XY}(k) = \begin{cases} \frac{1}{n}\sum_{t=1}^{n-k}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = 0, 1, \ldots, K \\ \frac{1}{n}\sum_{t=1-k}^{n}(X_t - \hat{\mu}_X)(Y_{t+k} - \hat{\mu}_Y) & k = -1, -2, \ldots, -K \end{cases}$$

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

$$\text{var}\{\hat{\rho}_{XY}(k)\} = \frac{1}{n-k}\sum_{i=-\infty}^{\infty}\rho_X(i)\rho_Y(i) \quad k \ge 0$$

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.

Fields

BARTLETTS_FORMULA

```
static final public int BARTLETTS_FORMULA
```

Indicates standard error computation using Bartlett's formula.

BARTLETTS_FORMULA_NOCC

```
static final public 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
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
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
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
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
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
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.

Parameter

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 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 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`.

Parameter

`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`.

Parameter

`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 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(CrossCorrelation.BARTLETTS_FORMULA));
        new PrintMatrix("Standard Errors using Bartlett #2 are:  ").
                print(cc.getStandardErrors(
                                CrossCorrelation.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
```

CrossCorrelation.NonPosVariancesException class

```
static public class com.imsl.stat.CrossCorrelation.NonPosVariancesException
extends com.imsl.IMSLException
```

The problem is ill-conditioned.

Constructors

CrossCorrelation.NonPosVariancesException

```
public CrossCorrelation.NonPosVariancesException(String message)
```

Description

Constructs a `NonPosVariancesException` object.

Parameter

`message` – a `String` containing the error message

CrossCorrelation.NonPosVariancesException

```
public CrossCorrelation.NonPosVariancesException(String key, Object[]
arguments)
```

Description

Constructs a `NonPosVariancesException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

Difference class

```
public class com.imsl.stat.Difference implements Serializable, Cloneable
```

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.

Constructor

Difference

`public Difference()`

Description

Constructor for `Difference`.

Methods

compute

```
final public double[] compute(double[] z, int[] periods) throws
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.

Parameter

`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. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

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

Parameter

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};
        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);
        int 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
```

GARCH class

```
public class com.imsl.stat.GARCH implements Serializable, Cloneable
```

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 $\alpha_i, i = 1, \ldots, q > and the \beta_j, j = 1, \ldots, p > parameters will be called the ARCH and GARCH coefficients, respectively. When \beta_i = 0, i = 1, 2, \ldots, p$, the above model reduces to ARCH(q) which was proposed by Engle (1982). That is, a GARCH(0,q) is equivalent to the ARCH(q) model. The non-negativity conditions on the parameters imply a nonnegative variance and the condition on the sum of the β_i's and α_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` (p. 429) , 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).

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 GARCH parameters.

`q` – An `int` scalar containing the number of ARCH 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.

Exception

`IllegalArgumentException` is thrown if the dimensions of `y`, and `xguess` are not consistent.

Methods

compute

```
final public void compute() throws GARCH.ConstrInconsistentException,
GARCH.EqConstrInconsistentException, GARCH.NoVectorXException,
GARCH.TooManyIterationsException, GARCH.VarsDeterminedException
```

Description

Computes estimates of the parameters of a GARCH(p,q) model.

Exceptions

`ConstrInconsistentException` is thrown if the equality constraints are inconsistent.

`EqConstrInconsistentException` is thrown if the equality constraints and the bounds on the variables are found to be inconsistent.

`NoVectorXException` is thrown if no vector X satisfies all of the constraints.

`TooManyIterationsException` is thrown if the number of function evaluations exceeded 1000.

`VarsDeterminedException` is thrown if the variables are determined by the equality constraints.

getARCH

```
public double[] getARCH()
```

Description

Returns the estimated values of the ARCH coefficients. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` array of size `q` containing the estimated values of the moving average (ARCH) coefficients.

---

getAkaike

```
public double getAkaike()
```

Description

Returns the value of Akaike Information Criterion evaluated at the estimated parameter array. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

Returns

a `double` scalar containing the value of Akaike Information Criterion evaluated at the estimated parameter array.

---

getGARCH

```
public double[] getGARCH()
```

Description

Returns the estimated values of the GARCH coefficients. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` array of size `p` containing the estimated values of autoregressive (GARCH) parameters.

---

getLogLikelihood

```
public double getLogLikelihood()
```

Description

Returns the value of Log-likelihood function evaluated at the estimated parameter array. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `0`.

Returns

a `double` scalar containing the value of Log-likelihood function evaluated at the estimated parameter array.

---

getSigma

```
public double getSigma()
```

Description

Returns the estimated value of sigma squared. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the return value is `NaN`.

Returns

a `double` scalar containing the estimated value of sigma squared.

getVarCovarMatrix

`public double[][] getVarCovarMatrix()`

Description

Returns the variance-covariance matrix. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

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`. Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` array of size `1 + q + p` containing the estimated values of sigma squared, the ARCH parameters, and the GARCH 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.

Parameter

`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 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;
        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("ARCH estimate is ").print(garch.getARCH());
        new PrintMatrix("GARCH estimate is ").print(garch.getGARCH());
        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

ARCH estimate is
     0
0  0.245

GARCH estimate is
     0
0  0.337
1  0.31

Log-likelihood function value is -2,707.072
Akaike Information Criterion value is 5,422.144
```

GARCH.VarsDeterminedException class

`static public class com.imsl.stat.GARCH.VarsDeterminedException extends com.imsl.IMSLException`

The variables are determined by the equality constraints.

Constructors

GARCH.VarsDeterminedException

```
public GARCH.VarsDeterminedException(String message)
```

Description

Constructs a `VarsDeterminedException` object.

Parameter

`message` – a `String` containing the error message

GARCH.VarsDeterminedException

```
public GARCH.VarsDeterminedException(String key, Object[] arguments)
```

Description

Constructs a `VarsDeterminedException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

GARCH.TooManyIterationsException class

```
static public class com.imsl.stat.GARCH.TooManyIterationsException extends
com.imsl.IMSLException
```

Number of function evaluations exceeded 1000.

Constructors

GARCH.TooManyIterationsException

```
public GARCH.TooManyIterationsException(String message)
```

Description

Constructs a `TooManyIterationsException` object.

Parameter

`message` – a `String` containing the error message

GARCH.TooManyIterationsException

```
public GARCH.TooManyIterationsException(String key, Object[] arguments)
```

Description

Constructs a `TooManyIterationsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

GARCH.NoVectorXException class

```
static public class com.imsl.stat.GARCH.NoVectorXException extends
com.imsl.IMSLException
```

No vector X satisfies all of the constraints.

Constructors

GARCH.NoVectorXException

```
public GARCH.NoVectorXException(String message)
```

Description

Constructs a `NoVectorXException` object.

Parameter

`message` – a `String` containing the error message

GARCH.NoVectorXException

```
public GARCH.NoVectorXException(String key, Object[] arguments)
```

Description

Constructs a `NoVectorXException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

GARCH.EqConstrInconsistentException class

`static public class com.imsl.stat.GARCH.EqConstrInconsistentException extends com.imsl.IMSLException`

The equality constraints and the bounds on the variables are found to be inconsistent.

Constructors

GARCH.EqConstrInconsistentException

`public GARCH.EqConstrInconsistentException(String message)`

Description

Constructs a `EqConstrInconsistentException` object.

Parameter

`message` – a `String` containing the error message

GARCH.EqConstrInconsistentException

`public GARCH.EqConstrInconsistentException(String key, Object[] arguments)`

Description

Constructs a `EqConstrInconsistentException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

GARCH.ConstrInconsistentException class

`static public class com.imsl.stat.GARCH.ConstrInconsistentException extends com.imsl.IMSLException`

The equality constraints are inconsistent.

Constructors

GARCH.ConstrInconsistentException

`public GARCH.ConstrInconsistentException(String message)`

Description

Constructs a `ConstrInconsistentException` object.

Parameter

`message` – a `String` containing the error message

GARCH.ConstrInconsistentException

`public GARCH.ConstrInconsistentException(String key, Object[] arguments)`

Description

Constructs a `ConstrInconsistentException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

KalmanFilter class

`public class com.imsl.stat.KalmanFilter implements Serializable, Cloneable`

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, σ^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, σ^2. The maximum likelihood estimate of σ^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 σ^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 θ. 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 θ. 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,

$$\sum_{k=1}^{m} \ln[\det(H_k)]$$

(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 θ and σ^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 θ, consequently,

$$\hat{\sigma}^2(\theta)$$

can be substituted for σ^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).

Constructors

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 matrix of size b.length by b.length such that covb * σ^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 * σ^2 is the variance-covariance matrix of the observations.

Exception

IllegalArgumentException is thrown if the dimensions of b, and covb are not consistent.

Methods

filter

```
final public 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 matrix of size b.length by b.length such that covb * σ^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 * σ^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` * σ^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 σ^2 is given by `sumOfSquares / rank`.

resetQ

```
public void resetQ()
```

Description

Removes the Q matrix.

resetTransitionMatrix

```
public void resetTransitionMatrix()
```

Description

Removes the transition matrix.

resetUpdate

```
public void resetUpdate()
```

Description

Do not perform computation of the update equations.

setQ

```
public void setQ(double[][] q)
```

Description

Sets the Q matrix.

Parameter

`q` – A `double` matrix containing the `b.length by b.length` matrix such that `q` * σ^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.

Parameter

`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.

Parameter

`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 * σ^2 is the variance-covariance matrix of errors in the observation equation. σ^2 is a positive unknown scalar. Only elements in the upper triangle of r are referenced.

Example 1: 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_ks are identically and independently distributed normal with mean 0 and variance σ^2, the w_ks 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 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));

        KalmanFilter kalman
                = new KalmanFilter(b, covb, rank, ss, logDeterminant);

        for (int i = 0; i < nobs; i++) {
            double y[] = {ydata[i]};
            kalman.update(y, z, r);
            kalman.filter();
            double v[] = kalman.getPredictionError();
            double covv[][] = kalman.getCovV();
            argFormat[0] = new Integer(i);
            argFormat[1] = new Integer(i);
            argFormat[2] = new Double(kalman.getStateVector()[0]);
            argFormat[3] = new Double(kalman.getCovB()[0][0]);
            argFormat[4] = new Integer(kalman.getRank());
            argFormat[5] = new Double(kalman.getSumOfSquares());
            argFormat[6] = new Double(kalman.getLogDeterminant());
            argFormat[7] = new Double(v[0]);
            argFormat[8] = new Double(covv[0][0]);
            System.out.println(mf.format(argFormat));
            kalman.resetUpdate();

            kalman.setTransitionMatrix(t);
            kalman.setQ(q);
            kalman.filter();
            argFormat[0] = new Integer(i + 1);
            argFormat[1] = new Integer(i);
            argFormat[2] = new Double(kalman.getStateVector()[0]);
            argFormat[3] = new Double(kalman.getCovB()[0][0]);
            argFormat[4] = new Integer(kalman.getRank());
            argFormat[5] = new Double(kalman.getSumOfSquares());
            argFormat[6] = new Double(kalman.getLogDeterminant());
            argFormat[7] = new Double(v[0]);
            argFormat[8] = new Double(covv[0][0]);
            System.out.println(mf.format(argFormat));
            kalman.resetTransitionMatrix();
            kalman.resetQ();
        }
    }
}
```

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
```

Example 2: Kalman Filter Maximum Likelihood Estimate

KalmanFilter is used with the `MinUnconMultivar` class to find a maximum likelihood estimate of the parameter θ in an MA(1) time series represented by $y_k = \varepsilon_k - \theta\varepsilon_{k-1}$. The input is 200 observations from an MA(1) time series with $\theta = 0.5$ and $\sigma^2 = 1$. The MA(1) time series is cast as a state-space model of the following form (see Harvey 1981, pages 103-104, 112):

$$y_k = \begin{pmatrix} 1 & 0 \end{pmatrix} b_k$$

$$b_k = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} b_{k-1} + w_k$$

where the two-dimensional w_ks are independently distributed bivariate normal with mean 0 and variance $\sigma^2 Q_k$ and

$$Q_1 = \begin{pmatrix} 1+\theta^2 & -\theta \\ -\theta & \theta^2 \end{pmatrix}$$

$$Q_k = \begin{pmatrix} 1 & -\theta \\ -\theta & \theta^2 \end{pmatrix} \quad k = 2, 3, ..., 200$$

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

public class KalmanFilterEx2 implements MinUnconMultiVar.Function {

    public double f(double[] theta) {
        int nobs = 200, rank = 0;
        double ss = 0.0;
        double logDeterminant = 0.0;
        double res;
        double[][] covb = new double[2][2];
        double[][] q = new double[2][2];
        double[][] r = {{0.0}};
        double[] b = {0.0, 0.0};
        double[][] z = {{1.0, 0.0}};
        double[][] t = {{0.0, 1.0},
        {0.0, 0.0}};
        double[] y = new double[1];
        double[] ydata = {
```

```
            0.057466, -0.459067, 1.236446, -1.864825, 0.951507,
            -1.489367, -0.864401, 0.302984, -0.376017, 0.610208,
            -1.701583, 2.271734, -0.917110, 0.582576, 1.018681,
            0.107443, -0.557858, 0.502818, 0.858002, -1.417659,
            0.839981, 0.047021, 0.404448, 0.383749, -0.383227,
            1.179607, -1.154339, 1.927541, 0.996344, 1.019415,
            -0.816815, -0.100467, -0.125334, -0.068065, -1.891685,
            0.657241, -1.070823, 1.023510, 2.672653, -2.141434,
            -1.232266, 0.925311, -0.201665, -1.325580, -0.086926,
            -0.647157, 1.314749, -0.085708, 1.430485, 0.304040,
            0.305559, 1.669671, 1.004800, -1.678350, 0.631133,
            0.502284, 0.247378, -1.345484, 0.994982, 1.145546,
            -1.248822, 0.616784, -2.127822, 2.264872, -1.590343,
            0.365785, -1.056652, 0.969750, 1.028274, 0.332050,
            0.430686, 0.364553, 0.482446, -0.303248, 1.581931,
            -0.140942, -0.265280, -0.939284, 0.464963, -0.778145,
            0.583486, -0.113080, -0.009839, 1.580189, -1.116377,
            1.744513, -0.298106, 0.332944, -0.228859, 1.101747,
            0.772369, -1.608111, 2.671822, -0.504800, -1.647797,
            -0.596313, 0.845472, 0.507869, 0.833377, -0.460099,
            0.416891, -1.139069, 0.159028, -0.193971, -0.154656,
            1.720997, -0.403189, 0.400026, 0.285921, -1.914338,
            1.296864, 1.426898, -0.426181, 0.255961, -1.790193,
            0.721048, 1.150173, -0.980386, 0.940958, 0.313898,
            0.505735, -1.058126, 0.111918, 0.185493, -0.296146,
            -0.104457, 1.151733, 0.683025, -0.714269, -0.787972,
            -1.277062, 1.378816, -0.658115, -0.259860, -2.614008,
            2.251646, -0.006773, -0.738467, -0.260685, 1.896505,
            -0.094919, 0.089954, -1.627679, -1.675018, 0.896835,
            0.498690, -0.368775, 0.131849, -2.060292, 0.272666,
            2.115804, -1.323451, 0.557106, -0.602031, 1.424524,
            -0.107996, -1.580671, 0.672012, -1.668931, 2.474710,
            -1.471518, 1.780317, -0.419588, 2.008474, -1.246855,
            0.231161, 0.706104, -0.474320, -0.705431, 0.599358,
            -2.469494, 2.024374, 0.849572, -2.410618, 2.812321,
            -1.066520, -0.539768, -0.067784, 1.978078, 0.592879,
            -0.184623, -1.403912, -0.995537, 1.727320, -0.313251,
            0.472437, -0.241800, -0.875680, -0.159557, 0.508238,
            -0.116888, -0.981759, -0.472451, 0.847273, -1.713030,
            2.010192, -0.981891, 1.190901, 0.453348, -0.743333
        };
        if (Math.abs(theta[0]) > 1.0) {
            res = 1.0e10;
        } else {
            q[0][0] = 1.0;
            q[0][1] = -theta[0];
            q[1][0] = -theta[0];
            q[1][1] = theta[0] * theta[0];

            covb[0][0] = 1.0 + theta[0] * theta[0];
            covb[0][1] = -theta[0];
            covb[1][0] = -theta[0];
            covb[1][1] = theta[0] * theta[0];

            KalmanFilter kalman
                    = new KalmanFilter(b, covb, rank, ss, logDeterminant);
```

```
            for (int i = 0; i < nobs; i++) {
                y[0] = ydata[i];
                kalman.update(y, z, r);
                kalman.setTransitionMatrix(t);
                kalman.setQ(q);
                kalman.filter();
            }
            ss = kalman.getSumOfSquares();
            logDeterminant = kalman.getLogDeterminant();
            rank = kalman.getRank();
            res = rank * Math.log(ss / rank) + logDeterminant;
        }
        return (res);
    }

    public static void main(String args[]) throws com.imsl.IMSLException {
        MinUnconMultiVar solver = new MinUnconMultiVar(1);
        double[] x = solver.computeMin(new KalmanFilterEx2());
        System.out.println("Maximum likelihood estimate, THETA = " + x[0]);
    }
}
```

Output

```
Maximum likelihood estimate, THETA = 0.4529402135287534
```

MultiCrossCorrelation class

`public class com.imsl.stat.MultiCrossCorrelation implements Serializable, Cloneable`

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, ..., 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, ..., p, j = 1, ..., 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

$$\hat{\rho}_{X_jY_j}(k) = \frac{\hat{\sigma}_{X_jY_j(k)}}{[\hat{\sigma}_{X_i}(0)\hat{\sigma}_{X_j}(0)]^{\frac{1}{2}}} \qquad k = 0, \pm 1, \ldots, \pm K$$

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
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
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
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
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.

Parameter

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.

Parameter

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

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 = 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]);
        }
        double[][][] 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
```

MultiCrossCorrelation.NonPosVariancesException class

```
static public class
com.imsl.stat.MultiCrossCorrelation.NonPosVariancesException extends
com.imsl.IMSLException
```

The problem is ill-conditioned.

Constructors

MultiCrossCorrelation.NonPosVariancesException

`public MultiCrossCorrelation.NonPosVariancesException(String message)`

Description

Constructs a `NonPosVariancesException` object.

Parameter

`message` – a `String` containing the error message

MultiCrossCorrelation.NonPosVariancesException

```
public MultiCrossCorrelation.NonPosVariancesException(String key, Object[]
arguments)
```

Description

Constructs a `NonPosVariancesException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

LackOfFit class

```
public class com.imsl.stat.LackOfFit
```

Performs lack-of-fit test for a univariate time series or transfer function given the appropriate correlation function.

`LackOfFit` may be used to diagnose lack of fit in both ARMA and transfer function models. Typical arguments for these situations are:

Model	`lagMin`	`lagMax`	`npFree`
ARMA (p, q)	1	$\sqrt{\texttt{nObservations}}$	$p + q$
Transfer function	0	$\sqrt{\texttt{nObservations}}$	$r + s$

`LackOfFit` performs a portmanteau lack of fit test for a time series or transfer function containing `nObservations` observations given the appropriate sample correlation function $\hat{\rho}(k)$ for $k = L, L+1, ..., K$ where L = `lagMin` and K = `lagMax`.

The basic form of the test statistic Q is

$$Q = n(n+2)\sum_{k=L}^{K}(n-k)^{-1}\hat{\rho}(k)$$

with $L = 1$ if $\hat{\rho}(k)$ is an autocorrelation function. Given that the model is adequate, Q has a chi-squared distribution with $K-L+1-m$ degrees of freedom where m = `npFree` is the number of parameters estimated in the model. If the mean of the time series is estimated, Woodfield (1990) recommends not including this in the count of the parameters estimated in the model. Thus, for an ARMA(p, q) model set `npFree` = $p + q$ regardless of whether the mean is estimated or not. The original derivation for time series models is due to Box and Pierce (1970) with the above modified version discussed by Ljung and Box (1978). The extension of the test to transfer function models is discussed by Box and Jenkins (1976, pages 394-395).

Methods

compute

`static public double[] compute(int nObservations, double[] correlations, int npFree, int lagMax)`

Description

Performs lack-of-fit test for a univariate time series or transfer function given the appropriate correlation function using a minimum lag of 1.

Parameters

`nObservations` – an `int` containing the number of observations of the stationary time series.

`correlations` – a `double` array of length `lagMax+1` containing the correlation function.

`npFree` – an `int` scalar specifying the number of free parameters in the formulation of the time series model. `npfree` must be greater than or equal to zero and less than `lagMax`. Woodfield (1990) recommends `npFree = p + q`.

`lagMax` – an `int` scalar specifying the maximum lag of the correlation function.

Returns

a `double` array of length 2 with the test statistic, Q, and its *p*-value, *p*. Under the null hypothesis, Q has an approximate chi-squared distribution with `lagMax-lagMin+1-npFree` degrees of freedom.

compute

`static public double[] compute(int nObservations, double[] correlations, int npFree, int lagMax, int lagMin)`

Description

Performs lack-of-fit test for a univariate time series or transfer function given the appropriate correlation function.

Parameters

`nObservations` – an `int` containing the number of observations of the stationary time series.

`correlations` – a `double` array of length `lagMax+1` containing the correlation function.

`npFree` – an `int` scalar specifying the number of free parameters in the formulation of the time series model. `npfree` must be greater than or equal to zero and less than `lagMax`. Woodfield (1990) recommends `npFree = p + q`.

`lagMax` – an `int` scalar specifying the maximum lag of the correlation function.

`lagMin` – an `int` scalar specifying the minimum lag of the correlation function. `lagMin` corresponds to the lower bound of summation in the lack of fit test statistic. Default value is 1.

Returns

a `double` array of length 2 with the test statistic, Q, and its *p*-value, *p*. Under the null hypothesis, Q has an approximate chi-squared distribution with `lagMax-lagMin+1-npFree` degrees of freedom.

Example: Lack Of Fit

Consider the Wölfer Sunspot Data (Anderson 1971, page 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. An ARMA(2,1) with nonzero mean is fitted using the `ARMA` class. The autocorrelations of the residuals are estimated using the `AutoCorrelation` class. Class `LackOfFit` is used to compute a portmanteau lack of fit test using 10 lags. The warning message from `ARMA` in the output can be ignored. (See Example 2 in `ARMA` for a full explanation of the warning message.)

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

public class LackOfFitEx1 {

    public static void main(String args[]) throws Exception {
        int p = 2, q = 1, lagmax = 10, npfree = 4;
        double tolerance = 0.125;

        //  sunspot data for 1770 through 1869
        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
        };

        // Get residuals from ARMA(2,1) for autocorrelation/lack of fit
        ARMA arma = new ARMA(p, q, x);
        arma.setMethod(ARMA.LEAST_SQUARES);
        arma.setConvergenceTolerance(tolerance);
        arma.compute();
        double[] residuals = arma.getResidual();

        // Get autocorrelations from residuals for lack of fit test
        AutoCorrelation ac = new AutoCorrelation(residuals, lagmax);
        double[] correlations = ac.getAutoCorrelations();

        // Get lack of fit test statistic and p-value
        double[] result
                = LackOfFit.compute(x.length, correlations, npfree, lagmax);

        // Print parameter estimates, test statistic, and p-value
        // NOTE: Test Statistic Q follows a Chi-squared dist.
        System.out.println("Lack of Fit Statistic,  Q = " + result[0]);
        System.out.println("P-value of Q = " + result[1]);
    }
}
```

Output

```
Lack of Fit Statistic,  Q = 14.60601806330646
P-value of Q = 0.0235525998617882
```

HoltWintersExponentialSmoothing class

```
public class com.imsl.stat.HoltWintersExponentialSmoothing implements
Serializable, Cloneable
```

Calculates parameters and forecasts using the Holt-Winters Multiplicative or Additive forecasting method for seasonal data.

`HoltWintersExponentialSmoothing` performs the Holt-Winters forecasting method to an equally spaced time series, $\{y_t\}$ where $N =$ `nobs` and $t = 1,\ldots,N$ (or $t = 0, 1 \cdot \text{incy}, 2 \cdot \text{incy},\ldots,K \cdot \text{incy}$ where $K \cdot \text{incy} \leq N$ and $\text{incy} \geq 1$). The Holt-Winters procedure fits three component sequences known as the *level*, the *trend*, and the *seasonal* sequence. There are two formulations, depending on whether the seasonal component of the time series is thought to be additive or multiplicative. The seasonal component depends on the length of the season, `nseason` $= s$, where $s = 2,\ldots,N$.

Holt-Winters Additive Model

$L_t = \alpha(y_t - S_{t-s}) + (1-\alpha)(L_{t-1} + b_{t-1})$	the *level* sequence
$b_t = \beta(L_t - L_{t-1}) + (1-\beta)b_{t-1}$	the *trend* sequence
$S_t = \gamma(y_t - L_t) + (1-\gamma)S_{t-s}$	the *seasonal* sequence
$F_{t+k} = L_t + kb_t + S_{t+k-s}$	the *forecast* sequence

Holt-Winters Multiplicative Model

$L_t = \alpha(y_t/S_{t-s}) + (1-\alpha)(L_{t-1} + b_{t-1})$	the *level* sequence
$b_t = \beta(L_t - L_{t-1}) + (1-\beta)b_{t-1}$	the *trend* sequence
$S_t = \gamma(y_t/L_t) + (1-\gamma)S_{t-s}$	the *seasonal* sequence
$F_{t+k} = L_t + kb_t + S_{t+k-s}$	the *forecast* sequence

Note that without a seasonal component, both the additive and multiplicative formulations reduce to the same methods. (The seasonal sequence is set to 1 for the multiplicative model, and identically 0 for the additive model.)

Default Starting Values

Initial values are required for these sequences. The software allows the user code to define these initial values (see `setInitialValues` method). If they are not provided, then the first two seasons of data are

used:

$$L_s = \frac{1}{s}\sum_{i=1}^{s} y_i, \qquad s > 1$$
$$b_s = \frac{1}{s}\sum_{i=1}^{s} \frac{(y_{i+s}-y_i)}{s}, \qquad s > 1$$
$$S_i = y_i/L_s \text{ or } y_i - L_s, \quad i = 1,\ldots,s \text{ (Multiplicative or Additive)}$$

The smoothing parameters (α, β, γ) are each in the interval [0,1] and can be specified by the user (see `setParameters` method), or automatically set by minimizing the within sample one-step ahead mean squared forecast error. Note that this objective function is not necessarily convex and solutions will depend on starting values. See Chatfield and Yar(1988) for further discussion. Starting values for (α, β, γ) are obtained by minimizing the mean squared error over `nsamples` bootstrap samples of the time series. Experiments suggest that this approach helps prevent poor starting values. Note, that solutions may vary for different settings of the number of random samples, `nsamples`.

The return value of the `compute` method is an (N + 1) by 3 = (`nobs` + 1) by 3 matrix containing the smoothing parameter values on the first row, followed by the calculated level, trend, and seasonal sequences. When N = `nobs` and s = `nseason`, the format of the return value is as follows:

Series Storage			
Row	Value		
0	$\hat{\alpha}$	$\hat{\beta}$	$\hat{\gamma}$
1	0	0	S_1
2	0	0	S_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
s	L_s	b_s	S_s
$s+1$	L_{s+1}	b_{s+1}	S_{s+1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
N	L_N	b_N	S_N

If one of the nonseasonal options is specifed, the return value will be (`nobs` + 1) by 2 or (`nobs` + 1) by 1 accordingly.

The variance-covariance matrix for the parameters (α, β, γ) is

$$\text{cov} = \frac{\sum e_i^2}{N - 2s - 3}\left(J^T J\right)^{-1}$$

where e_i is the one-step-ahead forecast error, and J is the Jacobian matrix of the series using the forecast model and calculating forecasts one step ahead of each datum. Prediction intervals are calculated following Chatfield and Yar (1991).

Constructor

HoltWintersExponentialSmoothing

```
public HoltWintersExponentialSmoothing(int nseason, double[] y)
```

Description

Constructor for `HoltWintersExponentialSmoothing`.

Parameters

`nseason` – an `int` scalar containing the number of time points in a season, or the length of the season. The class requires that $2 \leq$ nseason $\leq$ nobs unless one of the non-seasonal methods is used, in which case `nseason` is ignored. See the `setNonseasonalTrend` and `setNonseasonalConstant` methods for details.

`y` – a `double` array containing the values of the time series.

Methods

compute

```
final public double[][] compute()
```

Description

Computes the values of the smoothing parameters.

Returns

an (`nobs` + 1) by 3 `double` matrix containing the values of the smoothing parameters followed by the *level*, *trend*, and *seasonal* component sequences. Note that if the `setNonseasonalTrend` method is used, the matrix is of dimension (`nobs` + 1) by 2 and if the `setNonseasonalConstant` method is used, it is (`nobs` + 1) by 1.

getForecasts

```
public double[][] getForecasts()
```

Description

Returns the forecasts past the series data.

Returns

an `nforecasts` by 1 `double` matrix containing the forecasts past the series data. The value of the i-th row is the forecast $(i + 1)$ steps past the series data. If `setConfidence` is used, the matrix will be of dimension `nforecasts` by 3 and the value of the i-th row is the forecast $(i+1)$ steps ahead followed by the prediction interval lower and upper bounds.

getSmoothedSeries

```
public double[] getSmoothedSeries()
```

Description

Returns the fitted series values.

Returns

a `double` array containing the fitted series values.

getSumOfSquares

```
public double getSumOfSquares()
```

Description

Returns the sum of squares of the one step ahead forecast errors.

Returns

a `double` scalar containing the sum of squares of the one step ahead forecast errors.

getVarCovarMatrix

```
public double[][] getVarCovarMatrix()
```

Description

Returns the variance-covariance matrix of the smoothing parameters estimated by minimizing the mean squared forecast error. The matrix is 3 by 3 unless the `setNonseasonalTrend` method is used, in which case it is 2 by 2, or unless the `setNonseasonalConstant` method is used, in which case it is 1 by 1. This method cannot be used together with the `setParameters` method. Note that the `compute` method must be invoked before invoking this method. Otherwise, the method throws a `NullPointerException` exception.

Returns

a `double` matrix containing the variance-covariance matrix of the smoothing parameters estimated by minimizing the mean squared forecast error.

setAdditive

```
public void setAdditive()
```

Description

Specifies the use of the Additive time series model. The Multiplicative model is the default.

setConfidence

```
public void setConfidence(double confid)
```

Description

Sets the confidence level to use in the calculation of the prediction intervals.

Parameter

`confid` – a `double` scalar containing the confidence level to use in the calculation of the prediction intervals. If $0.0 < \text{confid} < 100.0$, prediction intervals are provided for each forecast.

Default: Prediction intervals are not provided.

setInitialValues

```
public void setInitialValues(double[][] sequences)
```

Description

Sets the initial values for the *level*, *trend*, and *seasonal* component sequences.

Parameter

`sequences` – an (`nobs` + 1) by 3 `double` matrix containing the initial values for the *level* , *trend*, and *seasonal* component sequences. The values must be stored in rows $1, 2, \ldots, nseason$. Rows 0 and `nseason` + 1 to `nobs` are ignored on input. `sequences[0].length` should be equal to 3 unless the `setNonseasonalTrend` method is used, in which case it is 2 containing initial values for level (α) and the trend (β). Likewise, if the `setNonseasonalConstant` method is used, `sequences[0].length` is 1 and contains the initial values for the level parameter (α) only.

Default: Initial values are computed by the function.

setLowerBounds

```
public void setLowerBounds(double[] xlb)
```

Description

Sets the lower bounds for each of the smoothing parameters, (α, β, γ).

Parameter

`xlb` – a `double` array containing the lower bounds for each of the smoothing parameters, (α, β, γ). Note that the lower bounds must be in the interval [0,1], inclusive. The array is ignored if the `setParameters` method is used. The array should be length 3 unless the `setNonseasonalTrend` method is used, in which case it is of length 2 containing lower bounds for level (α) and the trend (β) components. Likewise, if the `setNonseasonalConstant` method is used, `xlb` is of length 1 and contains the lower bound for the level parameter (α) component only.

Default: Lower bounds are all 0.

setNonseasonalConstant

```
public void setNonseasonalConstant()
```

Description

Remove the trend and the seasonal components and fit only the level component. If used, the models involve only the level (α) parameter. The method is simple exponential smoothing.

Default: The method includes all three components.

setNonseasonalTrend

```
public void setNonseasonalTrend()
```

Description

Remove the seasonal component and fit only the level and trend components. If used, the models involve only the level (α) and trend (β) parameters. The method is equivalent to double exponential smoothing.

Default: The method includes all three components.

setNumberEval

```
public void setNumberEval(int nsamples)
```

Description

Sets the number of evaluations of the residual norm that are sampled to obtain starting values for the smoothing parameters, (α, β, γ).

Parameter

`nsamples` – an `int` scalar containing the number of evaluations.

Default: `nsamples` = `nobs`.

setNumberForecasts

```
public void setNumberForecasts(int nforecasts)
```

Description

Sets the number of forecasts desired past the series data.

Parameter

`nforecasts` – an `int` scalar containing the number of forecasts desired past the series data.

Default: No forecasts are computed past the series data.

setNumberOfObservations

```
public void setNumberOfObservations(int nobs)
```

Description

Sets the number of equally spaced series values.

Parameter

`nobs` – an `int` scalar containing the number of equally spaced series values.

Default: `nobs` = `y.length`.

setParameters

```
public void setParameters(double[] params)
```

Description

Sets the values of the smoothing parameters for the level (α), the trend (β), and the seasonal (γ) component sequences.

Parameter

`params` – a `double` array containing the values of the smoothing parameters for the level (α), the trend (β), and the seasonal (γ) component sequences. The array should be length 3 unless the `setNonseasonalTrend` method is used, in which case it is of length 2 containing values for level (α) and the trend (β) components. Likewise, if the `setNonseasonalConstant` method is used, `params` is of length 1 and contains the value for the level parameter (α) component only.

Default: Parameter values are selected by minimizing the mean squared one step ahead forecast error.

setSeriesIncrement

```
public void setSeriesIncrement(int incy)
```

Description

Sets the constant stride through the series data y.

Parameter

incy – an int scalar containing the constant stride through the series data y. y.length must be at least $(\text{nobs} - 1) * |\text{incy}| + 1$. When $incy < 0$, the series is incremented in reverse order beginning at index $\text{nobs}\,(-\text{incy}) - 1$.

Default: incy = 1.

setUpperBounds

public void setUpperBounds(double[] xub)

Description

Sets the upper bounds for each of the smoothing parameters, (α, β, γ).

Parameter

xub – a double array containing the upper bounds for each of the smoothing parameters, (α, β, γ). Note that the upper bounds must be in the interval [0,1], inclusive. The array is ignored if the setParameters method is used. The array should be length 3 unless the setNonseasonalTrend method is used, in which case it is of length 2 containing upper bounds for level (α) and the trend (β) components. Likewise, if the setNonseasonalConstant method is used, xub is of length 1 and contains the upper bound for the level parameter (α) component only.

Default: Upper bounds are all 1.

Example: HoltWintersExponentialSmoothing

A series of 12 seasonal data values are analysed using the Multiplicative and the Additive Holt-Winters method. The season size is nseason = 4. The objective is to predict one season ahead, nforecasts = 4 using each method. The forecasts and prediction interval lower and upper bounds are returned. The mean sum of squares of the one-step ahead forecast errors is shown to be smallest using the Multiplicative model.

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

public class HoltWintersExponentialSmoothingEx1 {

    public static void main(String args[]) {
        double[] y = {23, 25, 36, 31, 26, 28, 48, 36, 31, 42, 53, 43};
        double confidence = 95.0;
        int nobs = y.length, nseason = 4, nforecasts = nseason;

        // Compute the time series and forecasts
        // using the Multiplicative model.
        HoltWintersExponentialSmoothing hw
                = new HoltWintersExponentialSmoothing(nseason, y);
        hw.setConfidence(confidence);
        hw.setNumberForecasts(nforecasts);
        double[][] ser = hw.compute();
        double[] ysm = hw.getSmoothedSeries();
```

```
        double[][] forecasts = hw.getForecasts();

        new PrintMatrix("Input time series").print(y);
        new PrintMatrix("Smoothed Multiplicative series").print(ysm);
        new PrintMatrix("Parameters and internal sequence").print(ser);
        new PrintMatrix("   Multiplicative forecasts\nwith "
                + confidence + "% prediction interval").print(forecasts);
        System.out.println("MSS - Multiplicative "
                + (hw.getSumOfSquares() / (double) (nobs - nseason)));

        // Compute the time series and forecasts
        // using the Additive model.
        hw = new HoltWintersExponentialSmoothing(nseason, y);
        hw.setConfidence(confidence);
        hw.setNumberForecasts(nforecasts);
        hw.setAdditive();
        ser = hw.compute();
        ysm = hw.getSmoothedSeries();
        forecasts = hw.getForecasts();

        new PrintMatrix("\n\nSmoothed Additive series").print(ysm);
        new PrintMatrix("Parameters and internal sequence").print(ser);
        new PrintMatrix("      Additive forecasts\nwith "
                + confidence + "% prediction interval").print(forecasts);
        System.out.println("MSS - Additive "
                + (hw.getSumOfSquares() / (double) (nobs - nseason)));
    }
}
```

Output

```
Input time series
    0
 0  23
 1  25
 2  36
 3  31
 4  26
 5  28
 6  48
 7  36
 8  31
 9  42
10  53
11  43

Smoothed Multiplicative series
      0
 0  23
 1  25
 2  36
 3  31
 4  24.15
 5  27.652
 6  41.767
```

```
 7  38.034
 8  30.435
 9  33.715
10  54.504
11  45.243

Parameters and internal sequence
      0       1      2
 0   0.038  1      0.437
 1   0      0      0.8
 2   0      0      0.87
 3   0      0      1.252
 4  28.75   1.438  1.078
 5  30.275  1.525  0.826
 6  31.815  1.54   0.874
 7  33.544  1.729  1.33
 8  35.202  1.657  1.054
 9  36.885  1.683  0.832
10  38.928  2.043  0.964
11  40.927  2      1.315
12  42.846  1.919  1.032

   Multiplicative forecasts
with 95.0% prediction interval
     0       1       2
0  37.252  27.539  46.964
1  44.99   35.24   54.739
2  63.908  53.989  73.826
3  52.136  42.166  62.105

MSS - Multiplicative 15.347754266704447


Smoothed Additive series
      0
 0  23
 1  25
 2  36
 3  31
 4  23
 5  26.323
 6  38.58
 7  38.542
 8  34.048
 9  35.73
10  56.627
11  43.827

Parameters and internal sequence
      0       1      2
 0   0.268  0.643   1
 1   0      0      -5.75
 2   0      0      -3.75
 3   0      0       7.25
 4  28.75   0       2.25
 5  29.555  0.518  -3.555
```

```
 6  30.523  0.807  -2.523
 7  33.859  2.432  14.141
 8  35.609  1.994   0.391
 9  36.785  1.468  -5.785
10  39.936  2.55    2.064
11  41.513  1.924  11.487
12  43.215  1.781  -0.215

      Additive forecasts
with 95.0% prediction interval
     0       1       2
0  39.211  27.801  50.621
1  48.841  36.371  61.311
2  60.046  45.745  74.347
3  50.125  33.244  67.007

MSS - Additive 21.181134361693125
```

TimeSeries class

`public class com.imsl.stat.TimeSeries implements Serializable, Cloneable`

A specialized class for time series data and analysis.

Constructor

TimeSeries

`public TimeSeries()`

Description

Constructor for TimeSeries.

Methods

getDateIncrement

`public long getDateIncrement()`

Description

Returns the date increment for this TimeSeries object.

Returns

a `long`, the date increment

getDates

`public Date[] getDates()`

Description

Returns the date array associated with the time series.

See Java documentation on the Date class.

Returns

a `Date` array containing the dates of the series

getLength

`public int getLength()`

Description

Returns the length of the time series.

Returns

an `int`, the length of the time series

getMissingIndicator

`public int[][] getMissingIndicator()`

Description

Returns an array of missing value indicators.

A missing value is represented in the series by a `Double.NaN`.

Returns

an `int` matrix having the same dimension as the time series where `getMissingIndicator()[i][j]=1` if `y[i*numSeries+j]` or `y[i][j]` is a missing value, otherwise `getMissingIndicator()[i][j]=0`.

getNumMissing

`public int getNumMissing()`

Description

Returns the number of missing values.

A missing value is represented in the series by a `Double.NaN`

Returns

an `int` representing the number of missing values detected in the time series.

getNumberOfSeries

`public int getNumberOfSeries()`

Description

Returns the number of series stored in this TimeSeries object.

Returns

an `int`, the number of different series

getSeriesValues

```
public double[][] getSeriesValues()
```

Description

Returns the values associated with the time series.

Returns

a `double` matrix containing the values of the series

getSeriesValues

```
public double[] getSeriesValues(int k)
```

getStartDate

```
public Date getStartDate()
```

Description

Returns the starting date of the time series.

Returns

a `Date` containing the first date of the series

getTimeZone

```
public TimeZone getTimeZone()
```

Description

Returns the time zone of the time series.

See Java documentation for the TimeZone class.

Returns

a `TimeZone` object

getTimeZoneOffset

```
public long getTimeZoneOffset()
```

Description

Returns the number of hours (+ or -) from GMT of the time zone associated with the time series. The time zone is the same as the local time zone unless it is changed by the user.

To convert to milliseconds, multiply by 60*60*1000. Note that the displayed time zone will be the name of the local time zone, despite what is set for the TimeSeries object. For further information, see the Examples provided and Java documentation on Date and Calendar classes.

Returns

a `long` value, the the number of hours

isDateIncrementInMillis

```
public boolean isDateIncrementInMillis()
```

Description

Returns a boolean indicating whether or not the TimeSeries object has a date increment expressed in milliseconds. If `true` the date increment is considered to be expressed in milliseconds; otherwise, it is assumed to be in the number of days.

Returns

a `boolean`

isHasDates

`public boolean isHasDates()`

Description

Returns a boolean indicating whether or not the TimeSeries has a non-null dates attribute.

Returns

a `boolean`. When `true`, the dates attribute is not `null`.

setDateIncrement

`public void setDateIncrement(int increment)`

Description

Sets the date increment in number of days.

Parameter

> `increment` – an `int` value representing the number of days between observations. By using increments, the time series is presumed to be evenly spaced.

setDateIncrementInMillis

`public void setDateIncrementInMillis(long incrementInMillis)`

Description

Sets the date increment in milliseconds.

Parameter

> `incrementInMillis` – a `long` value representing the number of milliseconds between series values. By using increments, the time series is presumed to be evenly spaced.

setDates

`public void setDates()`

Description

Sets the date array using the start date and date increment.

setDates

`public void setDates(Date[] dateArray)`

Description

Sets the date array equal to user supplied dates.

Parameter

`dateArray` – a `Date` array containing the dates associated with each value of the time series

setSeriesValues

`public void setSeriesValues(double[] y)`

Description

Sets the values of a univariate time series and initializes the time index. If the start date is set by the user, the date array starts at the starting date and increments by 1 day up to the series length. The start date can be set using `setStartDate` and the increment in days can be set using `setDateIncrement` or in milliseconds using `setDateIncrementInMillis` .

Parameter

`y` – a `double` array containing the numeric values of the series. Missing values must be provided as NaN's.

setSeriesValues

`public void setSeriesValues(double[][] y)`

Description

Sets the values of the TimeSeries.

Parameter

`y` – a `double` matrix containing the values of the time series. The number of series is set to the number of columns of `y`. Missing values must be provided as `Double.NaN`.

setSeriesValues

`public void setSeriesValues(double[] y, int numSeries)`

Description

Sets the values of a multivariate time series and initializes the time index array.

Parameters

`y` – a `double` array containing the numeric values of the series. It is assumed that the values are row-oriented, meaning that the series index varies the fastest. Missing values must be provided as `Double.NaN`.

`numSeries` – an `int`, the number of different series.

setStartDate

`public void setStartDate(Date startDate)`

Description

Sets the start date of the series.

Parameter

startDate – a Date value, the date associated with the first observation. The default start date is set to null.

setTimeZone

```
public void setTimeZone(int offset)
```

Description

Sets the time zone for the time series to the time zone associated with the given offset from GMT.

Parameter

offset – an int representing the number of hours (+ or -) difference from GMT.

setTimeZone

```
public void setTimeZone(TimeZone tz)
```

Description

Sets the time zone for the time series to the given TimeZone.

Parameter

tz – a java.util.TimeZone object.

setTimeZone

```
public void setTimeZone(int offset, String id)
```

Description

Sets the time zone for the time series using the offset and String id.

Parameters

offset – an int representing the number of hours (+ or -) difference from GMT.

id – a String representing the desired name of the time zone. The provided id must be among the available ids, or else the first element of TimeZone.getAvailableIDS(offset*60*60*1000) will be used.

Example: TimeSeries

This example illustrates setting up a TimeSeries class for a segment of S&P 500 returns during a period in the late 80's.

```
import com.imsl.stat.*;
import java.util.Date;
import java.text.*;

public class TimeSeriesEx1 {

    public static void main(String args[]) throws ParseException {
        double data1[] = {
            0.940543272, 2.664279727, 0.252045652, 1.221098464, 0.270575973,
            0.446882726, 0.073672969, -4.450030305, 0.007699415, -0.215808015,
```

```
            -0.498908466, -0.696433123, 0.113159704, 1.440322251, 0.887874183,
            0.5585227, -0.049267619, -0.452119841, -0.488606538, 0.087972619,
            0.549032948, -1.024238871, -0.589472147, -0.507487482, -0.365744931,
            0.42010331, -1.687159586, 0.212950629, 1.147614577, 0.748805609,
            -1.311175287, -1.574960025, 0.484204788, 0.178009897, -0.869311816,
            1.062785115, 0.098566846, 0.342257263, -0.47633197
        };

        String datestrings1[] = {
            "4/5/1988", "4/6/1988", "4/7/1988",
            "4/8/1988", "4/11/1988", "4/12/1988",
            "4/13/1988", "4/14/1988", "4/15/1988",
            "4/18/1988", "4/19/1988", "4/20/1988",
            "4/21/1988", "4/22/1988", "4/25/1988",
            "4/26/1988", "4/27/1988", "4/28/1988",
            "4/29/1988", "5/2/1988", "5/3/1988",
            "5/4/1988", "5/5/1988", "5/6/1988",
            "5/9/1988", "5/10/1988", "5/11/1988",
            "5/12/1988", "5/13/1988", "5/16/1988",
            "5/17/1988", "5/18/1988", "5/19/1988",
            "5/20/1988", "5/23/1988", "5/24/1988",
            "5/25/1988", "5/26/1988", "5/27/1988"
        };

        SimpleDateFormat dateFormat = new SimpleDateFormat("M/d/y");

        TimeSeries ts = new TimeSeries();
        ts.setSeriesValues(data1);

        System.out.println("The number of observations  = " + ts.getLength());
        System.out.println("The number of missing observations = "
                + ts.getNumMissing());

        Date dates1[] = new Date[ts.getLength()];

        for (int i = 0; i < ts.getLength(); i++) {
            dates1[i] = dateFormat.parse(datestrings1[i]);
        }
        ts.setDates(dates1);
        System.out.println("The starting date is = " + ts.getStartDate());
    }
}
```

Output

```
The number of observations  = 39
The number of missing observations = 0
The starting date is = Tue Apr 05 00:00:00 CDT 1988
```

Example: TimeSeries

This example illustrates setting up a different time zone.

```
import com.imsl.stat.TimeSeries;
```

```
import java.text.*;
import java.util.*;

public class TimeSeriesEx2 {

    public static void main(String args[]) throws ParseException {
        String dateStrings[] = {
            "11/23/2011 11:13:27", "9/14/2011 13:15:10", "7/28/2012 20:18:32",
            "8/7/2012 00:00:16", "6/3/2011 1:21:03", "9/14/2011 17:18:22"
        };
        double data[] = {-1.0, 2.5, 6.773, -8.92, 4.117, 16.27};
        Date dates[] = new Date[data.length];
        SimpleDateFormat dateFormat = new SimpleDateFormat("M/d/y H:mm:ss");
        SimpleDateFormat printDateFormat
                = new SimpleDateFormat("M/d/y H:mm:ss, a");

        for (int i = 0; i < data.length; i++) {
            dates[i] = dateFormat.parse(dateStrings[i]);
        }

        TimeSeries ts;
        ts = new TimeSeries();

        ts.setSeriesValues(data);
        ts.setDates(dates);

        System.out.println("Local Timezone offset in hours is "
                + ts.getTimeZoneOffset());
        System.out.println("Local Timezone name is " + ts.getTimeZone().getID());

        /* Note: changing the time zone does not change the time values that
         were set in setValues().  Subtract the offset in order to adjust the
         times, if necessary.
         */
        ts.setTimeZone(-8, "PST");
        TimeZone tz = ts.getTimeZone();
        System.out.println("New offset is " + ts.getTimeZoneOffset());
        System.out.println("New name is " + tz.getID());


        /* Default printing will use the local time zone.  Here is the manual
         way to print the time zone that was set. */
        System.out.println(printDateFormat.format(ts.getDates()[4]) + " "
                + tz.getDisplayName());

        /* Use the offset to display the equivalent GMT time */
        Date gmtTime = new Date(ts.getDates()[4].getTime()
                - ts.getTimeZoneOffset() * 60 * 60 * 1000);
        System.out.println(printDateFormat.format(gmtTime) + " GMT ");
    }
}
```

Output

```
Local Timezone offset in hours is -6
```

```
Local Timezone name is America/Chicago
New offset is -8
New name is PST
7/28/2012 20:18:32, PM Pacific Standard Time
7/29/2012 4:18:32, AM GMT
```

TimeSeriesOperations class

`public class com.imsl.stat.TimeSeriesOperations implements Serializable, Cloneable`

A class of operations and methods for objects of class TimeSeries.

Constructor

TimeSeriesOperations

`public TimeSeriesOperations()`

Description

Constructor for TimeSeriesOperations.

Methods

backshift

`public TimeSeries backshift(TimeSeries ts, int lag)`

Description

Returns the backshifted version of the time series. For a given lag $l \geq 0$ the backshift operation is defined as:

$$B^l y_t = y_{t-l}$$

.

Parameters

`ts` – a `TimeSeries` object.

`lag` – an `int` specifying the lag.

Returns

`TimeSeries` with values at index `i` the same as the original series values at position `i-lag`;

merge

```
public TimeSeries merge(TimeSeries ts1, TimeSeries ts2)
```

Description

Merges two time series objects.

Parameters

`ts1` – a `TimeSeries` object.

`ts2` – a `TimeSeries` object.

Returns

a `TimeSeries` object resulting from merging `ts1` and `ts2`.

setCombineFunction

```
public void setCombineFunction(TimeSeriesOperations.Function function)
```

Description

Sets the combine function to a user supplied function. Must be defined when combineMethod = CombineMethod.CUSTOM.

Parameter

`function` – a `Function` that defines how two values should be combined.

setCombineMethod

```
public void setCombineMethod(TimeSeriesOperations.CombineMethod combineMethod)
```

Description

Sets the method for combining synchronous time series values.

Parameter

`combineMethod` – a `CombineMethod` value.

setMergeRule

```
public void setMergeRule(TimeSeriesOperations.MergeRule mergeRule)
```

Description

Sets the rule that defines how two time series are merged.

Parameter

`mergeRule` – a `MergeRule` value.

stack

```
public double[] stack(TimeSeries ts)
```

Description

Stacks or vectorizes the values of a multivariate TimeSeries.

Parameter

ts – TimeSeries

Returns

A double[] array of length T*numberOfSeries with the index 1,...,numberOfSeries varying fastest.

Example 1: TimeSeriesOperations

This example illustrates the action of the Merge function under different settings of the Merge rule.

```
import com.imsl.stat.*;
import java.text.*;

public class TimeSeriesOperationsEx1 {

    public static void main(String args[]) throws ParseException {
        SimpleDateFormat dateFormat = new SimpleDateFormat("M/d/y");
        TimeSeries ts1 = new TimeSeries();
        TimeSeries ts2 = new TimeSeries();

        /* Create dates in different increments */
        ts1.setStartDate(dateFormat.parse("12/12/2010"));
        ts1.setDateIncrement(3);

        ts2.setStartDate(dateFormat.parse("12/12/2010"));
        ts2.setDateIncrement(5);

        int n = 10;

        double y1[] = new double[n];
        double y2[] = new double[n];

        /* Create series values */
        for (int i = 0; i < n; i++) {
            y1[i] = (i + 1.0);
            y2[i] = -(i + 1.0);
        }
        ts1.setSeriesValues(y1);
        ts2.setSeriesValues(y2);

        double[][] values1 = ts1.getSeriesValues();
        double[][] values2 = ts2.getSeriesValues();

        System.out.println("Time series 1 \t \tTime series 2");
        for (int i = 0; i < n; i++) {
            System.out.println(dateFormat.format(ts1.getDates()[i])
                    + "\t" + values1[i][0]
                    + "\t" + dateFormat.format(ts2.getDates()[i])
                    + "\t" + values2[i][0]);
```

```
        }
        TimeSeriesOperations tsOps = new TimeSeriesOperations();
        TimeSeries ts3 = tsOps.merge(ts1, ts2);
        double[][] values3 = ts3.getSeriesValues();
        System.out.println("Merge result using the union (default) merge rule: ");
        System.out.println("Series length: " + ts3.getLength());
        for (int i = 0; i < ts3.getLength(); i++) {
            System.out.println(dateFormat.format(ts3.getDates()[i])
                    + "\t" + values3[i][0]);
        }
        tsOps.setMergeRule(TimeSeriesOperations.MergeRule.INTERSECTION);
        ts3 = tsOps.merge(ts1, ts2);
        values3 = ts3.getSeriesValues();
        System.out.println("Merge results using the intersection merge rule: ");
        System.out.println("Series length: " + ts3.getLength());
        for (int i = 0; i < ts3.getLength(); i++) {
            System.out.println(dateFormat.format(ts3.getDates()[i])
                    + "\t" + values3[i][0]);
        }
        tsOps.setMergeRule(TimeSeriesOperations.MergeRule.UNION_MISSING);
        ts3 = tsOps.merge(ts1, ts2);
        values3 = ts3.getSeriesValues();
        System.out.println("Merge results using the union with missing rule: ");
        System.out.println("Series length: " + ts3.getLength());
        System.out.println("Number of missing values: " + ts3.getNumMissing());
        for (int i = 0; i < ts3.getLength(); i++) {
            System.out.println(dateFormat.format(ts3.getDates()[i])
                    + "\t" + values3[i][0]);
        }
    }
}
```

Output

```
Time series 1    Time series 2
12/12/2010 1.0 12/12/2010 -1.0
12/15/2010 2.0 12/17/2010 -2.0
12/18/2010 3.0 12/22/2010 -3.0
12/21/2010 4.0 12/27/2010 -4.0
12/24/2010 5.0 1/1/2011 -5.0
12/27/2010 6.0 1/6/2011 -6.0
12/30/2010 7.0 1/11/2011 -7.0
1/2/2011 8.0 1/16/2011 -8.0
1/5/2011 9.0 1/21/2011 -9.0
1/8/2011 10.0 1/26/2011 -10.0
Merge result using the union (default) merge rule:
Series length: 18
12/12/2010 0.0
12/15/2010 2.0
12/17/2010 -2.0
12/18/2010 3.0
12/21/2010 4.0
12/22/2010 -3.0
12/24/2010 5.0
12/27/2010 1.0
```

```
12/30/2010 7.0
1/1/2011 -5.0
1/2/2011 8.0
1/5/2011 9.0
1/6/2011 -6.0
1/8/2011 10.0
1/11/2011 -7.0
1/16/2011 -8.0
1/21/2011 -9.0
1/26/2011 -10.0
Merge results using the intersection merge rule:
Series length: 2
12/12/2010 0.0
12/27/2010 1.0
Merge results using the union with missing rule:
Series length: 18
Number of missing values: 16
12/12/2010 0.0
12/15/2010 NaN
12/17/2010 NaN
12/18/2010 NaN
12/21/2010 NaN
12/22/2010 NaN
12/24/2010 NaN
12/27/2010 1.0
12/30/2010 NaN
1/1/2011 NaN
1/2/2011 NaN
1/5/2011 NaN
1/6/2011 NaN
1/8/2011 NaN
1/11/2011 NaN
1/16/2011 NaN
1/21/2011 NaN
1/26/2011 NaN
```

Example 2: TimeSeriesOperations

This example illustrates merging two time series using different combine methods.

```
import com.imsl.stat.*;
import java.util.Date;
import java.text.*;

public class TimeSeriesOperationsEx2 {

    public static void main(String args[]) throws ParseException {
        double data1[] = {
            0.940543272, 2.664279727, 0.252045652, 1.221098464, 0.270575973,
            0.446882726, 0.073672969, -4.450030305, 0.007699415, -0.215808015
        };

        double data2[] = {
            0.42010331, -1.687159586, 0.212950629, 1.147614577, 0.748805609,
```

```
    -1.311175287, -1.574960025, 0.484204788, 0.178009897, -0.869311816,
    1.062785115, 0.098566846, 0.342257263, -0.47633197
};

String datestrings1[] = {
    "4/5/1988", "4/6/1988", "4/7/1988", "4/8/1988", "4/11/1988",
    "4/12/1988", "4/13/1988", "4/14/1988", "4/15/1988", "4/18/1988"
};

String datestrings2[] = {
    "4/13/1988", "4/14/1988", "4/15/1988", "4/19/1988", "4/20/1988",
    "4/21/1988", "4/22/1988", "4/25/1988", "4/26/1988", "4/27/1988",
    "4/28/1988", "4/29/1988", "5/2/1988", "5/3/1988"
};

SimpleDateFormat dateFormat = new SimpleDateFormat("M/d/y");

TimeSeries ts1 = new TimeSeries();
ts1.setSeriesValues(data1);
Date dates1[] = new Date[ts1.getLength()];

for (int i = 0; i < ts1.getLength(); i++) {
    dates1[i] = dateFormat.parse(datestrings1[i]);
}
ts1.setDates(dates1);

TimeSeries ts2 = new TimeSeries();
ts2.setSeriesValues(data2);
Date dates2[] = new Date[ts2.getLength()];

for (int i = 0; i < ts2.getLength(); i++) {
    dates2[i] = dateFormat.parse(datestrings2[i]);
}
ts2.setDates(dates2);

TimeSeriesOperations tsOps = new TimeSeriesOperations();
TimeSeries ts3 = tsOps.merge(ts1, ts2);
double[][] values3 = ts3.getSeriesValues();

System.out.println("Result using MergeRule.UNION and "
        + "CombineFunction.AVERAGE (defaults):");
for (int i = 0; i < ts3.getLength(); i++) {
    System.out.println(dateFormat.format(ts3.getDates()[i])
            + "\t" + values3[i][0]);
}

tsOps.setMergeRule(TimeSeriesOperations.MergeRule.INTERSECTION);
tsOps.setCombineMethod(TimeSeriesOperations.CombineMethod.ABS_DIFF);
ts3 = tsOps.merge(ts1, ts2);
values3 = ts3.getSeriesValues();

System.out.println("Result using MergeRule.INTERSECTION and "
        + "CombineFunction.ABS_DIFF:");
for (int i = 0; i < ts3.getLength(); i++) {
    System.out.println(dateFormat.format(ts3.getDates()[i])
            + "\t" + values3[i][0]);
```

```
        }
    }
}
```

Output

```
Result using MergeRule.UNION and CombineFunction.AVERAGE (defaults):
4/5/1988 0.940543272
4/6/1988 2.664279727
4/7/1988 0.252045652
4/8/1988 1.221098464
4/11/1988 0.270575973
4/12/1988 0.446882726
4/13/1988 0.2468881395
4/14/1988 -3.0685949455
4/15/1988 0.110325022
4/18/1988 -0.215808015
4/19/1988 1.147614577
4/20/1988 0.748805609
4/21/1988 -1.311175287
4/22/1988 -1.574960025
4/25/1988 0.484204788
4/26/1988 0.178009897
4/27/1988 -0.869311816
4/28/1988 1.062785115
4/29/1988 0.098566846
5/2/1988 0.342257263
5/3/1988 -0.47633197
Result using MergeRule.INTERSECTION and CombineFunction.ABS_DIFF:
4/13/1988 0.34643034100000003
4/14/1988 2.7628707190000004
4/15/1988 0.20525121400000002
```

Example 3: TimeSeriesOperations

This example illustrates the backshift operation.

```
import com.imsl.stat.*;
import java.text.*;

public class TimeSeriesOperationsEx3 {

    public static void main(String args[]) throws ParseException {
        SimpleDateFormat dateFormat = new SimpleDateFormat("M/d/y");
        TimeSeries ts1 = new TimeSeries();
        /* Construct the time series*/
        ts1.setStartDate(dateFormat.parse("12/12/2010"));
        ts1.setDateIncrement(3);

        int n = 10;
        double y1[] = new double[n];

        /* Create series values */
```

```
        for (int i = 0; i < n; i++) {
            y1[i] = (i + 1.0);
        }
        ts1.setSeriesValues(y1);
        TimeSeriesOperations tsOps = new TimeSeriesOperations();

        TimeSeries ts2 = tsOps.backshift(ts1, 2);
        double[][] values1 = ts1.getSeriesValues();
        double[][] values2 = ts2.getSeriesValues();

        System.out.println("Time series 1 \t \tTime series 2");
        for (int i = 0; i < n; i++) {
            System.out.println(dateFormat.format(ts1.getDates()[i]) + "\t"
                    + values1[i][0] + "\t"
                    + dateFormat.format(ts2.getDates()[i]) + "\t"
                    + values2[i][0]);
        }
    }
}
```

Output

```
Time series 1    Time series 2
12/12/2010 1.0 12/6/2010 NaN
12/15/2010 2.0 12/9/2010 NaN
12/18/2010 3.0 12/12/2010 1.0
12/21/2010 4.0 12/15/2010 2.0
12/24/2010 5.0 12/18/2010 3.0
12/27/2010 6.0 12/21/2010 4.0
12/30/2010 7.0 12/24/2010 5.0
1/2/2011 8.0 12/27/2010 6.0
1/5/2011 9.0 12/30/2010 7.0
1/8/2011 10.0 1/2/2011 8.0
```

Example 4: TimeSeriesOperations

This example illustrates the stacking or vectorizing operation.

```
import com.imsl.stat.*;
import java.text.*;

public class TimeSeriesOperationsEx4 {

    public static void main(String args[]) throws ParseException {
        SimpleDateFormat dateFormat = new SimpleDateFormat("M/d/y");
        TimeSeries ts1 = new TimeSeries();
        /* Construct the time series*/
        ts1.setStartDate(dateFormat.parse("12/12/2010"));
        ts1.setDateIncrement(3);

        int n = 10;
        int nSeries = 3;
        double y1[] = new double[nSeries * n];
```

```
        /* Create series values */
        for (int i = 0; i < n; i++) {
            y1[3 * i] = (i + 1.0);
            y1[3 * i + 1] = 2 * (i + 1.0);
            y1[3 * i + 2] = 3 * (i + 1.0);
        }
        ts1.setSeriesValues(y1, nSeries);
        TimeSeriesOperations tsOps = new TimeSeriesOperations();
        double[][] values1 = ts1.getSeriesValues();
        double[] vecy = tsOps.stack(ts1);

        System.out.println("Date\t\tSeries 1\tSeries 2\tSeries 3 ");
        for (int i = 0; i < n; i++) {
            System.out.println(dateFormat.format(ts1.getDates()[i])
                    + "\t\t" + values1[i][0]
                    + "\t\t" + values1[i][1]
                    + "\t\t" + values1[i][2]);
        }

        System.out.println("\nDate\t\tStacked (or vectorized) Series  ");
        for (int j = 0; j < nSeries; j++) {
            for (int i = 0; i < n; i++) {
                System.out.println(dateFormat.format(ts1.getDates()[i])
                        + "\t\t" + vecy[j * n + i]);
            }
        }
    }
}
```

Output

```
Date Series 1 Series 2 Series 3
12/12/2010 1.0 2.0 3.0
12/15/2010 2.0 4.0 6.0
12/18/2010 3.0 6.0 9.0
12/21/2010 4.0 8.0 12.0
12/24/2010 5.0 10.0 15.0
12/27/2010 6.0 12.0 18.0
12/30/2010 7.0 14.0 21.0
1/2/2011 8.0 16.0 24.0
1/5/2011 9.0 18.0 27.0
1/8/2011 10.0 20.0 30.0

Date Stacked (or vectorized) Series
12/12/2010 1.0
12/15/2010 2.0
12/18/2010 3.0
12/21/2010 4.0
12/24/2010 5.0
12/27/2010 6.0
12/30/2010 7.0
1/2/2011 8.0
1/5/2011 9.0
1/8/2011 10.0
```

```
12/12/2010 2.0
12/15/2010 4.0
12/18/2010 6.0
12/21/2010 8.0
12/24/2010 10.0
12/27/2010 12.0
12/30/2010 14.0
1/2/2011 16.0
1/5/2011 18.0
1/8/2011 20.0
12/12/2010 3.0
12/15/2010 6.0
12/18/2010 9.0
12/21/2010 12.0
12/24/2010 15.0
12/27/2010 18.0
12/30/2010 21.0
1/2/2011 24.0
1/5/2011 27.0
1/8/2011 30.0
```

TimeSeriesOperations.Function interface

`public interface com.imsl.stat.TimeSeriesOperations.Function`

Public interface for the user-supplied function that defines how to combine two synchronous time series values.

Method

compute

`public double compute(double x, double y)`

Description

Public interface for the user-supplied function to combine two time series values that occur at the same date and time.

Parameters

`x` – a `double` representing the value of the first time series.

`y` – a `double` representing the value of the second time series.

Returns

a `double` representing the combination of the two inputs.

TimeSeriesOperations.MergeRule class

`static public final class com.imsl.stat.TimeSeriesOperations.MergeRule extends java.lang.Enum`

Public enum of merge rules that defines how two time series should be merged.

Fields

INTERSECTION

`static final public TimeSeriesOperations.MergeRule INTERSECTION`

The merge operation includes time points and values only at the matching time points and applies the CombineMethod to the values at the matching time points.

UNION

`static final public TimeSeriesOperations.MergeRule UNION`

The merge operation includes all time points and values from each time series and applies the CombineMethod to the values at the matching time points.

UNION_MISSING

`static final public TimeSeriesOperations.MergeRule UNION_MISSING`

The merge operation includes all time points but applies the CombineMethod only to values at matching time points, indicating a missing value `Double.NaN` for time points that are not in the intersection.

Methods

valueOf

`static public TimeSeriesOperations.MergeRule valueOf(String name)`

values

`static public TimeSeriesOperations.MergeRule[] values()`

TimeSeriesOperations.CombineMethod class

```
static public final class com.imsl.stat.TimeSeriesOperations.CombineMethod
extends java.lang.Enum
```

Public enum of methods for combining synchronous time series values.

Fields

ABS_DIFF

```
static final public TimeSeriesOperations.CombineMethod ABS_DIFF
```

Takes the absolute difference |ts1-ts2| between the two values.

AVERAGE

```
static final public TimeSeriesOperations.CombineMethod AVERAGE
```

Takes the average of the two values.

CUSTOM

```
static final public TimeSeriesOperations.CombineMethod CUSTOM
```

Uses a custom combine function that is provided by the user.

DIFF

```
static final public TimeSeriesOperations.CombineMethod DIFF
```

Takes the difference (ts1-ts2) of the two values.

FIRST

```
static final public TimeSeriesOperations.CombineMethod FIRST
```

Uses the value from ts1, the first series in the call.

MAX

```
static final public TimeSeriesOperations.CombineMethod MAX
```

Uses the maximum of the two values.

MIN

```
static final public TimeSeriesOperations.CombineMethod MIN
```

Uses the minimum of the two values.

SUM

```
static final public TimeSeriesOperations.CombineMethod SUM
```

Takes the sum of the two values.

Methods

valueOf

```
static public TimeSeriesOperations.CombineMethod valueOf(String name)
```

values

```
static public TimeSeriesOperations.CombineMethod[] values()
```

VectorAutoregression class

```
public class com.imsl.stat.VectorAutoregression implements Serializable,
Cloneable
```

Performs vector autoregression for a multivariate time series.

This class contains methods for modeling multivariate time series of the form

$$Y_t = (y_{1t}, y_{2t}, \ldots, y_{Kt})^\top$$

where each

$$y_{it}, i = 1, \ldots, K, t = \ldots, -2, -1, 0, 1, 2, \ldots$$

is a real-valued time series indexed by t. Define the K dimensional mean process

$$\mu_t = E[Y_t] = (E[y_{1t}], \ldots, E[y_{Kt}])^\top$$

and the cross-covariance matrix

$$\Sigma_{t,h} = \mathrm{Cov}(Y_t, Y_{t+h})$$

where for each *(t,h)*, $\Sigma_{t,h}$ is a $K * K$ real-valued matrix whose $(i,j)^{t,h}$ element is the covariance between $y_{it}, y_{j(t+h)}$:

$$\mathrm{Cov}(y_{it}, y_{j(t+h)}) = E[(y_{it} - \mu_{it})(y_{j(t+h)} - \mu_{j(t+h)})].$$

The vector autoregressive model, or VAR, has a number of equivalent forms. A general form is

$$A(L)(Y_t - \mu_t) = u_t + DX_t$$

where

$$A(L) = A_0 + A_1 L + A_2 L^2 + \ldots + A_p L^p$$

is a *p-th* degree matrix polynomial in the lag or backshift operator, L. Note that p is the specified autoregressive lag and u_t is the error process. The error process is assumed to be a K dimensional white noise series, with $E[u_t] = 0$ and nonsingular covariance matrix, Σ_u, which is constant over time. The coefficient matrices $A_j, j = 0, \ldots, p$ are $K * K$ matrices containing the coefficients of the model. The matrix X_t represents a matrix of deterministic terms such as trend or seasonal variables, and D represents the associated coefficient matrix of X_t.

Constructor

VectorAutoregression

```
public VectorAutoregression(TimeSeries ts)
```

Description

Constructor for the class.

Parameter

`ts` – a `com.imsl.stat.TimeSeries` (p. 1088) object.

Methods

getARConstants

```
public double[] getARConstants()
```

Description

Returns the current settings of the constants used in the autoregression model.

Returns

a `double` array containing the constants used in the autoregression model. If the value is `null`, the default values are active.

getARModel

```
public int[] getARModel()
```

Description

Returns the autoregressive model configuration.

Returns

an `int` array specifying the autoregressive configuration. If `null`, then all parameters are active in the current model.

getEstimates

```
public double[][] getEstimates()
```

Description

Returns the parameter estimates (coefficients) of the vector autoregression model.

Returns

a `double` array containing the estimated parameters (coefficients).

getForecasts

```
public double[][] getForecasts()
```

Description

Returns the *h*-step ahead forecast at times *t*=*nT*, *nT*+1, ..., *T*, where *h*=1,2, ..., `maxStepsAhead`.

The *h*-step ahead forecast

$$\hat{Y}_{t+h} = E[Y_{t+h}|Y_s, 0 \leq s \leq t]$$

uses the conditional expectation of Y_{t+h} given the historical information known at time *t*. The estimated VAR model approximates the conditional expectation and the forecasts are generated from the estimated VAR model via the following recursion:

$$\hat{Y}_{t+h} = \sum_{j=1}^{p} \hat{A}_j \hat{Y}_{t+h-j} + \delta$$

where $Y = (Y_1, ..., Y_K)^T$ is the vector time series and $\delta = (I - \hat{A}_1 - \hat{A}_2 - \cdots - \hat{A}_p)\mu$.

Returns

a `double` matrix *T*-*nT* by `maxStepsAhead` * *K* containing the forecasts.

Note that the forecast $\hat{Y}_{kt+h}$ is stored in location [*t*-*nT*][*(h-1)***K* + *k*] of the returned forecast matrix.

isA0Flag

```
public boolean isA0Flag()
```

Description

Returns the state of `A0Flag`.

Returns

a `boolean` which specifies whether or not the leading coefficient matrix is nontrivial.

setA0Flag

```
public void setA0Flag(boolean A0Flag)
```

Description

Sets the flag to include the leading autoregressive coefficient matrix in the model.

When `true`, a nontrivial, lower-triangular leading autoregressive coefficient matrix, A_0, will be estimated in the model. When `false`, A_0 is the constant identity matrix, which is also the default case.

Parameter

`A0Flag` – `boolean` indicating whether to fit the leading coefficient matrix.
Default: `A0Flag=false`.

setARConstants

```
public void setARConstants(double[] arConstants)
```

Description

Sets the constants for the autoregressive model. See discussion in `com.imsl.stat.VectorAutoregression.setARModel` (p. 1111) for details.

Parameter

`arConstants` – a `double` array specifying the autoregressive model constants.

The input array `arConstants` must be of length (`A0Flag`+`arLag`)*K*K, such that for indices 1=0,...,(`A0Flag` + `arLag`), i=0,...,K-1, and j=0,...,K-1, `arConstants[`l*K*K +i*K + j`]` specifies a constant value for parameter $a_{ij,l}$.Default: `arConstants[`i`]`=0.

---

setARLag

```
public void setARLag(int arLag)
```

Description

Sets the autoregressive lag parameter. It must be nonnegative and less than `maxLag`.

Parameter

`arLag` – an `int` specifying the desired lag for the autoregressive terms. Default: `arLag`=1.

---

setARModel

```
public void setARModel(int[] arModel)
```

Description

Sets the form of the autoregressive terms of the model.

Without loss of generality, assume that the series mean is 0 and that there are no deterministic terms. Then we can write the vector autoregression model of lag p as

$$A_0Y_t + A_1Y_{t-1} + \cdots + A_pY_{t-p} = u_t.$$

Each of the matrices A_j represents K * K unknown coefficients. In practice, many of the $(p+ 1)$ * K * K coefficients are restricted to be 0 or otherwise constant. The leading coefficient matrix A_0 must be lower-triangular and is very often equal to the identity matrix (this is the default behavior). Fully expanded the model looks like K equations in $K(p+1)$ unknowns.

$$Y_{t,k} = \sum_{j=1}^{K} a_{kj,0}Y_{t,j} + \sum_{l=1}^{p}\sum_{j=1}^{K} a_{kj,l}Y_{t-l,k}$$

Consider each coefficient matrix as the vector

$$\text{vec}(A_l) = (a_{11,l}, a_{21,l}, \ldots, a_{K1,l}, a_{12,l}, a_{22,l}, \ldots, a_{K2,l}, \cdots, a_{1K,l}, a_{2K,l}, \ldots, a_{KK,l})^T$$

for $l = 0, \ldots, p$.

$$\text{vec}(A_l)$$

is just the columns of

$$A_l$$

appended to each other. To specify a configuration different from the default, append the vectorized coefficient matrices as

$$\text{vec}(A_1, A_2, \ldots, A_p)$$

or

$$\text{vec}(A_0, A_1, \ldots, A_p)$$

when including a nontrivial leading coefficient matrix. Set the value at index $l*K*K + j*K + i$ to 1 or -1 to fit $a_{ij,l}$ or $-a_{ij,l}$ in the model. Set the value to 0 to exclude $a_{ij,l}$ as a parameter. Note that for the leading coefficient matrix,l= 0, the value at $j*K + i$ is ignored unless i is at least j, i.e., only the lower triangle of A_0 can be nontrivial.

Parameter

`arModel` – an `int` array specifying the autoregressive model parameters.
For indices l=0,...,(`A0Flag+arLag`), i=0,...,K-1, and j=0,...,K-1, `arModel[`$l*K*K + i*K + j$`]` = {-1,1} indicates that the coefficient $\pm a_{ij,l}$ is a parameter (to be estimated) in the model.
Default: A_0 is the identity matrix and `arModel[`i`]`=1 for all i.

---

setCenter

`public void setCenter(boolean center)`

Description

Sets the flag to center the data. If `center=true`, column means are subtracted from the data.

Parameter

`center` – a `boolean` indicating whether or not column means should be subtracted from the data.
Default: `center=false`.

---

setMaxLag

`public void setMaxLag(int maxLag)`

Description

Sets the maximum lag.

Parameter

`maxLag` – an `int` specifying the maximum lag to consider in the first (pure VAR) stage regression.

---

setScale

`public void setScale(boolean scale)`

Description

Sets the flag to scale the data.

If `scale=true`, the data values are mean-centered and then divided by the standard deviation.

Parameter

`scale` – a `boolean`
Default: `scale=false`.

---

setTrend

`public void setTrend(boolean trend)`

Description

Sets the flag to fit a trend parameter in the model. Set to `true` to include a deterministic trend in the model.

Parameter

`trend` – a `boolean` specifying whether or not to include a deterministic trend.

Default: `trend=false`.

Example 1: Vector Autoregression

This example fits a VAR(1) to a 2-dimensional vector time series. Then, forecasts for up to 4 steps ahead are requested, illustrating the default forecasting behavior.

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

public class VectorAutoregressionEx1 {

    public static void main(String args[]) {
        double data1[] = {
            0, 0,
            -0.148105526820657, 0.0507420782620461,
            -1.13674727366735, 0.862892721126079,
            1.54366183541037, -1.13804802266371,
            -0.0923211737957721, 1.65649055346038,
            0.521764564424907, -2.11208211206815,
            0.843683397890515, 2.56656798707681,
            -2.70330819114831, -2.83452914666041,
            4.93683704766295, 3.95965377457266,
            -4.78314880243807, -2.23136673998374,
            6.24911547448071, 0.40543051970714,
            -6.76582567372196, 0.816818897274206,
            6.21142944093012, -4.247694573354,
            -5.29817270483491, 5.08246614505046,
            4.19557583927549, -5.35697380907112,
            -3.21572784971078, 7.89716072829777,
            0.485717553966479, -8.25930665413043,
            2.69490292018773, 10.9017252520684,
            -5.41090143363388, -10.400477539227,
            8.29423327234419, 9.10321370415527
        };

        TimeSeries ts = new TimeSeries();
        // Set the data for the two-dimensional time series
        ts.setSeriesValues(data1, 2);

        // Set up the vector autoregression model. Use default AR lag = 1.
        VectorAutoregression VAR1 = new VectorAutoregression(ts);

        // Get estimates of coefficient matrix A1
        double[][] coefs = VAR1.getEstimates();

        // Print estimates
```

```
        PrintMatrix pm = new PrintMatrix("Estimated Coefficient Matrix A1");
        pm.print(coefs);

        // Get h =1, 2, 3, 4 step ahead forecasts for all time points past
        // the default presample value (nT = 6)
        double[][] forecasts = VAR1.getForecasts();

        // Print forecasts
        pm = new PrintMatrix("VAR Forecasts");
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        String[] colLabels = {"(h=1,k=1)", "(h=1,k=2)", "(2,1)",
            "(2,2)", "(3,1)", "(3,2)", "(4,1)", "(4,2)"};
        pmf.setColumnLabels(colLabels);
        pmf.setNoRowLabels();
        pm.print(pmf, forecasts);
    }
}
```

Output

```
Estimated Coefficient Matrix A1
     0       1
0  -1.017  -0.296
1   0.273  -1.053

                               VAR Forecasts
  (h=1,k=1)  (h=1,k=2)   (2,1)   (2,2)    (3,1)   (3,2)    (4,1)   (4,2)
  0.094       2.366    -0.795  -2.466    1.538   2.381   -2.269  -2.088
 -1.617      -2.473     2.377   2.163   -3.058  -1.63     3.593   0.883
  3.589       2.248    -4.316  -1.389    4.802   0.286   -4.97    1.007
 -6.194      -2.824     7.137   1.286   -7.642   0.592    7.6    -2.706
  5.526       1.046    -5.932   0.405    5.916  -2.043   -5.415   3.764
 -6.478       1.276     6.214  -3.11    -5.403   4.969    4.028  -6.705
  6.642      -2.704    -5.959   4.659    4.685  -6.53    -2.837   8.154
 -5.064       6.166     3.329  -7.874   -1.059   9.199   -1.642  -9.976
  3.888      -6.796    -1.946   8.217   -0.449  -9.183    3.172   9.548
 -2.685       6.785     0.726  -7.877    1.591   8.492   -4.129  -8.509
  0.937      -9.193     1.765   9.936   -4.733  -9.982    7.767   9.221
  1.948       8.83     -4.593  -8.767    7.265   7.981   -9.752  -6.424
 -5.965     -10.746     9.247   9.69   -12.273  -7.683   14.759   4.746
  8.58        9.477   -11.532  -7.641   13.993   4.903  -15.687  -1.349
```

Chapter 20: Multivariate Analysis

Types

class ClusterKMeans 1117
class ClusterKNN 1134
class Dissimilarities 1139
class ClusterHierarchical 1146
class FactorAnalysis 1155
class DiscriminantAnalysis 1174

Usage Notes

Cluster Analysis

Clustering deals with the problem of segmenting or organizing a set of individual objects into groups or "clusters" of similar objects, so that objects within a cluster are more similar to each other than they are to objects in other clusters. In practice, objects are virtually anything for which attributes or features can be measured and quantified into a data matrix X. Typically a set of clusters is constructed from a training data set. Then the clusters are used to predict the class label for a new object.

The library includes three classes that perform clustering and classification: `ClusterKMeans`, `ClusterKNN`, and `ClusterHierarchical`. `ClusterKMeans` performs a K-means cluster analysis. The K-means method 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-squares differences. When no transfer occurs in a pass through the entire data set, the algorithm stops. The clustering can be 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.

`ClusterKNN` performs k-Nearest Neighbor, or KNN, cluster analysis. For a new object, KNN finds a cluster consisting of the object's K closest neighbors, where "closest" is in the sense of a specified distance metric, such as the L2 norm or the L1 norm. The new object then receives the class label that the majority of the objects in its neighborhood have (majority vote). The algorithm works with a training

set. For each observation in the data X (representing a new object), the distance to each object in a training set is calculated. The distances are sorted, and the smallest K distances are noted. The objects with the K-smallest distances form the cluster. Note that unlike K-Means, KNN does not provide a clustering of the data set. In a sense KNN forms clusters "on the fly" for the purpose of classification.

`ClusterHierarchical` performs a variant of hierarchical cluster analysis. `ClusterHierarchical` implements a bottom-up, or agglomerative approach. Starting with every individual object in a data set as a cluster, the two clusters (objects) that are closest in the sense of some distance metric are then combined into a new cluster. With the new set of n - 1 clusters, the process is repeated, so that at the h-th stage, there are n - h clusters. The clustering continues until there is only one cluster. Each level represents a different clustering of the data that can be evaluated using fit statistics and other metrics.

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, μ is the p vector of variable means, Λ 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.

ClusterKMeans class

```
public class com.imsl.stat.ClusterKMeans implements Serializable, Cloneable
```

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 ν_{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 ν_{im} is missing, otherwise δ 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.

An algorithm for choosing efficient initial seeds was proposed by Arthur and Vassilvitskii (2007), and is known as *K*-means++. This algorithm has advantages over choosing arbitrary or random initial seeds from the population of observations. Access the implementation of this algorithm in this class using the constructor with the signature ClusterKMeans(double[][], int) or by calling the setClusterCenters(Random) method.

Constructors

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

ClusterKMeans

```
public ClusterKMeans(double[][] x, int k)
```

Description

Constructor for `ClusterKMeans` using the K-means++ algorithm to select the initial seeds.

Parameters

`x` – a `double` matrix containing the observations to be clustered

`k` – an `int` indicating the number of clusters

ClusterKMeans

```
public ClusterKMeans(double[][] x, int k, Random r)
```

Description

Constructor for `ClusterKMeans` using the K-means++ algorithm to set the initial seeds.

Parameters

`x` – a `double` matrix containing the observations to be clustered

`k` – an `int` indicating the number of clusters

`r` – a Random object, the random number generator to be used in the KMeans++ algorithm
Note that `r` can be initialized with a seed for repeatable results.

Methods

compute

```
final public double[][] compute() throws ClusterKMeans.NoConvergenceException,
ClusterKMeans.ClusterNoPointsException
```

Description

Computes the cluster means.

Returns

a `double` matrix containing computed result

Exceptions

`NonnegativeFreqException` is thrown if a frequency is negative

`NonnegativeWeightException` is thrown if a weight is negative

`NoConvergenceException` is thrown if convergence did not occur within the maximum number of iterations

`ClusterNoPointsException` is thrown if the cluster centers yield a cluster with no points

getClusterCounts

`public int[] getClusterCounts()`

Description

Returns the number of observations in each cluster.

Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `IllegalStateException` exception.

Returns

an `int` array containing the number of observations in each cluster

getClusterMembership

`public int[] getClusterMembership()`

Description

Returns the cluster membership for each observation.

Note that the `compute()` method must be invoked first before invoking this method. Otherwise, the method throws a `IllegalStateException` exception.

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.

Note that the `compute` method must be invoked first before invoking this method. Otherwise, the method throws a `IllegalStateException` exception.

Returns

a `double` array containing the within sum of squares for each cluster

getInitialCenters

`public double[][] getInitialCenters()`

Description

Returns the initial cluster centers.

The cluster centers are those set by the K-Means++ center selection algorithm or those supplied to the constructor.

Returns

a `double` matrix containing the cluster centers

setFrequencies

```
public void setFrequencies(double[] frequencies)
```

Description

Sets the frequency for each observation.

Parameter

`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.

Parameter

`iterations` – an `int` scalar specifying the maximum number of iterations

Default: `interations` = 30

setWeights

```
public void setWeights(double[] weights)
```

Description

Sets the weight for each observation.

Parameter

`weights` – a `double` array of size `x.length` containing the weight for each observation

Default: `weights[]` = 1

Example 1: 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
```

Example 2: K-means Cluster Analysis with K-means++ Seeds

This example performs K-means cluster analysis on Fisher's iris data. The initial cluster seeds are generated using the K-means++ algorithm.

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

public class ClusterKMeansEx2 {

    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}
        };

        Random r = new Random(123457L);
        r.setMultiplier(16807);
        ClusterKMeans kmean = new ClusterKMeans(x, 3, r);

        double[][] cm = kmean.compute();
        double[][] cc = kmean.getInitialCenters();
        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 Centers from K-means++").print(cc);
        new PrintMatrix("Sum of Squares").print(wss);
        new PrintMatrix("Number of observations").print(nc);
    }
}
```

Output

```
           Cluster Means
     0       1       2       3
0  6.8500  3.0737  5.7421  2.0711
1  5.0060  3.4280  1.4620  0.2460
2  5.9016  2.7484  4.3935  1.4339

Cluster Centers from K-means++
    0    1    2    3
0  6.7  3.3  5.7  2.5
```

```
1  5.7  4.4  1.5  0.4
2  5.8  2.8  5.1  2.4

Sum of Squares
     0
0  23.879
1  15.151
2  39.821

Number of observations
   0
0  38
1  50
2  62
```

ClusterKMeans.NoConvergenceException class

`static public class com.imsl.stat.ClusterKMeans.NoConvergenceException extends com.imsl.IMSLException`

Convergence did not occur within the maximum number of iterations.

Constructors

ClusterKMeans.NoConvergenceException

`public ClusterKMeans.NoConvergenceException(String message)`

Description

Constructs a `NoConvergenceException` object.

Parameter

`message` – a `String` containing the error message

ClusterKMeans.NoConvergenceException

`public ClusterKMeans.NoConvergenceException(String key, Object[] arguments)`

Description

Constructs a `NoConvergenceException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ClusterKMeans.ClusterNoPointsException class

```
static public class com.imsl.stat.ClusterKMeans.ClusterNoPointsException
extends com.imsl.IMSLException
```

There is a cluster with no points

Constructors

ClusterKMeans.ClusterNoPointsException

```
public ClusterKMeans.ClusterNoPointsException(String message)
```

Description

Constructs a `ClusterNoPointsException` object.

Parameter

`message` – a `String` containing the error message

ClusterKMeans.ClusterNoPointsException

```
public ClusterKMeans.ClusterNoPointsException(String key, Object[] arguments)
```

Description

Constructs a `ClusterNoPointsException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ClusterKMeans.NonnegativeFreqException class

```
static public class com.imsl.stat.ClusterKMeans.NonnegativeFreqException
extends com.imsl.IMSLException
```

Frequencies must be nonnegative.

Constructors

ClusterKMeans.NonnegativeFreqException

```
public ClusterKMeans.NonnegativeFreqException(String message)
```

Description

Constructs a `NonnegativeFreqException` object.

Parameter

`message` – a `String` containing the error message

ClusterKMeans.NonnegativeFreqException

```
public ClusterKMeans.NonnegativeFreqException(String key, Object[] arguments)
```

Description

Constructs a `NonnegativeFreqException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ClusterKMeans.NonnegativeWeightException class

```
static public class com.imsl.stat.ClusterKMeans.NonnegativeWeightException
extends com.imsl.IMSLException
```

Weights must be nonnegative.

Constructors

ClusterKMeans.NonnegativeWeightException

```
public ClusterKMeans.NonnegativeWeightException(String message)
```

Description

Constructs a `NonnegativeWeightException` object.

Parameter

`message` – a `String` containing the error message

ClusterKMeans.NonnegativeWeightException

```
public ClusterKMeans.NonnegativeWeightException(String key, Object[] arguments)
```

Description

Constructs a `NonnegativeWeightException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

ClusterKNN class

```
public class com.imsl.stat.ClusterKNN implements Serializable, Cloneable
```

Perform a k-Nearest Neighbor classification.

`ClusterKNN` implements an algorithm to classify objects based on a training set. Among the simpler algorithms for classification, classifying a new object is essentially a majority vote of its closest `k` neighbors. `k` must be a positive integer and is typically small and odd. The method is straightforward in that the distance from the new point to every point in the training set is computed and sorted. The `k` closest points are examined and the new object is assigned to the class that is most common in that set. For the case `k = 1` the object is assigned to the class of its nearest neighbor.

The default distance method is the Euclidean distance, but other options are available by using the `setDistanceMethod` method. The supported methods are:

method	**Description**
L2_NORM	The Euclidean distance method, L_2 norm, defined as the sum of the squares of the difference of each coordinate. (Default)
L1_NORM	The rectilinear norm or city block method, L_1 norm, defined as the sum of the absolute values of the difference of each coordinate. This is most useful for integer input data.
INFINITY_NORM	The Chebyshev distance method, L_∞ norm, defined as the maximum of the absolute values of the difference of each coordinate.

For cases where the data are poorly scaled, it may be necessary to normalize the input data first. For example, if in a 2D space the X values range from 0 to 1 and the Y values, from 0 to 1000, the distance calculations will be dominated by the Y coordinate unless they are normalized.

Fields

INFINITY_NORM

`static final public int INFINITY_NORM`

Indicates the distance is computed using the L_∞ norm method. This is also known as the maximum difference or Chebyshev distance.

L1_NORM

`static final public int L1_NORM`

Indicates the distance is computed using the L_1 norm method. Also known as rectilinear distance or city block distance, it is most useful for integer input data.

L2_NORM

`static final public int L2_NORM`

Indicates the distance is computed using the L_2 norm, or Euclidean distance measurement.

Constructor

ClusterKNN

`public ClusterKNN(double[][] x, int[] c)`

Description

Constructor for `ClusterKNN`.

Parameters

`x` – A `double` matrix containing the known `x.length` observations of `x[0].length` variables.

`c` – An `int` array containing the categories for the `x.length` observations. All integer values are valid.

Methods

classify

`public int classify(double[] value, int k)`

Description

Classify an observation using `k` nearest neighbors.

Parameters

`value` – A `double` array of `x[0].length` variables containing the observations to classify.

`k` – An `int` containing the number of nearest neigbors to use. An odd value is recommended.

Returns

An `int` containing the cluster to which the observation belongs.

classify

`public int[] classify(double[][] value, int k)`

Description

Classify a set of observations using `k` nearest neighbors.

Parameters

`value` – A `double` matrix of `value.length` observations and `x[0].length` variables to classify.

`k` – An `int` containing the number of nearest neigbors to use. An odd value is recommended.

Returns

An `int` array containing the cluster to which each of the observations belong.

setDistanceMethod

`public void setDistanceMethod(int method)`

Description

Sets the distance calculation method to be used.

Parameter

`method` – An `int` identifying the distance calculation method to be used. By default, `method` = `L2_NORM`.

method	*Description*
`L2_NORM`	$\mathrm{d}(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$
`L1_NORM`	$\mathrm{d}(\mathbf{p},\mathbf{q}) = \sum_{i=1}^{n}\lvert q_i - p_i\rvert$
`INFINITY_NORM`	$\mathrm{d}(\mathbf{p},\mathbf{q}) = \max_i(\lvert p_i - q_i\rvert)$

Example: K-Nearest Neighbor Analysis

This example performs K-Nearest Neighbor analysis on Fisher's iris data. Three observations are withheld from the full data set and then classified by the iris type.

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

public class ClusterKNNEx1 {

    public static void main(String argv[]) {
        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},
// removed data. added to argument values for method classify
// { 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},
// removed data. added to argument values for method classify
//{ 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},
            // removed data. added to argument values for method classify
            //{ 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}
        };

        int[] y = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
            2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
            2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
            3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
            3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3
        };

        ClusterKNN k = new ClusterKNN(x, y);

        // the removed observations
        double[][] values = {
            {5.400, 3.900, 1.300, 0.400},
            {5.700, 2.600, 3.500, 1.000},
            {6.800, 3.200, 5.900, 2.300}
        };
        int[] classes = k.classify(values, 5);
        for (int i = 0; i < classes.length; i++) {
            System.out.println("Observation #" + (i + 1) + " is iris type "
                    + classes[i]);
        }
    }
}
```

Output

```
Observation #1 is iris type 1
Observation #2 is iris type 2
Observation #3 is iris type 3
```

Dissimilarities class

```
public class com.imsl.stat.Dissimilarities implements Serializable, Cloneable
```

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, use `row` = `true` in the `setRow` method.

The distance method and scaling option used by Dissimilarities can be set via methods `setDistanceMethod` and `setScalingOption`, respectively. For distance methods `L2_NORM`, `L1_NORM`, or `INFINITY_NORM`, each row of `x` is first scaled according to the value specified by the `setScalingOption` method. 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 no scaling is performed, the parameters in the following discussion are all 1.0 (see `setScalingOption`). 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, y_k denotes the corresponding element in the j-th column, and *ndstm* is the number of rows if differencing columns and the number of columns if differencing rows. For given z_i, the distance methods that allow scaling are defined as:

`distanceMethod`	**Metric**
`L2_NORM`	Euclidean distance (L_2 norm)
`L1_NORM`	Sum of the absolute differences (L_1 norm)
`INFINITY_NORM`	Maximum difference (L_∞ norm)

The following distance measures do not allow for scaling.

`distanceMethod`	**Metric**
`MAHALANOBIS`	Mahalanobis distance
`ABS_COSINE`	Absolute value of the cosine of the angle between the vectors
`ANGLE_IN_RADIANS`	Angle in radians (0, π) between the lines through the origin defined by the vectors
`CORRELATION_COEFFICIENT`	Correlation coefficient
`ABS_CORRELATION_COEFFICIENT`	Absolute value of the correlation coefficient
`EXACT_MATCHES`	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 from the `setIndex` method is omitted from the distance measure.

Fields

ABS_CORRELATION_COEFFICIENT

`static final public int ABS_CORRELATION_COEFFICIENT`

Indicates the absolute value of the correlation coefficient distance method.

ABS_COSINE

`static final public int ABS_COSINE`

Indicates the absolute value of the cosine of the angle between the vectors distance method.

ANGLE_IN_RADIANS

`static final public int ANGLE_IN_RADIANS`

Indicates the angle in radians $(0, \pi)$ between the lines through the origin defined by the vectors distance method.

CORRELATION_COEFFICIENT

`static final public int CORRELATION_COEFFICIENT`

Indicates the correlation coefficient distance method.

EXACT_MATCHES

`static final public int EXACT_MATCHES`

Indicates the number of exact matches distance method.

INFINITY_NORM

`static final public int INFINITY_NORM`

Indicates the maximum difference (L_∞ norm) distance method.

L1_NORM

`static final public int L1_NORM`

Indicates the sum of the absolute differences (L_1 norm) distance method.

L2_NORM

`static final public int L2_NORM`

Indicates the Euclidean distance method (L_2 norm).

MAHALANOBIS

`static final public int MAHALANOBIS`

Indicates the Mahalanobis distance method.

NO_SCALING

```
static final public int NO_SCALING
```

Indicates no scaling.

RANGE

```
static final public int RANGE
```

Indicates scaling by the range.

STD_DEV

```
static final public int STD_DEV
```

Indicates scaling by the standard deviation.

Constructors

Dissimilarities

```
public Dissimilarities(double[][] x)
```

Description

Constructor for `Dissimilarities`.

Parameter

`x` – A `double` matrix containing the data input matrix.

Methods

compute

```
public void compute() throws Dissimilarities.ScaleFactorZeroException,
Dissimilarities.ZeroNormException, Dissimilarities.NoPositiveVarianceException
```

Description

Computes a matrix of dissimilarities (or similarities) between the columns (or rows) of a matrix.

Exceptions

`ScaleFactorZeroException` is thrown when computations cannot continue because a scale factor is zero

`NoPositiveVarianceException` is thrown when no variable has positive variance

`ZeroNormException` is thrown when the Euclidean norm of a column is equal to zero

getDistanceMatrix

```
final public double[][] getDistanceMatrix()
```

Description

Returns the distance matrix.

Returns

A `double` matrix containing the distance matrix.

getDistanceMethod

```
public int getDistanceMethod()
```

Description

Returns the method used in computing the dissimilarities or similarities.

getIndex

```
public int[] getIndex()
```

Description

Returns the indices of the rows (columns) used in computing the distance measure.

getRow

```
public boolean getRow()
```

Description

Returns a `boolean` indicating whether distances are computed between rows or columns of `x`.

getScalingOption

```
public int getScalingOption()
```

Description

Returns the scaling option.

setDistanceMethod

```
public void setDistanceMethod(int distanceMethod)
```

Description

Sets the method to be used in computing the dissimilarities or similarities.

Parameter

`distanceMethod` – An `int` identifying the method to be used in computing the dissimilarities or similarities. Acceptable values of `distanceMethod` are:

distanceMethod	**Metric**
L2_NORM	Euclidean distance (L_2 norm)
L1_NORM	Sum of the absolute differences (L_1 norm)
INFINITY_NORM	Maximum difference (L_∞ norm)
MAHALANOBIS	Mahalanobis distance
ABS_COSINE	Absolute value of the cosine of the angle between the vectors
ANGLE_IN_RADIANS	Angle in radians (0, π) between the lines through the origin defined by the vectors
CORRELATION_COEFFICIENT	Correlation coefficient
ABS_CORRELATION_COEFFICIENT	Absolute value of the correlation coefficient
EXACT_MATCHES	Number of exact matches, where $x_i = y_i$.

See class description for more details. By default, `distanceMethod` = `L2_NORM`.

setIndex

```
public void setIndex(int[] indexArray)
```

Description

Sets the indices of the rows (columns).

Parameter

`indexArray` – An `int` array containing the indices of the columns (rows if `row` = `false`) to be used in computing the distance measure. By default, if `row` = `true`, `indexArray` = `0, 1, ..., x[0].length-1`. If `row` = `false`, `indexArray` = `0, 1, ..., x.length-1`, see `setRow`.

setRow

```
public void setRow(boolean row)
```

Description

Identifies whether distances are computed between rows or columns of `x`.

Parameter

`row` – A `boolean` identifying whether distances are computed between rows or columns of `x`. If `row` = `true`, distances are computed between the rows of `x`. Otherwise, distances between the columns of `x` are computed. By default, `row` = `true`.

setScalingOption

```
public void setScalingOption(int distanceScale)
```

Description

Sets the scaling option used if the `L2_NORM`, `L1_NORM`, or `INFINITY_NORM` distance methods are specified. See `setDistanceMethod`.

Parameter

`distanceScale` – An `int` containing the scaling option. By default, `distanceScale` = `NO_SCALING`.

distanceScale	**Method**
NO_SCALING	No scaling is performed.
STD_DEV	If setRow(false), scale each column by the standard deviation of the column. If setRow(true), scale each row by the standard deviation of the row.
RANGE	If setRow(false), scale each column by the range of the column. If setRow(true), scale each row by the range of the row.

Example: Dissimilarities

The following example illustrates the use of Dissimilarities for computing the Euclidean distance between the rows of a matrix:

```
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.}
        };

        Dissimilarities dist = new Dissimilarities(x);
        dist.compute();
        double[][] distanceMatrix = dist.getDistanceMatrix();

        new PrintMatrix().print(distanceMatrix);
    }
}
```

Output

```
   0  1  2  3
0  0  1  2  1
1  0  0  1  2
2  0  0  0  3
3  0  0  0  0
```

Dissimilarities.ScaleFactorZeroException class

`static public class com.imsl.stat.Dissimilarities.ScaleFactorZeroException extends com.imsl.IMSLException`

The computations cannot continue because a scale factor is zero.

Constructor

Dissimilarities.ScaleFactorZeroException

`public Dissimilarities.ScaleFactorZeroException(int index)`

Description

Constructs a `ScaleFactorZeroException`.

Parameter

`index` – An `int` which specifies the index of the scale factor array at which scale factor is zero.

Dissimilarities.ZeroNormException class

`static public class com.imsl.stat.Dissimilarities.ZeroNormException extends com.imsl.IMSLException`

The computations cannot continue because the Euclidean norm of the column is equal to zero.

Constructor

Dissimilarities.ZeroNormException

`public Dissimilarities.ZeroNormException(int index)`

Description

Constructs a `ZeroNormException`.

Parameter

`index` – An `int` which specifies the column index for which the norm has been found to be zero.

Dissimilarities.NoPositiveVarianceException class

```
static public class com.imsl.stat.Dissimilarities.NoPositiveVarianceException
extends com.imsl.IMSLException
```

No variable has positive variance. The Mahalanobis distances cannot be computed.

Constructor

Dissimilarities.NoPositiveVarianceException

```
public Dissimilarities.NoPositiveVarianceException()
```

Description

Constructs a `NoPositiveVarianceException`.

ClusterHierarchical class

```
public class com.imsl.stat.ClusterHierarchical implements Serializable,
Cloneable
```

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 transformation specified in the method `setTransformType`, based upon a similarity matrix. Only the upper triangular part of the input matrix is used.

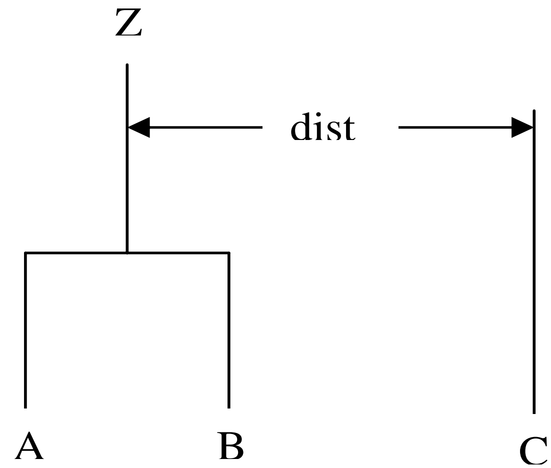
Hierarchical clustering in `ClusterHierarchical` proceeds as follows:

Initially, each data point is considered to be a cluster, numbered 1 to `n`, where `n` is the number of rows in the input matrix, `dist`.

1. If the input matrix contains similarities, the matrix is transformed to a distance matrix using the transform type specified by the method `setTransformType`. 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` in `setMethod` 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.



method	**Description**
LINKAGE_SINGLE	Single linkage (minimum distance). The distance from *Z* to *C* is the minimum of the distances (*A* to *C*, *B* to *C*).
LINKAGE_COMPLETE	Complete linkage (maximum distance). The distance from *Z* to *C* is the maximum of the distances (*A* to *C*, *B* to *C*).
LINKAGE_AVG_WITHIN_CLUSTERS	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).
LINKAGE_AVG_BETWEEN_CLUSTERS	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).
LINKAGE_WARDS	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.

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 `n`, where `n` is the number of rows in `dist`. The `n` - 1 clusters formed by merging clusters are numbered `n` + 1 to `n` + (`n` - 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 `transform` = `RECIPROCAL_ABS`, in the `setTransformType` method.

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` (p. 1139) may be used to compute the matrix `dist` for either the variables or observations.

Fields

LINKAGE_AVG_BETWEEN_CLUSTERS

`static final public int LINKAGE_AVG_BETWEEN_CLUSTERS`

Indicates the average distance between (average distance between objects in the two clusters) method.

LINKAGE_AVG_WITHIN_CLUSTERS

`static final public int LINKAGE_AVG_WITHIN_CLUSTERS`

Indicates the average distance within (average distance between objects within the merged cluster) method.

LINKAGE_COMPLETE

`static final public int LINKAGE_COMPLETE`

Indicates the complete linkage (maximum distance) method.

LINKAGE_SINGLE

`static final public int LINKAGE_SINGLE`

Indicates the single linkage (minimum distance) method.

LINKAGE_WARDS

`static final public int LINKAGE_WARDS`

Indicates the Ward's method.

MULTIPLICATION

`static final public int MULTIPLICATION`

Indicates transformation by multiplication by -1.0.

NONE

`static final public int NONE`

Indicates no transformation.

RECIPROCAL_ABS

`static final public int RECIPROCAL_ABS`

Indicates transformation by taking the reciprocal of the absolute value.

Constructors

ClusterHierarchical

`public ClusterHierarchical(double[][] dist)`

Description

Constructor for `ClusterHierarchical`.

Parameter

`dist` – A `double` symmetric matrix containing the distance (or similarity) matrix. Only the upper triangular part is used.

Methods

compute

`public void compute()`

Description

Performs a hierarchical cluster analysis.

getClusterLeftSons

`final public int[] getClusterLeftSons()`

Description

Returns the left sons of each merged cluster.

Returns

An `int` array containing the left sons of each merged cluster.

getClusterLevel

`final public 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 `n` + k was formed.

getClusterMembership

`final public int[] getClusterMembership(int nClusters)`

Description

Returns the cluster membership of each observation.

Parameter

`nClusters` – An `int` which specifies the desired number of clusters.

Returns

An `int` array containing the cluster membership of each observation.

getClusterRightSons

`final public int[] getClusterRightSons()`

Description

Returns the right sons of each merged cluster.

Returns

An `int` array containing the right sons of each merged cluster.

getMethod

`public int getMethod()`

Description

Returns the clustering method used.

getObsPerCluster

`final public int[] getObsPerCluster(int nClusters)`

Description

Returns the number of observations in each cluster.

Parameter

`nClusters` – An `int` which specifies the desired number of clusters.

Returns

An `int` array containing the number of observations in each cluster.

getTransformType

`public int getTransformType()`

Description

Returns the type of transformation.

setMethod

`public void setMethod(int method)`

Description

Sets the clustering method to be used.

Parameter

method – An int identifying the clustering method to be used. By default, method = LINKAGE_SINGLE.

method	Description
LINKAGE_SINGLE	Single linkage (minimum distance).
LINKAGE_COMPLETE	Complete linkage (maximum distance).
LINKAGE_AVG_WITHIN_CLUSTERS	Average distance within (average distance between objects within the merged cluster).
LINKAGE_AVG_BETWEEN_CLUSTERS	Average distance between (average distance between objects in the two clusters).
LINKAGE_WARDS	Ward's method (minimize the within-cluster sums of squares). For Ward's method, the elements of dist are assumed to be Euclidean distances.

setTransformType

```
public void setTransformType(int transform)
```

Description

Sets the type of transformation.

Parameter

transform – An int identifying the type of transformation applied to the measures in dist. By default, transform = NONE.

transform	Description
NONE	No transformation is required. The elements of dist are distances.
MULTIPLICATION	Convert similarities to distances by multiplication by -1.0.
RECIPROCAL_ABS	Convert similarities (usually correlations) to distances by taking the reciprocal of the absolute value.

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 com.imsl.stat.*;

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);
        dist.setScalingOption(Dissimilarities.STD_DEV);
        dist.compute();

        ClusterHierarchical clink
                = new ClusterHierarchical(dist.getDistanceMatrix());
        clink.setMethod(ClusterHierarchical.LINKAGE_AVG_WITHIN_CLUSTERS);
        clink.compute();

        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("\nObservations 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
```

FactorAnalysis class

`public class com.imsl.stat.FactorAnalysis implements Serializable, Cloneable`

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 Γ 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 Σ is the $p \times p$ population covariance matrix. Λ is the $p \times k$ matrix of factor loadings relating the factors f to

the observed variables x, and Ψ 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 Ψ 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.

Fields

ALPHA_FACTOR_ANALYSIS

```
static final public int ALPHA_FACTOR_ANALYSIS
```

Indicates alpha factor analysis.

CORRELATION_MATRIX

```
static final public int CORRELATION_MATRIX
```

Indicates correlation matrix.

GENERALIZED_LEAST_SQUARES

```
static final public int GENERALIZED_LEAST_SQUARES
```

Indicates generalized least squares method.

IMAGE_FACTOR_ANALYSIS

```
static final public int IMAGE_FACTOR_ANALYSIS
```

Indicates image factor analysis.

MAXIMUM_LIKELIHOOD

```
static final public int MAXIMUM_LIKELIHOOD
```

Indicates maximum likelihood method.

PRINCIPAL_COMPONENT_MODEL

```
static final public int PRINCIPAL_COMPONENT_MODEL
```

Indicates principal component model.

PRINCIPAL_FACTOR_MODEL

```
static final public int PRINCIPAL_FACTOR_MODEL
```

Indicates principal factor model.

UNWEIGHTED_LEAST_SQUARES

```
static final public int UNWEIGHTED_LEAST_SQUARES
```

Indicates unweighted least squares method.

VARIANCE_COVARIANCE_MATRIX

```
static final public int VARIANCE_COVARIANCE_MATRIX
```

Indicates variance-covariance matrix.

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.

Exception

`IllegalArgumentException` is thrown if `x.length`, and `x[0].length` are equal to 0.

Methods

getCorrelations

```
public double[][] getCorrelations() throws FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
FactorAnalysis.NonPositiveEigenvalueException
```

Description

Returns the unrotated factor loadings.

Returns

A `double` matrix containing the unrotated factor loadings.

getParameterUpdates

```
public double[] getParameterUpdates() throws FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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 FactorAnalysis.RankException,
FactorAnalysis.NotSemiDefiniteException,
FactorAnalysis.NotPositiveSemiDefiniteException,
FactorAnalysis.SingularException, FactorAnalysis.BadVarianceException,
FactorAnalysis.EigenvalueException,
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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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 Γ and the diagonal matrix Δ 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 Ψ 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 Ψ and Γ. (In the principal factor method, Ψ is assumed to be known. Given Ψ, estimates for Λ 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 Σ 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 Σ. 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 Λ in terms of Ψ and substituting the solution into the likelihood. This gives a criterion $\Phi(\Psi, \Lambda(\Psi))$, which is optimized with respect to Ψ. In the second stage, the estimates

$$\hat{\Lambda}$$

are obtained from the estimates for Ψ.

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, ρ, 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, Φ 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 Λ and eigenvectors Γ 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 ξ 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 ξ. 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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.*;

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(
                FactorAnalysis.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.*;

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(FactorAnalysis.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
```

FactorAnalysis.RankException class

`static public class com.imsl.stat.FactorAnalysis.RankException extends com.imsl.IMSLException`

Rank of covariance matrix error.

Constructors

FactorAnalysis.RankException

`public FactorAnalysis.RankException(String message)`

Description

Constructs a `RankException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.RankException

`public FactorAnalysis.RankException(String key, Object[] arguments)`

Description

Constructs a `RankException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

FactorAnalysis.NotPositiveSemiDefiniteException class

```
static public class
com.imsl.stat.FactorAnalysis.NotPositiveSemiDefiniteException extends
com.imsl.IMSLException
```

Covariance matrix not positive semi-definite.

Constructors

FactorAnalysis.NotPositiveSemiDefiniteException

```
public FactorAnalysis.NotPositiveSemiDefiniteException(String message)
```

Description

Constructs a `NotPositiveSemiDefiniteException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.NotPositiveSemiDefiniteException

```
public FactorAnalysis.NotPositiveSemiDefiniteException(String key, Object[]
arguments)
```

Description

Constructs a `NotPositiveSemiDefiniteException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

FactorAnalysis.NotSemiDefiniteException class

```
static public class com.imsl.stat.FactorAnalysis.NotSemiDefiniteException
extends com.imsl.IMSLException
```

Hessian matrix not semi-definite.

Constructors

FactorAnalysis.NotSemiDefiniteException

```
public FactorAnalysis.NotSemiDefiniteException(String message)
```

Description

Constructs a `NotSemiDefiniteException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.NotSemiDefiniteException

```
public FactorAnalysis.NotSemiDefiniteException(String key, Object[] arguments)
```

Description

Constructs a `NotSemiDefiniteException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

FactorAnalysis.SingularException class

```
static public class com.imsl.stat.FactorAnalysis.SingularException extends
com.imsl.IMSLException
```

Covariance matrix singular error.

Constructors

FactorAnalysis.SingularException

```
public FactorAnalysis.SingularException(String message)
```

Description

Constructs a `SingularException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.SingularException

```
public FactorAnalysis.SingularException(String key, Object[] arguments)
```

Description

Constructs a `SingularException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

FactorAnalysis.BadVarianceException class

```
static public class com.imsl.stat.FactorAnalysis.BadVarianceException extends
com.imsl.IMSLException
```

Bad variance error.

Constructors

FactorAnalysis.BadVarianceException

```
public FactorAnalysis.BadVarianceException(String message)
```

Description

Constructs a `BadVarianceException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.BadVarianceException

```
public FactorAnalysis.BadVarianceException(String key, Object[] arguments)
```

Description

Constructs a `BadVarianceException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

FactorAnalysis.EigenvalueException class

`static public class com.imsl.stat.FactorAnalysis.EigenvalueException extends com.imsl.IMSLException`

Eigenvalue error.

Constructors

FactorAnalysis.EigenvalueException

`public FactorAnalysis.EigenvalueException(String message)`

Description

Constructs a `EigenvalueException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.EigenvalueException

`public FactorAnalysis.EigenvalueException(String key, Object[] arguments)`

Description

Constructs a `EigenvalueException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

FactorAnalysis.NonPositiveEigenvalueException class

`static public class com.imsl.stat.FactorAnalysis.NonPositiveEigenvalueException extends com.imsl.IMSLException`

Non positive eigenvalue error.

Constructors

FactorAnalysis.NonPositiveEigenvalueException

```
public FactorAnalysis.NonPositiveEigenvalueException(String message)
```

Description

Constructs a `NonPositiveEigenvalueException` object.

Parameter

`message` – a `String` containing the error message

FactorAnalysis.NonPositiveEigenvalueException

```
public FactorAnalysis.NonPositiveEigenvalueException(String key, Object[]
arguments)
```

Description

Constructs a `NonPositiveEigenvalueException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

DiscriminantAnalysis class

```
public class com.imsl.stat.DiscriminantAnalysis
```

Performs a linear or a quadratic discriminant function analysis among several known groups.

`DiscriminantAnalysis` allows linear or a quadratic discrimination and the use of either reclassification, split sample, or the leaving-out-one methods in order to evaluate the rule. One or more observations can be added to the rule during each invocation of the `update` method.

`DiscriminantAnalysis` results in the measure of distance between the groups,(see `getMahalanobis` method), a table summarizing the classification results, (see `getClassTable`), a matrix containing the posterior probabilities of group membership for each classified observation, (see `getProbability`), the within-sample means, (see `getMeans`) and covariance matrices computed from their LU factorizations, (see `getCovariance`). The linear discriminant function coefficients are also computed, (see `getCoefficients` method).

All observations can be input during one call to the `update` method; this has the advantage of simplicity. Alternatively, one or more rows of observations can be input during separate calls to `update`. This does not require all observations be memory resident, a significant advantage with large data sets. Note, however, to classify the same data set requires a second pass of the data to the `classify` method. During the first pass to the `update` method the discriminant functions are computed while in the second

pass to the `classify` method the observations are classified. When known groups are available the method `getClassTable` is useful in comparing how well the alogorithm classifies. Multiple calls to the `classify` method are also allowed. The class table, `getClassTable`, is an accumulation of all observations classified. The class membership and probabilities, returned in `getClassMembership` and `getProbabilities`, will contain the membership for each observation from the most recent invocation of the `classify` method.

Pooled only and pooled with group covariance computation cannot be mixed. By default, both pooled and group covariance matrices will be computed. An IllegalStateException will be thrown if an attempt is made to change the covariance computation after the first call to the `update` method. See the `setCovarianceComputation` method for more details on specifying the covariance computation.

The within-group means are updated for all valid observations in `x`. Observations with invalid group numbers are ignored, as are observations with missing values (`Double.NaN`). The *LU* factorization of the covariance matrices are updated by adding (or deleting) observations via Givens rotations. See the `downdate` method to delete observations.

During the algorithm's training process, or each invocation of the `update` method, each observation in `x` is added to the means and the factorizations of the covariance matrices. Statistics of interest are computed: the linear discriminant functions, the prior probabilities, the log of the determinant of each of the covariance matrices, and a test statistic for testing that all of the within-group covariance matrices are equal. The matrix of Mahalanobis distances, which consists of the distances between the groups, 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 $\ln(p_i)$ 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

$$C^{-1} = \frac{1 - 2p^2 + 3p - 1}{6(p+1)(k-1)} \left(\sum_{i=1}^{k} \frac{1}{n_i} - \frac{1}{\sum_j n_j} \right)$$

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

$$D_i^2(x) = \begin{cases} (x - \overline{x_i})^T S_i^{-1} (x - \overline{x_i}) + ln|S_i| - 2ln(p_i) & \text{Linear or Quadratic, pooled, group} \\ (x - \overline{x_i})^T S_p^{-1} (x - \overline{x_i}) - 2ln(p_i) & \text{Linear, Pooled} \end{cases}$$

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`. If a group variable is provided and the group number is out of range, the classification table is not altered at this stage. If the reclassification method is specified, then all observations with no missing values 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 w_j and a frequency of 1.0. See Lachenbruch (1975, page 36) for the required adjustment.

Fields

LEAVE_OUT_ONE
`static final public int LEAVE_OUT_ONE`

Indicates leave-out-one classification method.

LINEAR
`static final public int LINEAR`

Indicates a linear discrimination method.

POOLED
`static final public int POOLED`

Indicates pooled covariances computation.

POOLED_GROUP
`static final public int POOLED_GROUP`

Indicates pooled, group covariances computation.

PRIOR_EQUAL
`static final public int PRIOR_EQUAL`

Indicates prior equal probabilities.

PRIOR_PROPORTIONAL
`static final public int PRIOR_PROPORTIONAL`

Indicates prior proportional probabilities.

QUADRATIC
`static final public int QUADRATIC`

Indicates a quadratic discrimination method.

RECLASSIFICATION
`static final public int RECLASSIFICATION`

Indicates reclassification classification method.

Constructor

DiscriminantAnalysis
`public DiscriminantAnalysis(int nVariables, int nGroups)`

Description

Constructs a `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

classify

```
public void classify(double[][] x) throws
DiscriminantAnalysis.SumOfWeightsNegException,
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Classify a set of observations using the linear or quadratic discriminant functions generated during the training process.

Parameter

`x` – a `double` matrix containing the observations with at least `nVariables` columns. The first `nVariables` columns correspond to the variables. Reclassification does not require group numbers be present. Any additional columns will be ignored.

Exceptions

`IllegalStateException` is thrown if the leave-out-one classification method is chosen.

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

`EmptyGroupException` is thrown when there are no observations in a group.

`CovarianceSingularException` is thrown when the variance-covariance matrix is singular.

classify

```
public void classify(double[][] x, int[] varIndex) throws
DiscriminantAnalysis.SumOfWeightsNegException,
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Classify a set of observations using the linear or quadratic discriminant functions generated during the training process.

Parameters

x – a double matrix containing the observations with at least nVariables columns. The columns indicated in varIndex correspond to the variables. Reclassification does not require group numbers be present. Additional columns will be ignored.

varIndex – an int array containing the column indices in x that correspond to the variables to be used in the analysis

Exceptions

IllegalStateException is thrown if the leave-out-one classification method is chosen.

SumOfWeightsNegException is thrown when the sum of the weights have become negative.

EmptyGroupException is thrown when there are no observations in a group.

CovarianceSingularException is thrown when the variance-covariance matrix is singular.

classify

```
public void classify(double[][] x, int[] frequencies, double[] weights) throws
DiscriminantAnalysis.SumOfWeightsNegException,
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Classify a set of observations and associated frequencies and weights using the linear or quadratic discriminant functions generated during the training process.

Parameters

x – a double matrix containing the observations with at least nVariables columns. The first nVariables columns correspond to the variables. Reclassification does not require group numbers be present. Any additional columns will be ignored.

frequencies – an int array containing the associated frequencies for each observation

weights – a double array containing the associated weights for each observation

Exceptions

IllegalStateException is thrown if the leave-out-one classification method is chosen

SumOfWeightsNegException is thrown when the sum of the weights have become negative

EmptyGroupException is thrown when there are no observations in a group

CovarianceSingularException is thrown when the variance-covariance matrix is singular

classify

```
public void classify(double[][] x, int[] group, int[] varIndex) throws
DiscriminantAnalysis.SumOfWeightsNegException,
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Classify a set of observations and compare against known groups using the linear or quadratic discriminant functions generated during the training process.

Parameters

`x` – a `double` matrix containing the observations with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables. Any additional columns will be ignored.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`varIndex` – an `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis

Exceptions

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative

`EmptyGroupException` is thrown when there are no observations in a group

`CovarianceSingularException` is thrown when the variance-covariance matrix is singular

classify

```
public void classify(double[][] x, int[] varIndex, int[] frequencies, double[]
weights) throws DiscriminantAnalysis.SumOfWeightsNegException,
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Classify a set of observations and associated frequencies and weights using the linear or quadratic discriminant functions generated during the training process.

Parameters

`x` – a `double` matrix containing the observations with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables. Reclassification does not require group numbers be present. Additional columns in `x` will be ignored.

`varIndex` – an `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis

`frequencies` – an `int` array containing the associated frequencies for each observation

`weights` – a `double` array containing the associated weights for each observation

Exceptions

`IllegalStateException` is thrown if the leave-out-one classification method is chosen

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative

`EmptyGroupException` is thrown when there are no observations in a group

`CovarianceSingularException` is thrown when the variance-covariance matrix is singular

classify

```
public void classify(double[][] x, int[] group, int[] varIndex, int[]
frequencies, double[] weights) throws
DiscriminantAnalysis.SumOfWeightsNegException,
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Classify a set of observations, associated frequencies and weights, and compare against known groups using the linear or quadratic discriminant functions generated during the training process.

Parameters

`x` – a `double` matrix containing the observations with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables. Additional columns are ignored.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`varIndex` – an `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis

`frequencies` – an `int` array containing the associated frequencies for each observation

`weights` – a `double` array containing the associated weights for each observation

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative

downdate

```
public void downdate(double[][] x, int[] group) throws
DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Removes a set of observations from the discriminant functions.

Parameters

`x` – a `double` matrix containing the observations to be removed, with at least `nVariables` columns. The first `nVariables` columns correspond to the variables. Any additional columns will be ignored.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

downdate

```
public void downdate(double[][] x, int[] group, int[] varIndex) throws
DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Removes a set of observations from the discriminant functions.

Parameters

`x` – a `double` matrix containing the observations to be removed, with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`varIndex` – an `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

downdate

```
public void downdate(double[][] x, int[] group, int[] frequencies, double[]
weights) throws DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Removes a set of observations and associated frequencies and weights from the discriminant functions.

Parameters

`x` – a `double` matrix containing the observations to be removed, with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`frequencies` – an `int` array containing the associated frequencies for each observation

`weights` – a `double` array containing the associated weights for each observation

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

downdate

```
public void downdate(double[][] x, int[] group, int[] varIndex, int[]
frequencies, double[] weights) throws
DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Removes a set of observations and associated frequencies and weights from the discriminant functions.

Parameters

`x` – a `double` matrix containing the observations to be removed, with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`varIndex` – an `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis

`frequencies` – an `int` array containing the associated frequencies for each observation

`weights` – a `double` array containing the associated weights for each observation

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

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. Note this will return the classmembership of the last set of observations classified.

Exception

`IllegalStateException` is thrown if no data has been classified.

getClassTable

`public double[][] getClassTable()`

Description

Returns the classification table.

Returns

an `nGroups` by `nGroups` `double` matrix containing the classification table. The accumulation of each observation that is classified and has a group number equal to 1, 2, ..., `nGroups` is entered into the table. If a known group is provided, the rows of the table correspond to the known group membership. The columns refer to the group to which the observation was classified. If a known group is not provided, the table will only contain the accumulated classified groups in the column coresponding to the group to which the observation was classified.

Exception

`IllegalStateException` is thrown if no data has been classified.

getCoefficients

```
public double[][] getCoefficients() throws
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Returns the linear discriminant function coefficients.

Returns

an nGroups by nVariables double matrix containing the linear discriminant function coefficients. The first column of the matrix contains the constant term, and the remaining columns contain the variable coefficients. The *i*-th row of the returned matrix corresponds to group *i*. The coefficients are always computed as linear discriminant function coefficients even when quadratic discrimination is specified.

Exceptions

EmptyGroupException is thrown when there are no observations in a group.

CovarianceSingularException is thrown when the variance-covariance matrix is singular.

getCovariance

```
public double[][][] getCovariance() throws
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Returns the array of covariances.

Returns

a g by nVariables by nVariables double array containing the covariances. Where, g = nGroups+1 if pooled, group covariance computation is specified or g=1 if pooled covariance computation is specified. 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.

If this method is invoked before classification, the unscaled covariance matrix will be returned.

Exceptions

EmptyGroupException is thrown when there are no observations in a group.

CovarianceSingularException is thrown when the variance-covariance matrix is singular.

getGroupCounts

```
public int[] getGroupCounts()
```

Description

Returns the group counts.

Returns

an int array of length nGroups containing the number of observations in each group. If an update has not preceeded the invocation of this method, an array of all zeros will be returned.

getMahalanobis

```
public double[][] getMahalanobis() throws
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Returns the Mahalanobis distances between the group means.

Returns

an `nGroups` by `nGroups double` matrix containing the Mahalanobis distances between the group means. For linear discrimination, the Mahalanobis distance

$$D^2_{ij}(x)$$

between group means i and j is computed using the within covariance matrix for group i in place of the pooled covariance matrix.

Exceptions

`EmptyGroupException` is thrown when there are no observations in a group.

`CovarianceSingularException` is thrown when the variance-covariance matrix is singular.

getMeans

```
public double[][] getMeans() throws DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Returns the variable means.

Returns

an `nGroups` by `nVariables double` matrix containing the variable means. The i-th row contains the variable means for group i.

If this method is invoked before classification, the unscaled means will be returned.

Exceptions

`EmptyGroupException` is thrown when there are no observations in a group.

`CovarianceSingularException` is thrown when the variance-covariance matrix is singular.

getNumberOfRowsMissing

```
public int getNumberOfRowsMissing()
```

Description

Returns the number of rows of data encountered containing missing values (`Double.NaN`).

Returns

an `int` representing the number of rows of data encountered containing missing values (`Double.NaN`) for the classification, group, weight, and/or frequency variables. If a row of data contains a missing value (`Double.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 array of length nGroups containing the prior probabilities for each group.

getProbability

```
public double[][] getProbability()
```

Description

Returns the posterior probabilities for each observation.

Returns

an x.length by nGroups double matrix containing the posterior probabilities for each observation. Note this will return the probabilities of the last set of observations classified.

Exception

IllegalStateException is thrown if no data has been classified.

getStatistics

```
public double[] getStatistics() throws
DiscriminantAnalysis.EmptyGroupException,
DiscriminantAnalysis.CovarianceSingularException
```

Description

Returns statistics.

Returns

a double array containing output statistics.

index	*Description*
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.

Exceptions

EmptyGroupException is thrown when there are no observations in a group.

CovarianceSingularException is thrown when the variance-covariance matrix is singular.

setClassificationMethod

```
public void setClassificationMethod(int method)
```

Description

Specifies the classification method to be either reclassification or leave-out-one.

Parameter

method – an int indicating the method of classification. Use class member RECLASSIFICATION or LEAVE_OUT_ONE. By default, the RECLASSIFICATION method is used.

setCovarianceComputation

```
public void setCovarianceComputation(int type)
```

Description

Specifies the covariance matrix computation to be either pooled or pooled, group.

Parameter

type – an int scalar indicating the type of covariance matrices to be computed. Use class member POOLED or POOLED_GROUP. By default, POOLED_GROUP is used.

setDiscriminationMethod

```
public void setDiscriminationMethod(int method)
```

Description

Specifies the discrimination method used to be either linear or quadratic discrimination.

Parameter

method – an int scalar indicating the method of discrimination. Use class member LINEAR or QUADRATIC. By default, the LINEAR method is used.

setPrior

```
public void setPrior(double[] prior)
```

Description

Specifies user supplied prior probabilities.

Parameter

prior – a double vector of length nGroups containing the prior probabilities for each group. The elements of prior should sum to 1.0. If the values of prior are less than 1.0e-20, they will be converted to the Math.log(1.0e-20). By default, the prior probablities are calculated to be equal, see setPrior(int).

setPrior

```
public void setPrior(int prior)
```

Description

Specifies the prior probabilities to be calculated as either equal or proportional priors.

Parameter

prior – an int specifying how to calculate prior probabilities as either equal or proportional prior probabilities. Use class member PRIOR_EQUAL to set equal prior probabilities, calculated as 1.0/nGroups. Use class member PRIOR_PROPORTIONAL to calculate the priors to be proportional to the sample size in each group. The sum of all prior probabilities is equal to 1.0. If the values calculated for the priors are less than 1.0e-20, they will be converted to the Math.log(1.0e-20). Prior probabilities are used in calculating statistics, coefficients, Mahalanobis, and classification probabilities. By default, PRIOR_EQUAL is used.

update

```
public void update(double[][] x, int[] group) throws
DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Trains a set of observations and associated frequencies and weights by performing a linear or quadratic discriminant function analysis among several known groups.

Parameters

x – a double matrix containing the observations with at least nVariables columns. The first nVariables correspond to the variables. Any additional columns will be ignored.

group – an int array containing the group numbers. The groups must be numbered 1,2, ..., nGroups for each observation.

Exception

SumOfWeightsNegException is thrown when the sum of the weights have become negative.

update

```
public void update(double[][] x, int[] group, int[] varIndex) throws
DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Trains a set of observations and associated frequencies and weights by performing a linear or quadratic discriminant function analysis among several known groups.

Parameters

x – a double matrix containing the observations with at least nVariables columns. The columns indicated in varIndex correspond to the variables. Any additional columns will be ignored.

group – an int array containing the group numbers. The groups must be numbered 1,2, ..., nGroups for each observation.

varIndex – an int array containing the column indices in x that correspond to the variables to be used in the analysis

Exception

SumOfWeightsNegException is thrown when the sum of the weights have become negative.

update

```
public void update(double[][] x, int[] group, int[] frequencies, double[]
weights) throws DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Trains a set of observations and associated frequencies and weights by performing a linear or quadratic discriminant function analysis among several known groups.

Parameters

`x` – a `double` matrix containing the observations with at least `nVariables` columns. The first `nVariables` correspond to the variables. Any additional columns will be ignored.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`frequencies` – an `int` array containing the associated frequencies for each observation

`weights` – a `double` array containing the associated weights for each observation

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

update

```
public void update(double[][] x, int[] group, int[] varIndex, int[]
frequencies, double[] weights) throws
DiscriminantAnalysis.SumOfWeightsNegException
```

Description

Trains a set of observations and associated frequencies and weights by performing a linear or quadratic discriminant function analysis among several known groups.

Parameters

`x` – a `double` matrix containing the observations with at least `nVariables` columns. The columns indicated in `varIndex` correspond to the variables.

`group` – an `int` array containing the group numbers. The groups must be numbered 1,2, ..., `nGroups` for each observation.

`varIndex` – an `int` array containing the column indices in `x` that correspond to the variables to be used in the analysis

`frequencies` – an `int` array containing the associated frequencies for each observation

`weights` – a `double` array containing the associated weights for each observation

Exception

`SumOfWeightsNegException` is thrown when the sum of the weights have become negative.

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

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[] group = new int[xorig.length];
int[] varIndex = {1, 2, 3, 4};

for (int i = 0; i < xorig.length; i++) {
    group[i] = (int) xorig[i][0];
}
int nvar = xorig[0].length - 1;

DiscriminantAnalysis da = new DiscriminantAnalysis(nvar, 3);
da.setCovarianceComputation(DiscriminantAnalysis.POOLED);
da.setClassificationMethod(DiscriminantAnalysis.RECLASSIFICATION);
da.update(xorig, group, varIndex);
da.classify(xorig, group, varIndex);
new PrintMatrix("Xmean:  ").print(da.getMeans());
new PrintMatrix("Coef:  ").print(da.getCoefficients());
new PrintMatrix("Counts: ").print(da.getGroupCounts());
new PrintMatrix("Stats: ").print(da.getStatistics());
int[] cm = da.getClassMembership();
int[][] cMem = new int[1][cm.length];
for (int i = 0; i < cm.length; i++) {
    cMem[0][i] = cm[i];
}
```

```
        new PrintMatrix("ClassMembership").setPageWidth(50).print(cMem);
        new PrintMatrix("ClassTable: ").print(da.getClassTable());
        double cov[][][] = da.getCovariance();
        for (int i = 0; i < cov.length; i++) {
            new PrintMatrix("Covariance Matrix " + i + " : ").print(cov[i]);
        }
        new PrintMatrix("Prior : ").print(da.getPrior());
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new java.text.DecimalFormat("0.00"));
        new PrintMatrix("PROB: ").print(pmf, da.getProbability());
        new PrintMatrix("MAHALANOBIS: ").print(da.getMahalanobis());
        System.out.println("nrmiss = " + da.getNumberOfRowsMissing());
    }
}
```

Output

```
           Xmean:
     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  1  2  3  4  5  6  7  8  9  10  11  12  13  14
0  1  1  1  1  1  1  1  1  1  1  1   1   1   1   1

   15  16  17  18  19  20  21  22  23  24  25  26  27  28  29
```

```
0  1    1    1    1    1    1    1    1    1    1    1    1    1    1    1

   30   31   32   33   34   35   36   37   38   39   40   41   42   43   44
0  1    1    1    1    1    1    1    1    1    1    1    1    1    1    1

   45   46   47   48   49   50   51   52   53   54   55   56   57   58   59
0  1    1    1    1    1    2    2    2    2    2    2    2    2    2    2

   60   61   62   63   64   65   66   67   68   69   70   71   72   73   74
0  2    2    2    2    2    2    2    2    2    2    3    2    2    2    2

   75   76   77   78   79   80   81   82   83   84   85   86   87   88   89
0  2    2    2    2    2    2    2    2    3    2    2    2    2    2    2

   90   91   92   93   94   95   96   97   98   99   100   101   102   103   104
0  2    2    2    2    2    2    2    2    2    2    3     3     3     3     3

   105   106   107   108   109   110   111   112   113   114   115   116   117   118   119
0   3     3     3     3     3     3     3     3     3     3     3     3     3     3     3

   120   121   122   123   124   125   126   127   128   129   130   131   132   133   134
0   3     3     3     3     3     3     3     3     3     3     3     3     3     2     3

   135   136   137   138   139   140   141   142   143   144   145   146   147   148   149
0   3     3     3     3     3     3     3     3     3     3     3     3     3     3     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  1.00  0.00  0.00
  1  1.00  0.00  0.00
  2  1.00  0.00  0.00
  3  1.00  0.00  0.00
  4  1.00  0.00  0.00
  5  1.00  0.00  0.00
  6  1.00  0.00  0.00
  7  1.00  0.00  0.00
  8  1.00  0.00  0.00
```

```
 9  1.00  0.00  0.00
10  1.00  0.00  0.00
11  1.00  0.00  0.00
12  1.00  0.00  0.00
13  1.00  0.00  0.00
14  1.00  0.00  0.00
15  1.00  0.00  0.00
16  1.00  0.00  0.00
17  1.00  0.00  0.00
18  1.00  0.00  0.00
19  1.00  0.00  0.00
20  1.00  0.00  0.00
21  1.00  0.00  0.00
22  1.00  0.00  0.00
23  1.00  0.00  0.00
24  1.00  0.00  0.00
25  1.00  0.00  0.00
26  1.00  0.00  0.00
27  1.00  0.00  0.00
28  1.00  0.00  0.00
29  1.00  0.00  0.00
30  1.00  0.00  0.00
31  1.00  0.00  0.00
32  1.00  0.00  0.00
33  1.00  0.00  0.00
34  1.00  0.00  0.00
35  1.00  0.00  0.00
36  1.00  0.00  0.00
37  1.00  0.00  0.00
38  1.00  0.00  0.00
39  1.00  0.00  0.00
40  1.00  0.00  0.00
41  1.00  0.00  0.00
42  1.00  0.00  0.00
43  1.00  0.00  0.00
44  1.00  0.00  0.00
45  1.00  0.00  0.00
46  1.00  0.00  0.00
47  1.00  0.00  0.00
48  1.00  0.00  0.00
49  1.00  0.00  0.00
50  0.00  1.00  0.00
51  0.00  1.00  0.00
52  0.00  1.00  0.00
53  0.00  1.00  0.00
54  0.00  1.00  0.00
55  0.00  1.00  0.00
56  0.00  0.99  0.01
57  0.00  1.00  0.00
58  0.00  1.00  0.00
59  0.00  1.00  0.00
60  0.00  1.00  0.00
61  0.00  1.00  0.00
62  0.00  1.00  0.00
63  0.00  0.99  0.01
64  0.00  1.00  0.00
```

```
 65  0.00  1.00  0.00
 66  0.00  0.98  0.02
 67  0.00  1.00  0.00
 68  0.00  0.96  0.04
 69  0.00  1.00  0.00
 70  0.00  0.25  0.75
 71  0.00  1.00  0.00
 72  0.00  0.82  0.18
 73  0.00  1.00  0.00
 74  0.00  1.00  0.00
 75  0.00  1.00  0.00
 76  0.00  1.00  0.00
 77  0.00  0.69  0.31
 78  0.00  0.99  0.01
 79  0.00  1.00  0.00
 80  0.00  1.00  0.00
 81  0.00  1.00  0.00
 82  0.00  1.00  0.00
 83  0.00  0.14  0.86
 84  0.00  0.96  0.04
 85  0.00  0.99  0.01
 86  0.00  1.00  0.00
 87  0.00  1.00  0.00
 88  0.00  1.00  0.00
 89  0.00  1.00  0.00
 90  0.00  1.00  0.00
 91  0.00  1.00  0.00
 92  0.00  1.00  0.00
 93  0.00  1.00  0.00
 94  0.00  1.00  0.00
 95  0.00  1.00  0.00
 96  0.00  1.00  0.00
 97  0.00  1.00  0.00
 98  0.00  1.00  0.00
 99  0.00  1.00  0.00
100  0.00  0.00  1.00
101  0.00  0.00  1.00
102  0.00  0.00  1.00
103  0.00  0.00  1.00
104  0.00  0.00  1.00
105  0.00  0.00  1.00
106  0.00  0.05  0.95
107  0.00  0.00  1.00
108  0.00  0.00  1.00
109  0.00  0.00  1.00
110  0.00  0.01  0.99
111  0.00  0.00  1.00
112  0.00  0.00  1.00
113  0.00  0.00  1.00
114  0.00  0.00  1.00
115  0.00  0.00  1.00
116  0.00  0.01  0.99
117  0.00  0.00  1.00
118  0.00  0.00  1.00
119  0.00  0.22  0.78
120  0.00  0.00  1.00
```

```
121  0.00  0.00  1.00
122  0.00  0.00  1.00
123  0.00  0.10  0.90
124  0.00  0.00  1.00
125  0.00  0.00  1.00
126  0.00  0.19  0.81
127  0.00  0.13  0.87
128  0.00  0.00  1.00
129  0.00  0.10  0.90
130  0.00  0.00  1.00
131  0.00  0.00  1.00
132  0.00  0.00  1.00
133  0.00  0.73  0.27
134  0.00  0.07  0.93
135  0.00  0.00  1.00
136  0.00  0.00  1.00
137  0.00  0.01  0.99
138  0.00  0.19  0.81
139  0.00  0.00  1.00
140  0.00  0.00  1.00
141  0.00  0.00  1.00
142  0.00  0.00  1.00
143  0.00  0.00  1.00
144  0.00  0.00  1.00
145  0.00  0.00  1.00
146  0.00  0.01  0.99
147  0.00  0.00  1.00
148  0.00  0.00  1.00
149  0.00  0.02  0.98

        MAHALANOBIS:
      0       1        2
0    0       89.864  179.385
1   89.864   0        17.201
2  179.385  17.201    0

nrmiss = 0
```

DiscriminantAnalysis.SumOfWeightsNegException class

```
static public class com.imsl.stat.DiscriminantAnalysis.SumOfWeightsNegException
extends com.imsl.IMSLException
```

The sum of the weights have become negative.

Constructors

DiscriminantAnalysis.SumOfWeightsNegException

```
public DiscriminantAnalysis.SumOfWeightsNegException(String message)
```

Description

The sum of the weights have become negative.

Parameter

`message` – a `String` containing the error message

DiscriminantAnalysis.SumOfWeightsNegException

```
public DiscriminantAnalysis.SumOfWeightsNegException(String key, Object[]
arguments)
```

Description

The sum of the weights have become negative.

Parameters

`key` – a `String` containing the exception message

`arguments` – an array containing arguments used within the error message string

DiscriminantAnalysis.EmptyGroupException class

```
static public class com.imsl.stat.DiscriminantAnalysis.EmptyGroupException
extends com.imsl.IMSLException
```

There are no observations in a group. Cannot compute statistics.

Constructors

DiscriminantAnalysis.EmptyGroupException

```
public DiscriminantAnalysis.EmptyGroupException(String message)
```

Description

There are no observations in a group. Cannot compute statistics.

Parameter

`message` – a `String` containing the exception message

DiscriminantAnalysis.EmptyGroupException

```
public DiscriminantAnalysis.EmptyGroupException(String key, Object[] arguments)
```

Description

There are no observations in a group. Cannot compute statistics.

Parameters

`key` – a `String` containing the key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

DiscriminantAnalysis.CovarianceSingularException class

```
static public class
com.imsl.stat.DiscriminantAnalysis.CovarianceSingularException extends
com.imsl.IMSLException
```

The variance-covariance matrix is singular.

Constructors

DiscriminantAnalysis.CovarianceSingularException

```
public DiscriminantAnalysis.CovarianceSingularException(String message)
```

Description

The variance-covariance matrix is singular.

Parameter

`message` – a `String` containing the exception message

DiscriminantAnalysis.CovarianceSingularException

```
public DiscriminantAnalysis.CovarianceSingularException(String key, Object[]
arguments)
```

Description

The variance-covariance matrix is singular.

Parameters

`key` – a `String` containing the key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

Chapter 21: Survival and Reliability Analysis

Types

class KaplanMeierECDF 1199
class KaplanMeierEstimates 1203
class ProportionalHazards 1211
class LifeTables 1231

Usage Notes

Survival Analysis

The functions described in this chapter have primary application in the areas of reliability and life testing, but they may find application in any situation in which analysis of binomial events over time is of interest. Kalbfleisch and Prentice (1980), Elandt-Johnson and Johnson (1980), Lee (1980), Gross and Clark (1975), Lawless (1982), and Chiang (1968) and Tanner and Wong (1984) are references for discussing the models and methods desribed in this chapter. Function KaplanMeierEstimates produces Kaplan-Meier (product-limit) estimates of the survival distribution in a single population. Function ProportionalHazards computes the parameter estimates in a proportional hazards model. Function SurvivalGLM fits any of several generalized linear models for survival data, and SurvivalEstimates computes estimates of survival probabilities based upon the same models. Function LifeTables computes and (optionally) prints an actuarial table based either upon a cohort followed over time or a cross-section of a population.

KaplanMeierECDF class

```
public class com.imsl.stat.KaplanMeierECDF implements Serializable, Cloneable
```

Computes the Kaplan-Meier reliability function estimates or the CDF based on failure data that may be

multi-censored.

The Kaplan-Meier (K-M) Product Limit procedure provides simple estimates of the reliability function or the CDF based on failure data that may be multi-censored. No underlying probability model is assumed; K-M estimation is an empirical (non-parametric) procedure. Exact times of failure are required.

Consider a situation in which we are reliability testing n (non-repairable) units taken randomly from a population. We are investigating the population to determine if its failure rate is acceptable. In the typical test scenario, we have a fixed time T to run the units to see if they survive or fail. The data obtained are called Censored Type I data.

During the T hours of test we observe r failures (where r can be any number from 0 to n). The failure times are $t_1, t_2, \ldots, t_r$, and there are $(n-r)$ units that survived the entire T-hour test without failing. Note that T is fixed in advance, and r is an output of the testing, since we don't know how many failures will occur until the test is run. Note that we assume the exact times of failure are recorded when they occur.

This type of data is also called "right censored" data since the times of failure to the right (i.e., larger than T) are missing. The steps for calculating K-M estimates are the following:

1. Order the actual failure times from t_1 through t_r, where there are r failures
2. Corresponding to each t_i, associate the number n_i with n_i = the number of operating units just before the ith failure occurred at time t_i
3. First estimate the survival $R(t_1) = (n_1 - 1)/n_1$
4. Estimate each ensuing survival $R(t_i) = R(t_{i-1})(n_i - 1)/n_1$, $i > 1$
5. Estimate the CDF $F(t_i) = 1 - R(t_i)$, $i = 1, 2, \ldots$

Note that non-failed units taken off testing (i.e., right-censored) only count up to the last actual failure time before they were removed. They are included in the n_i counts up to and including that failure time, but not after.

Constructor

KaplanMeierECDF

```
public KaplanMeierECDF(double[] t)
```

Description

Constructor for `KaplanMeierECDF`.

Parameter

`t` – a `double` array containing the failure times.

Methods

evaluateCDF

```
public double[] evaluateCDF()
```

Description

Computes the empirical CDF and returns the CDF values up to, but not including the time values returned by `getTimes`.

Returns

a `double` array of CDF values.

getNumberOfPoints

```
public int getNumberOfPoints()
```

Description

Retrieves the number of points in the empirical CDF

Returns

an `int` containing the number of points in the empirical CDF.

Exception

`IllegalStateException` is thrown if the CDF has not been evaluated.

getTimes

```
public double[] getTimes()
```

Description

Retrieves the time values where the step function CDF jumps to a greater value. This array has right-censored values of `t` removed.

Returns

a `double` array of time values.

Exception

`IllegalStateException` is thrown if the CDF has not been evaluated.

setCensor

```
public void setCensor(int[] censor)
```

Description

Set flags to note right-censoring

Parameter

`censor` – an `int` array of 0 or 1 flags to note right-censoring. Values of 0 = continue to use datum and 1 = remove datum. If this method is not called, no data is right-censored.

setFrequency

```
public void setFrequency(int[] freq)
```

Description

Sets the frequency for each entry in `t`

Parameter

`freq` – a `double` array containing the repeat count for each entry in `t`.

Example: Kaplan Meier Empirical CDF

This example illustrates the K-M procedure. Assume 20 units are on life test and 6 failures occur at the following times: 10, 32, 56, 98, 122, and 181 hours. There were 4 working units removed from the test for other experiments at the following times: 50, 100, 125, and 150 hours. The remaining 10 working units were removed from the test at 200 hours. The K-M estimates for this life test are:

R(10) = 19/20
R(32) = 19/20 x 18/19
R(56) = 19/20 x 18/19 x 16/17
R(98) = 19/20 x 18/19 x 16/17 x 15/16
R(122) = 19/20 x 18/19 x 16/17 x 15/16 x 13/14
R(181) = 19/20 x 18/19 x 16/17 x 15/16 x 13/14 x 10/11

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

public class KaplanMeierECDFEx1 {

    public static void main(String args[]) {
        double y[] = {
            10.0, 32.0, 56.0, 98.0, 122.0, 181.0, 50.0,
            100.0, 125.0, 150.0, 200.0
        };
        int freq[] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 10};
        int censor[] = {0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1};

        KaplanMeierECDF km = new KaplanMeierECDF(y);
        km.setFrequency(freq);
        km.setCensor(censor);
        double[] fx = km.evaluateCDF();

        int ntimes = km.getNumberOfPoints();
        System.out.println("Number of points = " + ntimes);

        double[] x = km.getTimes();
        PrintMatrix p
                = new PrintMatrix("CDF = 1 - survival of life test subjects");
        p.print(fx);
        p.setTitle("Times of change in CDF");
        p.print(x);
    }
}
```

Output

```
Number of points = 6
CDF = 1 - survival of life test subjects
     0
0  0.05
1  0.1
2  0.153
3  0.206
4  0.263
5  0.33

Times of change in CDF
    0
0   10
1   32
2   56
3   98
4  122
5  181
```

KaplanMeierEstimates class

```
public class com.imsl.stat.KaplanMeierEstimates implements Serializable,
Cloneable
```

Computes Kaplan-Meier (or product-limit) estimates of survival probabilities for a sample of failure times that possibly contain right consoring.

Class `KaplanMeierEstimates` computes Kaplan-Meier (or product-limit) estimates of survival probabilities for a sample of failure times that can be right censored or exact times. A survival probability $S(t)$ is defined as 1 - $F(t)$, where $F(t)$ is the cumulative distribution function of the failure times t. Greenwood's estimate of the standard errors of the survival probability estimates are also computed. (See Kalbfleisch and Prentice, 1980, pages 13 and 14.)

Let (t_i, δ_i), for i = 1,..., n denote the failure censoring times and the censoring codes for the n observations in a single sample. Here, $t_i = x_{i-l,responseIndex}$ is a failure time if δ_i is 0, where $\delta_i = x_{i-l,censorIndex}$. Also, t_i is a right censoring time if δ_i is 1. Rows in `x` containing values other than 0 or 1 for δ_i are ignored. Let the number of observations in the sample that have not failed by time $s_{(t)}$ be denoted by $n_{(t)}$, where $s_{(t)}$ is an ordered (from smallest to largest) listing of the distinct failure times (censoring times are omitted). Then the Kaplan-Meier estimate of the survival probabilities is a step function, which in the interval from $s_{(i)}$ to $s_{(i+1)}$ (including the lower endpoint) is given by

$$\hat{S}(t) = \prod_{j=1}^{i} \left(\frac{n_{(j)} - d_{(j)}}{n_{(j)}} \right)$$

where $d_{(j)}$ denotes the number of failures occurring at time $s_{(j)}$, and $n_{(j)}$ is the number of observations that have not failed prior to $s_{(j)}$.

Note that one row of `x` may correspond to more than one failed (or censored) observation when the frequency option is in effect (see `setFrequencyColumn`). The Kaplan-Meier estimate of the survival probability prior to time $s_{(1)}$ is 1.0, while the Kaplan-Meier estimate of the survival probability after the last failure time is not defined.

Greenwood's estimate of the variance of

$$\hat{S}(t)$$

in the interval from $s_{(i)}$ to $s_{(i+1)}$ is given as

$$\text{est.var}(\hat{S}(t)) = \hat{S}^2(t) \sum_{j=1}^{i} \frac{d_{(j)}}{n_{(j)}(n_{(j)} - d_{(j)})}$$

`KaplanMeierEstimates` computes the single sample estimates of the survival probabilities for all samples of data included in `x` during a single call. This is accomplished through the stratum column of `x`, which if present, must contain a distinct code for each sample of observations (see `setStratumColumn`). If a stratum column is not specified, there is no grouping , and all observations are assumed to come from the same sample.

When failures and right-censored observations are tied and the data are to be sorted by `KaplanMeierEstimates` (`setSorted(true)` is not used), `KaplanMeierEstimates` assumes that the time of censoring for the tied-censored observations is immediately after the tied failure (within the same sample). When `setSorted(true)` is used, the data are assumed to be sorted from smallest to largest according to the response time column of `x` within each stratum (see `setResponseColumn`). Furthermore, a small increment of time is assumed (theoretically) to elapse between the failed and censored observations that are tied (in the same sample). Thus, when `setSorted(true)` is used, the user must sort all of the data in `x` from smallest to largest according to the response time column (and the stratum column, if set). By appropriate sorting of the observations, the user can handle censored and failed observations that are tied in any manner desired.

Constructor

KaplanMeierEstimates

```
public KaplanMeierEstimates(double[][] x)
```

Description

Constructor for `KaplanMeierEstimates`.

Parameter

`x` – a `double` matrix containing the data, including optional data. By default it is assumed the response times are in column 0.

Methods

getCensorColumn

```
public int getCensorColumn()
```

Description

Returns the column index of x containing the optional censoring code for each observation.

Returns

an `int` specifying the column index of x containing the optional censoring code for each observation.

getFrequencyColumn

```
public int getFrequencyColumn()
```

Description

Returns the column index of x containing the frequency of response for each observation.

Returns

an `int` specifying the column index of x containing the frequency of response for each observation.

getGroupTotal

```
public int getGroupTotal(double groupValue)
```

Description

Returns the total number in the group for the specified group value.

Parameter

`groupValue` – a `double` specifying the group value.

Returns

an `int` representing the total number in the group which has value `groupValue`.

getLogLikelihood

```
public double getLogLikelihood(double groupValue)
```

Description

Returns the Kaplan-Meier log-likelihood of the group with the specified group value.

The Kaplan-Meier log-likelihood is computed as:

$$\ell = \sum_j d_{(j)} \ln d_{(j)} + (n_{(j)} - d_{(j)}) \ln(n_{(j)} - d_{(j)}) - n_{(j)} \ln n_{(j)}$$

where the sum is with respect to the distinct failure times $s_{(j)}$.

Parameter

`groupValue` – a `double` specifying the group value.

Returns

a `double` representing the Kaplan-Meier log-likelihood of the group which has value `groupValue`.

getNumberAtRisk

```
public int[] getNumberAtRisk()
```

Description

Returns the number of individuals at risk at each failure point.

Returns

an `int` array containing the number of individuals at risk at each failure point.

getNumberOfFailures

```
public int[] getNumberOfFailures()
```

Description

Returns the number of failures which occurred at each failure point.

Returns

an `int` array containing the number of failures which occurred at each failure point.

getNumberOfRowsMissing

```
public int getNumberOfRowsMissing()
```

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`.

getResponseColumn

```
public int getResponseColumn()
```

Description

Returns the column index of `x` containing the response time for each observation.

Returns

an `int` specifying the column index of `x` containing the response time for each observation.

getStandardErrors

```
public double[] getStandardErrors()
```

Description

Returns Greenwood's estimated standard errors.

Returns

a `double` array containing Greenwood's estimate of the standard errors for the survival probabilities.

getStratumColumn

```
public int getStratumColumn()
```

Description

Returns the column index of `x` containing the stratum number for each observation.

Returns

an `int` specifying the column index of `x` containing the stratum number for each observation.

getSurvivalProbabilities

```
public double[] getSurvivalProbabilities()
```

Description

Returns the estimated survival probabilities.

Returns

a `double` array containing the estimated survival probabilities.

getTotalNumberOfFailures

```
public int getTotalNumberOfFailures(double groupValue)
```

Description

Returns the total number failing in the group for the specified group value.

Parameter

`groupValue` – a `double` specifying the group value.

Returns

an `int` representing the total number failing in the group which has value `groupValue`.

setCensorColumn

```
public void setCensorColumn(int censorIndex)
```

Description

Sets the column index of `x` containing the optional censoring code for each observation.

Parameter

`censorIndex` – an `int` specifying the column index of `x` containing the optional censoring code for each observation. If `x[i][censorIndex]` equals 0, the failure time `x[i][responseIndex]` is treated as an exact time of failure. Otherwise, it is treated as right-censored time. Default: It is assumed that there is no censor code column in `x`. All observations are assumed to be exact failure times.

setFrequencyColumn

```
public void setFrequencyColumn(int frequencyIndex)
```

Description

Sets the column index of `x` containing the frequency of response for each observation.

Parameter

`frequencyIndex` – an `int` specifying the column index of `x` containing the frequency of response for each observation. Default: It is assumed that there is no frequency response column recorded in `x`. Each observation in the data array is assumed to be for a single failure.

setResponseColumn

```
public void setResponseColumn(int responseIndex)
```

Description

Sets the column index of `x` containing the response time for each observation.

Parameter

`responseIndex` – an `int` specifying the column index of `x` containing the response time for each observation. The interpretation of these times as either right-consored or exact failure times depends on the setting of the censor codes in the censor code column. See method `setCensorColumn`. Default: `responseIndex = 0`.

setSorted

```
public void setSorted(boolean isSorted)
```

Description

Sets the `boolean` to indicate that the column of response times in `x` are already sorted.

Parameter

`isSorted` – a `boolean` indicating whether or not column `responseIndex` of `x` is already sorted. `isSorted` equal to `true` indicates that column `responseIndex` of `x` is already sorted. Otherwise, a detached sort is performed prior to analysis. If sorting is performed, all censored individuals are assumed to follow tied failures. Default: It is assumed that column `responseIndex` of `x` is not sorted, so a detached sort is performed.

setStratumColumn

```
public void setStratumColumn(int stratumIndex)
```

Description

Sets the column index of `x` containing the stratum number for each observation.

Parameter

`stratumIndex` – an `int` specifying the column index of `x` containing the stratum number for each observation. Column `stratumIndex` of `x` contains a unique value for each stratum in the data. Kaplan-Meier estimates are computed within each stratum. Default: It is assumed that there is no stratum number column recorded `x`. The data is assumed to come from one statum.

Example : KaplanMeierEstimates

The following example is taken from Kalbfleisch and Prentice (1980, page 1). The first column in x contains the death/censoring times for rats suffering from vaginal cancer. The second column contains information as to which of two forms of treatment were provided, while the third column contains the censoring code. Finally, the fourth column contains the frequency of each observation. The product-limit estimates of the survival probabilities are computed for both groups along with their standard error estimates. Tables containing these values along with other statistics are printed.

```
import java.text.*;
import com.imsl.stat.*;

public class KaplanMeierEstimatesEx1 {

    public static void main(String args[]) {
        double[][] x = {
            {143, 5, 0, 1}, {164, 5, 0, 1}, {188, 5, 0, 2}, {190, 5, 0, 1},
            {192, 5, 0, 1}, {206, 5, 0, 1}, {209, 5, 0, 1}, {213, 5, 0, 1},
            {216, 5, 0, 1}, {220, 5, 0, 1}, {227, 5, 0, 1}, {230, 5, 0, 1},
            {234, 5, 0, 1}, {246, 5, 0, 1}, {265, 5, 0, 1}, {304, 5, 0, 1},
            {216, 5, 1, 1}, {244, 5, 1, 1}, {142, 7, 0, 1}, {156, 7, 0, 1},
            {163, 7, 0, 1}, {198, 7, 0, 1}, {205, 7, 0, 1}, {232, 7, 0, 2},
            {233, 7, 0, 4}, {239, 7, 0, 1}, {240, 7, 0, 1}, {261, 7, 0, 1},
            {280, 7, 0, 2}, {296, 7, 0, 2}, {323, 7, 0, 1}, {204, 7, 1, 1},
            {344, 7, 1, 1}
        };
        int nobs = x.length, censorIndex = 2, frequencyIndex = 3;
        int stratumIndex = 1, responseIndex = 0;
        int i, groupValue;

        // Get Kaplan-Meir Estimates
        KaplanMeierEstimates km = new KaplanMeierEstimates(x);
        km.setCensorColumn(censorIndex);
        km.setFrequencyColumn(frequencyIndex);
        km.setStratumColumn(stratumIndex);
        int[] atRisk = km.getNumberAtRisk();
        int[] nFailing = km.getNumberOfFailures();
        double[] prob = km.getSurvivalProbabilities();
        double[] se = km.getStandardErrors();

        // Print Results
        NumberFormat nf = NumberFormat.getInstance();
        nf.setMaximumFractionDigits(4);
        nf.setMinimumFractionDigits(4);
        i = 0;
        while (i < nobs) {
            groupValue = (int) x[i][stratumIndex];
            System.out.println("\n       Kaplan-Meier Survival Probabilities");
            System.out.println("               For Group Value = "
                    + groupValue);
            System.out.println("\nNumber     Number"
                    + "                Survival     Estimated");
            System.out.println("at risk    Failing     Time"
                    + "     Probability   Std. Error");
            while (i < nobs && ((int) x[i][stratumIndex] == groupValue)) {
```

```
                if ((int) x[i][censorIndex] != 1) {
                    System.out.println("  " + atRisk[i] + "          "
                            + nFailing[i] + "" + "         "
                            + ((int) x[i][responseIndex]) + "       "
                            + nf.format(prob[i]) + "        "
                            + nf.format(se[i]));
                }
                i++;
            }
            System.out.println("\nTotal number in group = "
                    + km.getGroupTotal(groupValue));
            System.out.println("Total number failing = "
                    + km.getTotalNumberOfFailures(groupValue));
            System.out.println("Product Limit likelihood = "
                    + nf.format(km.getLogLikelihood(groupValue)));
        }
        System.out.println("\nThe number of rows of x with missing values is "
                + km.getNumberOfRowsMissing());
    }
}
```

Output

```
       Kaplan-Meier Survival Probabilities
               For Group Value = 5

Number     Number                Survival     Estimated
at risk    Failing     Time     Probability   Std. Error
  19          1         143       0.9474        0.0512
  18          1         164       0.8947        0.0704
  17          2         188       0.7895        0.0935
  15          1         190       0.7368        0.1010
  14          1         192       0.6842        0.1066
  13          1         206       0.6316        0.1107
  12          1         209       0.5789        0.1133
  11          1         213       0.5263        0.1145
  10          1         216       0.4737        0.1145
  8          1         220       0.4145        0.1145
  7          1         227       0.3553        0.1124
  6          1         230       0.2961        0.1082
  5          1         234       0.2368        0.1015
  3          1         246       0.1579        0.0934
  2          1         265       0.0789        0.0728
  1          1         304       0.0000        ?

Total number in group = 19
Total number failing = 17
Product Limit likelihood = -49.1692

       Kaplan-Meier Survival Probabilities
               For Group Value = 7

Number     Number                Survival     Estimated
at risk    Failing     Time     Probability   Std. Error
```

```
  21          1         142       0.9524        0.0465
  20          1         156       0.9048        0.0641
  19          1         163       0.8571        0.0764
  18          1         198       0.8095        0.0857
  16          1         205       0.7589        0.0941
  15          2         232       0.6577        0.1053
  13          4         233       0.4554        0.1114
  9          1         239       0.4048        0.1099
  8          1         240       0.3542        0.1072
  7          1         261       0.3036        0.1031
  6          2         280       0.2024        0.0902
  4          2         296       0.1012        0.0678
  2          1         323       0.0506        0.0493

Total number in group = 21
Total number failing = 19
Product Limit likelihood = -50.4277

The number of rows of x with missing values is 0
```

ProportionalHazards class

`public class com.imsl.stat.ProportionalHazards implements Serializable, Cloneable`

Analyzes survival and reliability data using Cox's proportional hazards model.

Class `ProportionalHazards` computes parameter estimates and other statistics in Proportional Hazards Generalized Linear Models. These models were first proposed by Cox (1972). Two methods for handling ties are allowed. Time-dependent covariates are not allowed. The user is referred to Cox and Oakes (1984), Kalbfleisch and Prentice (1980), Elandt-Johnson and Johnson (1980), Lee (1980), or Lawless (1982), among other texts, for a thorough discussion of the Cox proportional hazards model.

Let $\lambda(t, z_i)$ represent the hazard rate at time t for observation number i with covariables contained as elements of row vector z_i. The basic assumption in the proportional hazards model (the proportionality assumption) is that the hazard rate can be written as a product of a time varying function $\lambda_0(t)$, which depends only on time, and a function $f(z_i)$, which depends only on the covariable values. The function $f(z_i)$ used in `ProportionalHazards` is given as $f(z_i) = \exp(w_i + \beta z_i)$ where w_i is a fixed constant assigned to the observation, and b is a vector of coefficients to be estimated. With this function one obtains a hazard rate $\lambda(t, z_i) = \lambda_0(t)\exp(w_i + \beta z_i)$. The form of $\lambda_0(t)$ is not important in proportional hazards models.

The constants w_i may be known theoretically. For example, the hazard rate may be proportional to a known length or area, and the w_i can then be determined from this known length or area. Alternatively, the w_i may be used to fix a subset of the coefficients β (say, β_1) at specified values. When w_i is used in this way, constants $w_i = \beta_1 z_{1i}$ are used, while the remaining coefficients in β are free to vary in the optimization algorithm. Constants are defined as 0.0 by default. If user-specified constants are desired,

use the setConstantColumn method to specify which column contains the constant.

With this definition of $\lambda(t, z_i)$, the usual partial (or marginal, see Kalbfleisch and Prentice (1980)) likelihood becomes

$$L = \prod_{i=1}^{n_d} \frac{\exp(w_i + \beta z_i)}{\sum_{j \in R(t_i)} \exp(w_j + \beta z_j)}$$

where $R(t_i)$ denotes the set of indices of observations that have not yet failed at time t_i (the risk set), t_i denotes the time of failure for the i-th observation, n_d is the total number of observations that fail. Right-censored observations (i.e., observations that are known to have survived to time t_i, but for which no time of failure is known) are incorporated into the likelihood through the risk set $R(t_i)$. Such observations never appear in the numerator of the likelihood. When setTieOptions is set to BRESLOWS_APPROXIMATE (the default), all observations that are censored at time t_i are not included in $R(t_i)$, while all observations that fail at time t_i are included in $R(t_i)$.

If it can be assumed that the dependence of the hazard rate upon the covariate values remains the same from stratum to stratum, while the time-dependent term, $\lambda_0(t)$, may be different in different strata, then ProportionalHazards allows the incorporation of strata into the likelihood as follows. Let k index the m strata (set with setStratumColumn). Then, the likelihood is given by

$$L_S = \prod_{k=1}^{m} \left[\prod_{i=1}^{n_k} \frac{\exp(w_{ki} + \beta z_{ki})}{\sum_{j \in R(t_{ki})} \exp(w_{kj} + \beta z_{kj})} \right]$$

In ProportionalHazards, the log of the likelihood is maximized with respect to the coefficients β. A quasi-Newton algorithm approximating the Hessian via the matrix of sums of squares and cross products of the first partial derivatives is used in the initial iterations. When the change in the log-likelihood from one iteration to the next is less than 100 times the convergence tolerance, Newton-Raphson iteration is used. If, during any iteration, the initial step does not lead to an increase in the log-likelihood, then step halving is employed to find a step that will increase the log-likelihood.

Once the maximum likelihood estimates have been computed, the algorithm computes estimates of a probability associated with each failure. Within stratum k, an estimate of the probability that the i-th observation fails at time t_i given the risk set $R(t_{ki})$ is given by

$$p_{ki} = \frac{\exp(w_{ki} + \beta z_{ki})}{\sum_{j \in R(t_{ki})} \exp(w_{kj} + \beta z_{kj})}$$

A diagnostic "influence" or "leverage" statistic is computed for each noncensored observation as:

$$l_{ki} = -g'_{ki} H_s^{-1} g'_{ki}$$

where H_s is the matrix of second partial derivatives of the log-likelihood, and

$$g'_{ki}$$

is computed as:

$$g'_{ki} = z_{ki} - \frac{z_{ki}\exp(w_{ki} + \beta z_{ki})}{\sum_{j \in R(t_{ki})} \exp(w_{kj} + \beta z_{kj})}$$

Influence statistics are not computed for censored observations.

A "residual" is computed for each of the input observations according to methods given in Cox and Oakes (1984, page 108). Residuals are computed as

$$r_{ki} = \exp(w_{ki} + \hat{\beta} z_{ki}) \sum_{j \in R(t_{ki})} \frac{d_{kj}}{\sum_{l \in R(t_{kj})} \exp(w_{kl} + \hat{\beta} z_{kl})}$$

where d_{kj} is the number of tied failures in group k at time t_{kj}. Assuming that the proportional hazards assumption holds, the residuals should approximate a random sample (with censoring) from the unit exponential distribution. By subtracting the expected values, centered residuals can be obtained. (The j-th expected order statistic from the unit exponential with censoring is given as

$$e_j = \sum_{l \le j} \frac{1}{h - l + 1}$$

where h is the sample size, and censored observations are not included in the summation.)

An estimate of the cumulative baseline hazard within group k is given as

$$\hat{H}_{k0}(t_{ik}) = \sum_{t_{kj} \le t_{ki}} \frac{d_{kj}}{\sum_{l \in R(t_{kj})} \exp(w_{kl} + \hat{\beta} z_{kl})}$$

The observation proportionality constant is computed as

$$\exp(w_{ki} + \hat{\beta} z_{ki})$$

Note that the user can use the JDK JAVA Logging API to generate intermediate output for the solver. 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	Logging is enabled, but observational statistics are not printed.
FINER	All output statistics are printed.
FINEST	Tracks progress through internal methods.

Fields

BRESLOWS_APPROXIMATE

```
static final public int BRESLOWS_APPROXIMATE
```

Breslows approximate method of handling ties. See `setTiesOption`.

SORTED_AS_PER_OBSERVATIONS

```
static final public int SORTED_AS_PER_OBSERVATIONS
```

Failures are assumed to occur in the same order as the observations input in `x`. The observations in `x` must be sorted from largest to smallest failure time within each stratum, and grouped by stratum. All observations are treated as if their failure/censoring times were distinct when computing the log-likelihood. See `setTiesOption`.

Constructor

ProportionalHazards

```
public ProportionalHazards(double[][] x, int[] nVarEffects, int[] indEffects)
```

Description

Constructor for `ProportionalHazards`.

Parameters

`x` – a `double` matrix containing the data, including optional data.

`nVarEffects` – an `int` array containing the number of variables associated with each effect in the model.

`indEffects` – an `int` array containing the column numbers of `x` associated with each effect. The first `nVarEffects[0]` elements of `indEffects` contain the column numbers of `x` for the variables in the first effect. The next `nVarEffects[1]` elements of `indEffects` contain the column numbers of `x` for the variables in the second effect, etc.

Methods

getCaseStatistics

```
public double[][] getCaseStatistics() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the case statistics for each observation.

There is one row for each observation, and the columns of the returned matrix contain the following:

Column	Statistic
0	Estimated survival probability at the observation time.
1	Estimated observation influence or leverage.
2	A residual estimate.
3	Estimated cumulative baseline hazard rate.
4	Observation proportionality constant.

Returns

a `double` matrix containing the case statistics.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getCensorColumn

```
public int getCensorColumn()
```

Description

Returns the column index of `x` containing the optional censoring code for each observation.

Returns

an `int` specifying the column index of `x` containing the optional censoring code for each observation.

getClassValueCounts

```
public int[] getClassValueCounts() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the number of values taken by each classification variable. The i-th element of the returned array is the number of distinct values taken by the i-th classification variable.

Returns

an `int` array containing the number of values taken by each classification variable.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getClassValues

```
public double[] getClassValues() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the class values taken by each classification variable. For description purposes, let *nclval* = `getClassValueCounts()`. Then the first *nclval[0]* elements contain the values for the first classification variable, the next *nclval[1]* elements contain the values for the second classification variable, etc.

Returns

a `double` array containing the values taken by each classification variable.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

---

getConstantColumn

```
public int getConstantColumn()
```

Description

Returns the column index of `x` containing the constant to be added to the linear response.

Returns

an `int` specifying the column index of `x` containing the constant to be added to the linear response.

---

getConvergenceTol

```
public double getConvergenceTol()
```

Description

Returns the convergence tolerance used.

Returns

a `double` specifying the convergence tolerance used.

---

getFrequencyColumn

```
public int getFrequencyColumn()
```

Description

Returns the column index of `x` containing the frequency of response for each observation.

Returns

an `int` specifying the column index of `x` containing the frequency of response for each observation.

---

getGradient

```
public double[] getGradient() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the inverse of the Hessian times the gradient vector, computed at the initial estimates.

Note that the `setHessianOption` method must be invoked with `wantHessian` set to `true` and the `setInitialEstimates` method must be invoked prior to invoking this method. Otherwise, the method throws an `IllegalStateException` exception.

Returns

a `double` array containing the inverse of the Hessian times the gradient vector, computed at the initial estimates.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getHessian

```
public double[][] getHessian() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the inverse of the Hessian of the negative of the log-likelihood, computed at the initial estimates.

Note that the `setHessianOption` method must be invoked with `wantHessian` set to `true` and the `setInitialEstimates` method must be invoked prior to invoking this method.Otherwise, the method throws an `IllegalStateException` exception.

Returns

a `double` matrix containing the inverse of the Hessian of the negative of the log-likelihood, computed at the initial estimates.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getHessianOption

```
public boolean getHessianOption()
```

Description

Returns the `boolean` used to indicate whether or not to compute the Hessian and gradient at the initial estimates.

Returns

a `boolean` specifying whether or not the Hessian and gradient are to be computed at the initial estimates. A return value equal to `true` indicates that the Hessian and gradient are to be computed.

getInitialEstimates

```
public double[] getInitialEstimates() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Gets the initial parameter estimates.

Returns

a `double` array containing the initial parameter estimates.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getLastUpdates

```
public double[] getLastUpdates() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Gets the last parameter updates.

Returns

a `double` array containing the last parameter updates (excluding step halvings).

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getLogger

```
public Logger getLogger()
```

Description

Returns the logger object and enables logging.

Returns

a `java.util.logging.Logger` object, if present, or `null`.

getMaxClass

```
public int getMaxClass()
```

Description

Returns the upper bound used on the sum of the number of distinct values found among the classification variables in `x`.

Returns

an `int` representing the upper bound used on the sum of the number of distinct values found among the classification variables in `x`.

getMaxIterations

```
public int getMaxIterations()
```

Description

Return the maximum number of iterations allowed.

Returns

an `int` specifying the maximum number of iterations allowed.

getMaximumLikelihood

```
public double getMaximumLikelihood() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the maximized log-likelihood.

The log-likelihood is fully described in the `ProportionalHazards` class description.

Returns

a `double` representing the maximized log-likelihood

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getMeans

```
public double[] getMeans() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the means of the design variables.

Returns

a `double` array containing the means of the design variables.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getNumberOfCoefficients

```
public int getNumberOfCoefficients() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the number of estimated coefficients in the model.

Returns

an `int` scalar representing the number of estimated coefficients in the model.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getNumberRowsMissing

```
public int getNumberRowsMissing() throws
ProportionalHazards.ClassificationVariableLimitException
```

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`.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getParameterStatistics

```
public double[][] getParameterStatistics() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the parameter estimates and associated statistics.

There is one row for each coefficient, and the columns of the returned matrix contain the following:

Column	Statistic
0	The coefficient estimate, $\hat{\beta}$
1	Estimated standard deviation of the estimated coefficient
2	Asymptotic normal score for testing that the coefficient is zero against the two-sided alternative
3	*p*-value associated with the normal score in column 2

Returns

a `double` matrix containing the parameter estimates and associated statistics.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getResponseColumn

```
public int getResponseColumn()
```

Description

Returns the column index of `x` containing the response time for each observation.

Returns

an `int` specifying the column index of `x` containing the response time for each observation.

getStratumColumn

```
public int getStratumColumn()
```

Description

Returns the column index of `x` containing the stratum number for each observation.

Returns

an `int` specifying the column index of `x` containing the stratum number for each observation.

getStratumNumbers

```
public int[] getStratumNumbers() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the stratum number used for each observation. If `stratumRatio` is not -1.0, additional "strata" (other than those specified by column `groupIndex` of `x` set via the `setStratumColumn` method) may be generated. The array also contains a record of the generated strata. See the `ProportionalHazards` class description for more detail.

Returns

an `int` array containing the stratum number used for each observation.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

getStratumRatio

```
public double getStratumRatio()
```

Description

Returns the ratio at which a stratum is split into two strata.

Returns

a `double` specifying the ratio at which a stratum is split into two strata.

getTiesOption

```
public int getTiesOption()
```

Description

Returns the method used for handling ties.

Returns

an `int` specifying the method to be used in handling ties as indicated by the value in the following table:

Value	**Method**
BRESLOWS_APPROXIMATE	Breslow's approximate method. This is the default.
SORTED_AS_PER_OBSERVATIONS	Failures are assumed to occur in the same order as the observations input in `x`. The observations in `x` must be sorted from largest to smallest failure time within each stratum, and grouped by stratum. All observations are treated as if their failure/censoring times were distinct when computing the log-likelihood.

getVarianceCovarianceMatrix

```
public double[][] getVarianceCovarianceMatrix() throws
ProportionalHazards.ClassificationVariableLimitException
```

Description

Returns the estimated asymptotic variance-covariance matrix of the parameters.

Returns

a `double` matrix containing the estimated asymptotic variance-covariance matrix of the parameters.

Exception

`ClassificationVariableLimitException` is thrown if the classification variable limit set by the user through `setUpperBound` has been exceeded.

setCensorColumn

```
public void setCensorColumn(int censorIndex)
```

Description

Sets the column index of `x` containing the optional censoring code for each observation.

If `x[i][censorIndex]` equals 0, the failure time `x[i][responseIndex]` is treated as an exact time of failure. Otherwise, it is treated as right-censored time. By default, it is assumed that there is no censor code column in `x` and all observations are assumed to be exact failure times.

Parameter

`censorIndex` – an `int` specifying the column index of `x` containing the optional censoring code for each observation.

setClassVarColumns

```
public void setClassVarColumns(int[] classVarIndices)
```

Description

Sets the column indices of `x` that are the classification variables.

Parameter

`classVarIndices` – an `int` array containing the column numbers of `x` that are the classification variables. By default it is assumed there are no classification variables.

setConstantColumn

```
public void setConstantColumn(int fixedIndex)
```

Description

Sets the column index of `x` containing the constant w_i to be added to the linear response.

The linear response is taken to be $w_i + z_i\hat{\beta}$ where w_i is the observation constant, z_i is the observation design row vector, and $\hat{\beta}$ is the vector of estimated parameters. The "fixed" constant allows one to test hypotheses about parameters via the log-likelihoods. If this method is not called, it is assumed that $w_i = 0$ for all observations.

Parameter

`fixedIndex` – an `int` specifying the column index of `x` containing the constant to be added to the linear response.

setConvergenceTol

```
public void setConvergenceTol(double convergenceTol)
```

Description

Set the convergence tolerance.

Convergence is assumed when the relative change in the maximum likelihood from one iteration to the next is less than `convergenceTol`. If `convergenceTol` is zero, `convergenceTol` = 0.0001 is assumed. The default value is 0.0001.

Parameter

`convergenceTol` – a `double` specifying the convergence tolerance.

setFrequencyColumn

```
public void setFrequencyColumn(int frequencyIndex)
```

Description

Sets the column index of `x` containing the frequency of response for each observation.

By default it is assumed that there is no frequency response column recorded in `x`. Each observation in the data array is assumed to be for a single failure; that is, the frequency of response for each observation is 1.

Parameter

`frequencyIndex` – an `int` specifying the column index of `x` containing the frequency of response for each observation.

setHessianOption

```
public void setHessianOption(boolean wantHessian)
```

Description

Set the option to have the Hessian and gradient be computed at the initial estimates.

Parameter

`wantHessian` – a `boolean` specifying whether or not the Hessian and gradient are to be computed at the initial estimates. If this option is set to `true` the user must set the initial estimates via the `setInitialEstimates` method. By default the Hessian and gradient are not computed at the initial estimates.

setInitialEstimates

```
public void setInitialEstimates(double[] initialCoef)
```

Description

Sets the initial parameter estimates.

Care should be taken to ensure that the supplied estimates for the model coefficients β correspond to the generated covariate vector z_{ki}.

Parameter

`initialCoef` – a `double` array containing the initial parameter estimates. By default the initial parameter estimates are all 0.0.

setMaxClass

```
public void setMaxClass(int maxClass)
```

Description

Sets an upper bound on the sum of the number of distinct values found among the classification variables in `x`.

For example, if the model consisted of of two class variables, one with the values $\{1, 2, 3, 4\}$ and a second with the values $\{0, 1\}$, then the total number of different classification values is 4 + 2 = 6, and `maxClass` $>=$ 6. The default value is the number of observations in `x`.

Parameter

`maxClass` – an `int` representing an upper bound on the sum of the number of distinct values found among the classification variables in `x`.

setMaxIterations

```
public void setMaxIterations(int maxIterations)
```

Description

Set the maximum number of iterations allowed.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations allowed. The default value is 30.

setResponseColumn

```
public void setResponseColumn(int responseIndex)
```

Description

Sets the column index of `x` containing the response variable.

For point observations, `x[i][responseIndex]` contains the time of the *i*-th event. For right-censored observations, `x[i][responseIndex]` contains the right-censoring time. Note that because `ProportionalHazards` only uses the order of the events, negative “times” are allowed. By default `responseIndex = 0`.

Parameter

`responseIndex` – an `int` specifying the column index of `x` containing the response variable.

setStratumColumn

```
public void setStratumColumn(int stratumIndex)
```

Description

Sets the column index of x containing the stratification variable.

Column `stratumIndex` of `x` contains a unique value for each stratum in the data. The risk set for an observation is determined by its stratum. By default it is assumed that all obvservations are from one statum.

Parameter

`stratumIndex` – an `int` specifying the column index of `x` containing the stratification variable.

setStratumRatio

```
public void setStratumRatio(double stratumRatio)
```

Description

Set the ratio at which a stratum is split into two strata.

Let

$$r_k = \exp(z_k \hat{\beta} + w_k)$$

be the observation proportionality constant, where z_k is the design row vector for the k-th observation and w_k is the optional fixed parameter specified by $x_{k,\text{fixedIndex}}$. Let r_{min} be the minimum value r_k in a stratum, where, for failed observations, the minimum is over all times less than or equal to the time of occurrence of the k-th observation. Let r_{max} be the maximum value of r_k for the remaining observations in the group. Then, if $r_{min} >$ `stratumRatio` $* r_{max}$, the observations in the group are divided into two groups at k. The default value of `stratumRatio` = 1000 is usually good.

Set `stratumRatio` to any negative value if no division into strata is to be made.

Parameter

`stratumRatio` – a `double` specifying the ratio at which a stratum is split into two strata.

setTiesOption

```
public void setTiesOption(int iTie)
```

Description

Sets the method for handling ties.

Parameter

`iTie` – an `int` specifying the method to be used in handling ties. It can be either `BRESLOWS_APPROXIMATE` or `SORTED_AS_PER_OBSERVATIONS`.

iTie	**Method**
`BRESLOWS_APPROXIMATE`	Breslow's approximate method. This is the default.
`SORTED_AS_PER_OBSERVATIONS`	Failures are assumed to occur in the same order as the observations input in `x`. The observations in `x` must be sorted from largest to smallest failure time within each stratum, and grouped by stratum. All observations are treated as if their failure/censoring times were distinct when computing the log-likelihood.

By default, iTie is BRESLOWS_APPROXIMATE.

Example: ProportionalHazards

The following example is taken from Lawless (1982, page 287) and involves the survival of lung cancer patients based upon their initial tumor types and treatment type. In the example, the likelihood is maximized with no strata present in the data. This corresponds to Example 7.2.3 in Lawless (1982, page 367). The model is given as:

$$\ln(\lambda) = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \alpha_i + \gamma_j$$

where α_i and γ_j correspond to dummy variables generated from classification variables in columns 5 and 6 of x. Respectively, x_1 corresponds to column index 2, x_2 corresponds to column index 3, and x_3 corresponds to column index 4 of x. Column 0 of x contains the response and column 1 of x contains the censoring code. Logging is used to print output statistics.

```
import com.imsl.stat.*;
import com.imsl.math.PrintMatrix;
import java.util.logging.*;

public class ProportionalHazardsEx1 {

    public static void main(String args[]) throws Exception {
        double[][] x = {
            {411, 0, 7, 64, 5, 1, 0},
            {126, 0, 6, 63, 9, 1, 0},
            {118, 0, 7, 65, 11, 1, 0},
            {92, 0, 4, 69, 10, 1, 0},
            {8, 0, 4, 63, 58, 1, 0},
            {25, 1, 7, 48, 9, 1, 0},
            {11, 0, 7, 48, 11, 1, 0},
            {54, 0, 8, 63, 4, 2, 0},
            {153, 0, 6, 63, 14, 2, 0},
            {16, 0, 3, 53, 4, 2, 0},
            {56, 0, 8, 43, 12, 2, 0},
            {21, 0, 4, 55, 2, 2, 0},
            {287, 0, 6, 66, 25, 2, 0},
            {10, 0, 4, 67, 23, 2, 0},
            {8, 0, 2, 61, 19, 3, 0},
            {12, 0, 5, 63, 4, 3, 0},
            {177, 0, 5, 66, 16, 4, 0},
            {12, 0, 4, 68, 12, 4, 0},
            {200, 0, 8, 41, 12, 4, 0},
            {250, 0, 7, 53, 8, 4, 0},
            {100, 0, 6, 37, 13, 4, 0},
            {999, 0, 9, 54, 12, 1, 1},
            {231, 1, 5, 52, 8, 1, 1},
            {991, 0, 7, 50, 7, 1, 1},
            {1, 0, 2, 65, 21, 1, 1},
            {201, 0, 8, 52, 28, 1, 1},
            {44, 0, 6, 70, 13, 1, 1},
            {15, 0, 5, 40, 13, 1, 1},
```

```
            {103, 1, 7, 36, 22, 2, 1},
            {2, 0, 4, 44, 36, 2, 1},
            {20, 0, 3, 54, 9, 2, 1},
            {51, 0, 3, 59, 87, 2, 1},
            {18, 0, 4, 69, 5, 3, 1},
            {90, 0, 6, 50, 22, 3, 1},
            {84, 0, 8, 62, 4, 3, 1},
            {164, 0, 7, 68, 15, 4, 1},
            {19, 0, 3, 39, 4, 4, 1},
            {43, 0, 6, 49, 11, 4, 1},
            {340, 0, 8, 64, 10, 4, 1},
            {231, 0, 7, 67, 18, 4, 1}
        };
        int indef[] = {2, 3, 4, 5, 6};
        int nvef[] = {1, 1, 1, 1, 1};
        int indcl[] = {5, 6};
        int maxcl = 6, icen = 1;
        double ratio = 10000.0;

        ProportionalHazards ph = new ProportionalHazards(x, nvef, indef);
        ph.setMaxClass(maxcl);
        ph.setCensorColumn(icen);
        ph.setClassVarColumns(indcl);
        ph.setStratumRatio(ratio);

        // Level.FINER prints most output statistics
        Logger logger = ph.getLogger();
        ConsoleHandler ch = new ConsoleHandler();

        // Set ConsoleHandler Level to ALL
        ch.setLevel(Level.ALL);
        logger.setLevel(Level.FINER);
        logger.addHandler(ch);
        ch.setFormatter(new com.imsl.IMSLFormatter());

        double coef[][] = ph.getParameterStatistics();
        new PrintMatrix("\nFinal Coefficient Matrix").print(coef);
    }
}
```

Output

```
Initial Estimates
   0
0  0
1  0
2  0
3  0
4  0
5  0
6  0


Method  Iteration  Step       Maximum scaled     Log
                   Size        coef. update      likelihood
```

```
  Q-N        0                            -102.40056551567265
  Q-N        1      1.0  0.503384047154466 -91.04395077806367
  Q-N        2      1.0  0.5781995573157528 -88.06808499091763
  N-R        3      1.0  0.11310452893592852 -87.92233446871525
  N-R        4      1.0  0.06957951110074143 -87.88778754387496
  N-R        5      1.0  8.149048655013355E-4 -87.88778009939108
Log-Likelihood = -87.888
                      Coefficient Statistics
   Coefficient  Std. error  Asymptotic z-stat  Asymptotic p-value
0    -0.5846      0.1368        -4.2721            0
1    -0.0131      0.0206        -0.6342            0.526
2     0.0008      0.0118         0.0645            0.9486
3    -0.367       0.4848        -0.7572            0.449
4    -0.0077      0.5068        -0.0152            0.9878
5     1.1129      0.6331         1.758             0.0787
6     0.3797      0.4058         0.9357            0.3494


                Asymptotic Coefficient Covariance
     0        1        2        3        4        5        6
0  0.0187   0.0003   0.0003   0.0057   0.0097   0.0043   0.0021
1           0.0004  -0       -0.0017  -0.0008  -0.0031  -0.0029
2                    0.0001   0.0008  -0.0018   0.0006   0.0017
3                             0.235    0.098    0.1184   0.0373
4                                      0.2568   0.1253  -0.0194
5                                               0.4008   0.0629
6                                                        0.1647


                             Case Analysis
    Survival Prob.  Influence  Residual  Cumulative hazard  Prop. constant
 0      0.0022       0.0414    2.0531         6.1032            0.3364
 1      0.2988       0.1088    0.7409         1.2078            0.6134
 2      0.3424       0.1184    0.3576         1.0719            0.3336
 3      0.4336       0.1554    1.5272         0.8357            1.8274
 4      0.9555       0.5567    0.0933         0.0455            2.0499
 5      0.7365       ?         0.1272         0.3058            0.4158
 6      0.9204       0.3729    0.0346         0.083             0.4164
 7      0.5876       0.2637    0.1446         0.5317            0.2719
 8      0.2577       0.1173    1.196          1.3561            0.882
 9      0.8457       0.1486    0.9656         0.1676            5.7608
10      0.5481       0.3133    0.2135         0.6012            0.3551
11      0.7365       0.2108    0.9551         0.3058            3.1232
12      0.0293       0.0602    3.018          3.5289            0.8552
13      0.9382       0.0935    0.173          0.0638            2.7135
14      0.9555       0.1595    1.3142         0.0455           28.8855
15      0.8854       0.2322    0.5864         0.1217            4.8164
16      0.1814       0.0918    2.6217         1.707             1.5358
17      0.8854       0.1869    0.3258         0.1217            2.6765
18      0.1414       0.2303    0.7187         1.9565            0.3673
19      0.0522       0.0943    1.6591         2.9529            0.5618
20      0.3899       0.2212    1.1745         0.9419            1.2469
21      0            0         1.7281        21.1049            0.0819
22      0.0806       ?         2.1865         2.5177            0.8684
23      0.0001       0.0049    2.4603         8.8921            0.2767
24      0.9892       0.3072    0.0462         0.0108            4.2758
```

```
25      0.1074       0.1724     0.3406         2.2311            0.1527
26      0.664        0.2513     0.1573         0.4095            0.3841
27      0.8655       0.2215     0.1472         0.1444            1.0196
28      0.3899       ?          0.4533         0.9419            0.4812
29      0.9781       0.2495     0.0561         0.0222            2.531
30      0.769        0.2556     1.0257         0.2627            3.9045
31      0.6291       0.3509     1.7991         0.4635            3.8817
32      0.8233       0.2598     1.0635         0.1944            5.4705
33      0.4739       0.26       1.6474         0.7468            2.2058
34      0.5104       0.3191     0.3886         0.6725            0.5779
35      0.2173       0.183      0.485          1.5267            0.3177
36      0.7979       0.2642     1.0764         0.2258            4.7675
37      0.7          0.16       0.2598         0.3567            0.7282
38      0.0094       0.2267     0.8668         4.6642            0.1858
39      0.0806       0.2045     0.8122         2.5177            0.3226

Last Coefficient Update

0  -5.835E-8
1   1.401E-9
2  -8.597E-9
3  -2.822E-7
4  -4.566E-8
5   1.256E-7
6   1.058E-8


Covariate Means

0   5.65
1  56.575
2  15.65
3   0.35
4   0.275
5   0.125
6   0.525


Distinct Values For Each Class Variable
Variable 0: 1 2 3 4
Variable 1: 0 1
Stratum Numbers For Each Observation

 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

Number of Missing Values = 0

Final Coefficient Matrix
     0       1      2       3
0  -0.585  0.137  -4.272  0
1  -0.013  0.021  -0.634  0.526
2   0.001  0.012   0.064  0.949
3  -0.367  0.485  -0.757  0.449
4  -0.008  0.507  -0.015  0.988
5   1.113  0.633   1.758  0.079
6   0.38   0.406   0.936  0.349
```

ProportionalHazards.ClassificationVariableLimitException class

```
static public class
com.imsl.stat.ProportionalHazards.ClassificationVariableLimitException extends
```

`com.imsl.IMSLException`

The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

Constructors

ProportionalHazards.ClassificationVariableLimitException

`public ProportionalHazards.ClassificationVariableLimitException(String message)`

Description

Constructs a `ClassificationVariableLimitException`.

Parameter

`message` – a `String` containing the error message

ProportionalHazards.ClassificationVariableLimitException

`public ProportionalHazards.ClassificationVariableLimitException(String key, Object[] arguments)`

Description

The Classification Variable limit set by the user through `setUpperBound` has been exceeded.

Parameters

`key` – the `String` key of the error message in the resource bundle

`arguments` – an array containing arguments used within the error message string

LifeTables class

`public class com.imsl.stat.LifeTables implements Serializable, Cloneable`

Computes population (current) or cohort life tables based upon the observed population sizes at the middle (for population table) or the beginning (for cohort table) of some user specified age intervals. The number of deaths in each of these intervals must also be observed.

The probability of dying prior to the middle of the interval, given that death occurs somewhere in the interval, may also be specified. Often, however, this probability is taken to be 0.5. For a discussion of the probability models underlying the life table here, see the references.

Let t_i, for i = 0, 1, ..., t_n denote the time grid defining the n age intervals, and note that the length of the age intervals may vary. Following Gross and Clark (1975, page 24), let d_i denote the number of individuals dying in age interval i, where age interval i ends at time t_i. For population table, the death rate at the middle of the interval is given by $r_i = d_i/(M_i h_i)$, where M_i is the number of individuals alive at the middle of the interval, and $h_i = t_i - t_{i-1}$, $t_0 = 0$. The number of individuals alive at the beginning

of the interval may be estimated by $P_i = M_i + (1 - a_i)d_i$ where a_i is the probability that an individual dying in the interval dies prior to the interval midpoint. For cohort table, P_i is input directly while the death rate in the interval, r_i, is not needed.

The probability that an individual dies during the age interval from t_{i-1} to t_i is given by $q_i = d_i/P_i$. It is assumed that all individuals alive at the beginning of the last interval die during the last interval. Thus, q_n = 1.0. The asymptotic variance of q_i can be estimated by

$$\sigma_i^2 = q_i(1 - q_i)/P_i$$

For a population table, the number of individuals alive in the middle of the time interval (input in `nCohort[i]`) must be adjusted to correspond to the number of deaths observed in the interval. The algorithm assumes that the number of deaths observed in interval h_i occur over a time period equal to h_i. If d_i is measured over a period u_i, where $u_i \neq d_i$, then `nCohort[i]` must be adjusted to correspond to d_i by multiplication by u_i/h_i, i.e., the value M_i input as `nCohort[i]` is computed as

$$M_i^* = M_i u_i/h_i$$

Let S_i denote the number of survivors at time t_i from a hypothetical (for population table) or observed (for cohort table) population. Then, S_0 =`initialPopulation` for population table, and S_0 = `nCohort[0]` for cohort table, and S_i is given by $S_i = S_{i-1} - \delta_{i-1}$ where $\delta_i = S_i q_i$ is the number of individuals who die in the i th interval. The proportion of survivors in the interval is given by $V_i = S_i/S_0$ while the asymptotic variance of V_i can be estimated as follows:

$$\text{var}(V_i) = V_i^2 \sum_{j=1}^{i-1} \frac{\sigma_j^2}{(1 - q_j)^2}$$

The expected lifetime at the beginning of the interval is calculated as the total lifetime remaining for all survivors alive at the beginning of the interval divided by the number of survivors at the beginning of the interval. If e_i denotes this average expected lifetime, then the variance of e_i can be estimated as (see Chiang 1968)

$$\text{var}(e_i) = \frac{\sum_{j=i}^{n-1} P_j^2 \sigma_j^2 \left[e_{j+1} + h_{j+1}(1 - a_j)\right]^2}{P_j^2}$$

where $\text{var}(e_n) = 0.0$.

Finally, the total number of time units lived by all survivors in the time interval can be estimated as:

$$U_i = h_i[S_i - \delta_i(1 - a_i)]$$

Constructor

LifeTables

```
public LifeTables(int[] nCohort, double[] age, double[] a)
```

Description

Constructs a new `LifeTables` instance. The number of classes, *nClasses* is equal to the length of the input array `nCohort` .

Parameters

`nCohort` – an `int` array of length *nClasses* containing the cohort sizes during each interval. If the Population Table option is used, then `nCohort[i]` contains the size of the population at the midpoint of interval `i`. Otherwise, `nCohort[i]` contains the size of the cohort at the beginning of interval `i`. When requesting a population table, the population sizes in `nCohort` may need to be adjusted to correspond to the number of deaths in `nDeaths`. See the class description section for more information.

`age` – a `double` array of length *nClasses* + 1 containing the lowest age in each age interval, and in `age[nClasses]`, the endpoint of the last age interval. Negative `age[0]` indicates that the age intervals are all of length |`age[0]`| and that the initial age interval is from 0.0 to |`age[0]`|. In this case, all other elements of age need not be specified. `age[nClasses]` need not be specified when getting a cohort table.

`a` – a `double` array of length *nClasses* containing the fraction of those dying within each interval who die before the interval midpoint. A common choice for all `a[i]` is 0.5. This choice may also be specified by setting `a[0]` to any negative value. In this case, the remaining values of `a` need not be specified.

Methods

getLifeTable

```
public double[][] getLifeTable()
```

Description

Compute a cohort table.

Returns

a `double` matrix of dimensions *nClasses* by 12 containing the population table. Entries in the *i*th row are for the age interval defined by `age[i]`. Column definitions are described in the following table.

Column	*Description*
0	Lowest age in the age interval.
1	Fraction of those dying within the interval who die before the interval midpoint.
2	Number surviving to the beginning of the interval.
3	Number of deaths in the interval.
4	Death rate in the interval. For cohort table, this column is set to NaN (not a number).
5	Proportion dying in the interval.
6	Standard error of the proportion dying in the interval.
7	Proportion of survivors at the beginning of the interval.
8	Standard error of the proportion of survivors at the beginning of the interval.
9	Expected lifetime at the beginning of the interval.
10	Standard error of the expected life at the beginning of the interval.
11	Total number of time units lived by all of the population in the interval.

getPopulationTable

`public double[][] getPopulationTable(int[] nDeaths)`

Description

Compute a population table.

Parameter

`nDeaths` – an `int` array of *nClasses* containing the number of deaths in each age interval.

Returns

a `double` matrix of dimensions *nClasses* by 12 containing the population table. Entries in the *i*th row are for the age interval defined by `age[i]`. Column definitions are the same as in `getLifeTable`.

setPopulationSize

`public void setPopulationSize(int initialPopulation)`

Description

Sets the population size at the beginning of the first age interval in requesting a population table. A default value of 10,000 is used to allow easy entry of `nCohorts` and `nDeaths` when numbers are available as percentages.

Parameter

`initialPopulation` – an `int` scalar specifying the initial population. Default: `initialPopulation = 10000`.

Example: Life Tables

This example is taken from Chiang (1968). The cohort life table has thirteen equally spaced intervals, so age[0] is set to -5.0. Similarly, the probabilities of death prior to the middle of the interval are all taken to be 0.5, so a[0] is set to -1.0.

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

public class LifeTablesEx1 {

    public static void main(String args[]) {
        int[] nCohort = {
            270, 268, 264, 261, 254, 251,
            248, 232, 166, 130, 76, 34, 13
        };
        double[] age = new double[nCohort.length + 1];
        double[] a = new double[nCohort.length];

        age[0] = -5.0;
        a[0] = -1.0;

        LifeTables lt = new LifeTables(nCohort, age, a);
        double[][] table = lt.getLifeTable();

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        String[] cols = {
            "Age", "PDHALF", "Alive", "Deaths", "Death Rate",
            "P(D)", "Std(P(D))", "P(S)", "Std(P(S))", "Lifetime",
            "Std(Life)", "Time Units"
        };
        pmf.setColumnLabels(cols);
        pmf.setNumberFormat(new java.text.DecimalFormat("0.0####"));
        PrintMatrix pm = new PrintMatrix("Life Table");
        pm.setPageWidth(40);
        pm.print(pmf, table);
    }
}
```

Output

```
          Life Table
    Age   PDHALF  Alive  Deaths
 0   0.0   0.5    270.0    2.0
 1   5.0   0.5    268.0    4.0
 2  10.0   0.5    264.0    3.0
 3  15.0   0.5    261.0    7.0
 4  20.0   0.5    254.0    3.0
 5  25.0   0.5    251.0    3.0
 6  30.0   0.5    248.0   16.0
 7  35.0   0.5    232.0   66.0
 8  40.0   0.5    166.0   36.0
 9  45.0   0.5    130.0   54.0
10  50.0   0.5     76.0   42.0
11  55.0   0.5     34.0   21.0
12  60.0   0.5     13.0   13.0

    Death Rate   P(D)    Std(P(D))   P(S)
 0      ?       0.00741   0.00522   1.0
 1      ?       0.01493   0.00741   0.99259
 2      ?       0.01136   0.00652   0.97778
```

```
 3      ?       0.02682   0.01      0.96667
 4      ?       0.01181   0.00678   0.94074
 5      ?       0.01195   0.00686   0.92963
 6      ?       0.06452   0.0156    0.91852
 7      ?       0.28448   0.02962   0.85926
 8      ?       0.21687   0.03199   0.61481
 9      ?       0.41538   0.04322   0.48148
10      ?       0.55263   0.05704   0.28148
11      ?       0.61765   0.08334   0.12593
12      ?       1.0       0.0       0.04815

   Std(P(S))  Lifetime  Std(Life)  Time Units
 0   0.0        43.18519  0.69929     1345.0
 1   0.00522    38.48881  0.67074     1330.0
 2   0.00897    34.03409  0.62303     1312.5
 3   0.01092    29.39655  0.59401     1287.5
 4   0.01437    25.1378   0.54028     1262.5
 5   0.01557    20.40837  0.52367     1247.5
 6   0.01665    15.625    0.51485     1200.0
 7   0.02116    11.53017  0.49815      995.0
 8   0.02962    10.12048  0.46017      740.0
 9   0.03041     7.23077  0.4328       515.0
10   0.02737     5.59211  0.43607      275.0
11   0.02019     4.41176  0.41671      117.5
12   0.01303     2.5      0.0           32.5
```

Chapter 22: Probability Distribution Functions and Inverses

Types

class Cdf 1239
class Pdf 1280
class InvCdf 1296
interface CdfFunction 1307
class InverseCdf 1308
interface Distribution 1311
interface ProbabilityDistribution 1311
class NormalDistribution 1313
class GammaDistribution 1315
class LogNormalDistribution 1317
class PoissonDistribution 1319

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 $\text{Prob}(\cdot)$ 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.

Cdf class

```
public final class com.imsl.stat.Cdf
```

Cumulative probability distribution functions.

Methods

F

```
static public double F(double x, double dfn, double dfd)
```

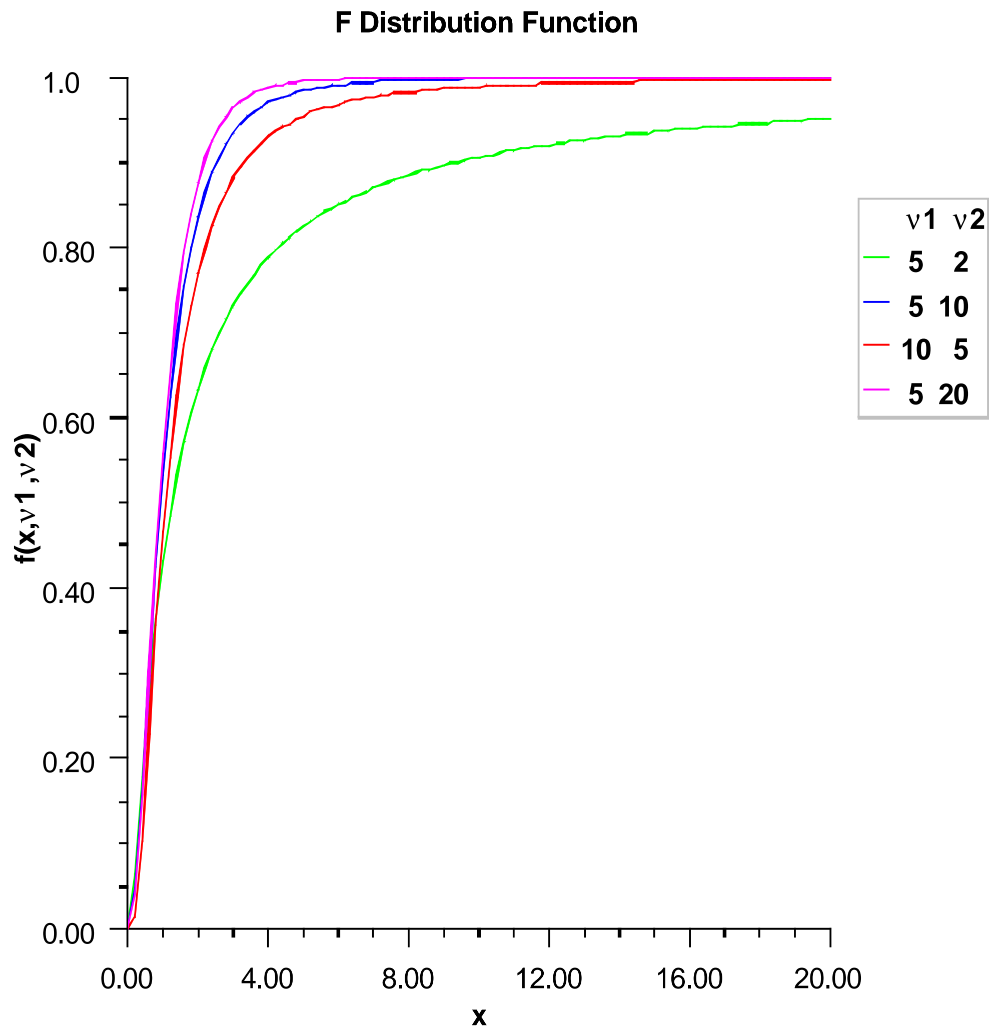
Description

Evaluates the F cumulative probability 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:

$$\mathrm{F}(X, dfn, dfd) = \mathrm{F}(1/X, dfd, dfn)$$

For greater right tail accuracy, see com.imsl.stat.Cdf.complementaryF (p. 1252) .



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`.

Pareto

```
static public double Pareto(double x, double xm, double k)
```

Description

Evaluates the Pareto cumulative probability distribution function.

Method `Pareto` evaluates the distribution function, F, of a Pareto random variable with scale parameter x_m and shape parameter k. It is given by

$$F(x, x_m, k) = 1 - \left(\frac{x_m}{x}\right)^k$$

where $x_m > 0$ and $k > 0$. The function is only defined for $x \geq x_m$

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`xm` – a `double` scalar value representing the scale parameter, x_m.

`k` – a `double` scalar value representing the shape parameter, k.

Returns

a `double` scalar value representing the probability that a Pareto random variable takes a value less than or equal to `x`.

Rayleigh

```
static public double Rayleigh(double x, double alpha)
```

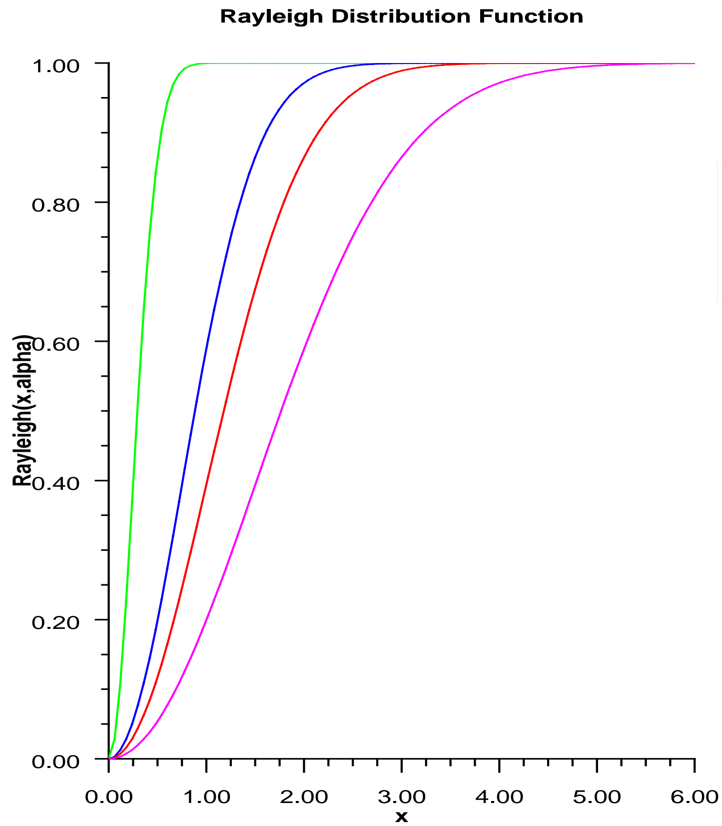
Description

Evaluates the Rayleigh cumulative probability distribution function.

Method `Rayleigh` is a special case of Weibull distribution function where the shape parameter `gamma` is 2.0; that is,

$F(x) = 1 - e^{-\frac{x^2}{2\alpha^2}}$

where `alpha` is the scale parameter.



Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated. It must be non-negative.

`alpha` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value representing the probability that a Rayleigh random variable takes a value less

than or equal to x.

Weibull

```
static public double Weibull(double x, double gamma, double alpha)
```

Description

Evaluates the Weibull cumulative probability distribution function.

Method Weibull evaluates the distribution function given by

$$F(x,\gamma,\alpha) = 1 - e^{-(x/\alpha)^{\gamma}}$$

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, γ.

alpha – a double scalar value representing the scale parameter, α.

Returns

a double scalar value representing the probability that a Weibull random variable takes a value less than or equal to x.

beta

```
static public double beta(double x, double pin, double qin)
```

Description

Evaluates the beta cumulative probability distribution function.

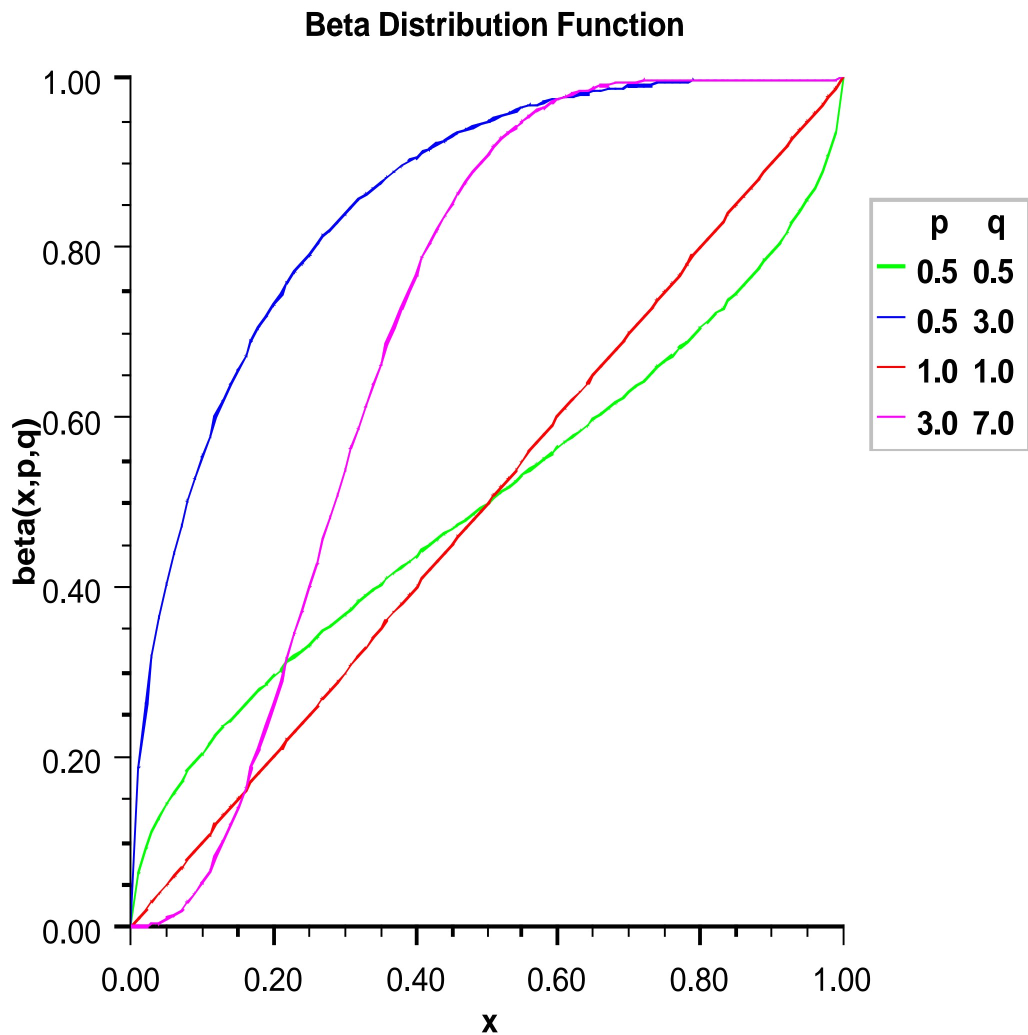
Method beta evaluates the distribution function of a beta random variable with parameters pin and qin. This function is sometimes called the *incomplete beta ratio* and, with p = pin and q = qin, is denoted by $I_x(p,q)$. It is given by

$$I_x(p,q) = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \int_0^x t^{p-1}(1-t)^{q-1}\,dt$$

where $\Gamma(\cdot)$ is the gamma function. The 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 beta function evaluated at x=1) and is denoted by $\beta_1(p,q)$.

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.

betaMean

```
static public double betaMean(double pin, double qin)
```

Description

Evaluates the mean of the beta cumulative probability distribution function

Parameters

pin – a double, the first beta distribution parameter.

qin – a double, the second beta distribution parameter.

Returns

a double, the mean of the beta distribution function.

betaVariance

```
static public double betaVariance(double pin, double qin)
```

Description

Evaluates the variance of the beta cumulative probability distribution function

Parameters

pin – a double, the first beta distribution parameter.

qin – a double, the second beta distribution parameter.

Returns

a double, the variance of the beta distribution function.

binomial

```
static public double binomial(int k, int n, double pin)
```

Description

Evaluates the binomial cumulative probability distribution function.

Method binomial evaluates the distribution function of a binomial random variable with parameters n and p with p=*pin*. 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

$$\Pr(X = j) = \frac{(n+1-j)\,p}{j\,(1-p)}\Pr(X = j-1)$$

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, ε, 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.

pin – a double scalar value representing the probability of success on each independent 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 pin probability of success.

bivariateNormal

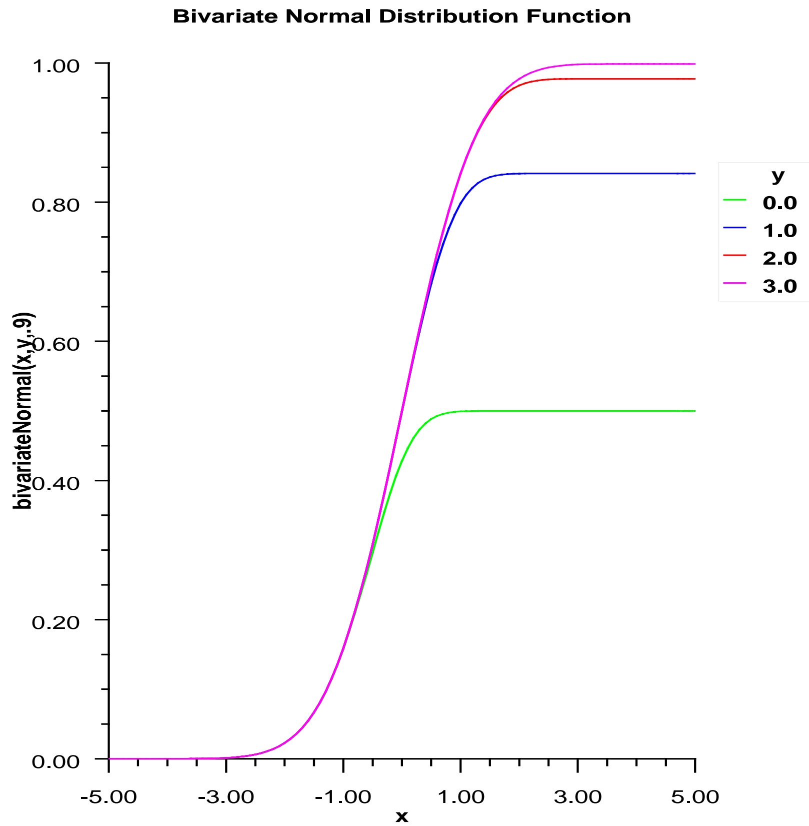
```
static public double bivariateNormal(double x, double y, double rho)
```

Description

Evaluates the bivariate normal cumulative probability distribution function. Let (X,Y) be a bivariate normal variable with mean $(0,0)$ and variance-covariance matrix

$$\begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix}$$

This method computes the probability that $X \leq$ x and $Y \leq$ y.



Parameters

`x` – is the *x*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

`y` – is the *y*-coordinate of the point for which the bivariate normal distribution function is to be evaluated.

`rho` – is the correlation coefficient.

Returns

the probability that a bivariate normal random variable (X,Y) with correlation `rho` satisfies $X \leq$ `x` and $Y \leq$ `y`.

chi

```
static public double chi(double chsq, double df)
```

Description

Evaluates the chi-squared cumulative distribution function.

Method `chi` evaluates the cumulative distribution function (CDF) F, of a chi-squared random variable with `df` degrees of freedom, that is, with x = `chsq` and ν = `df`,

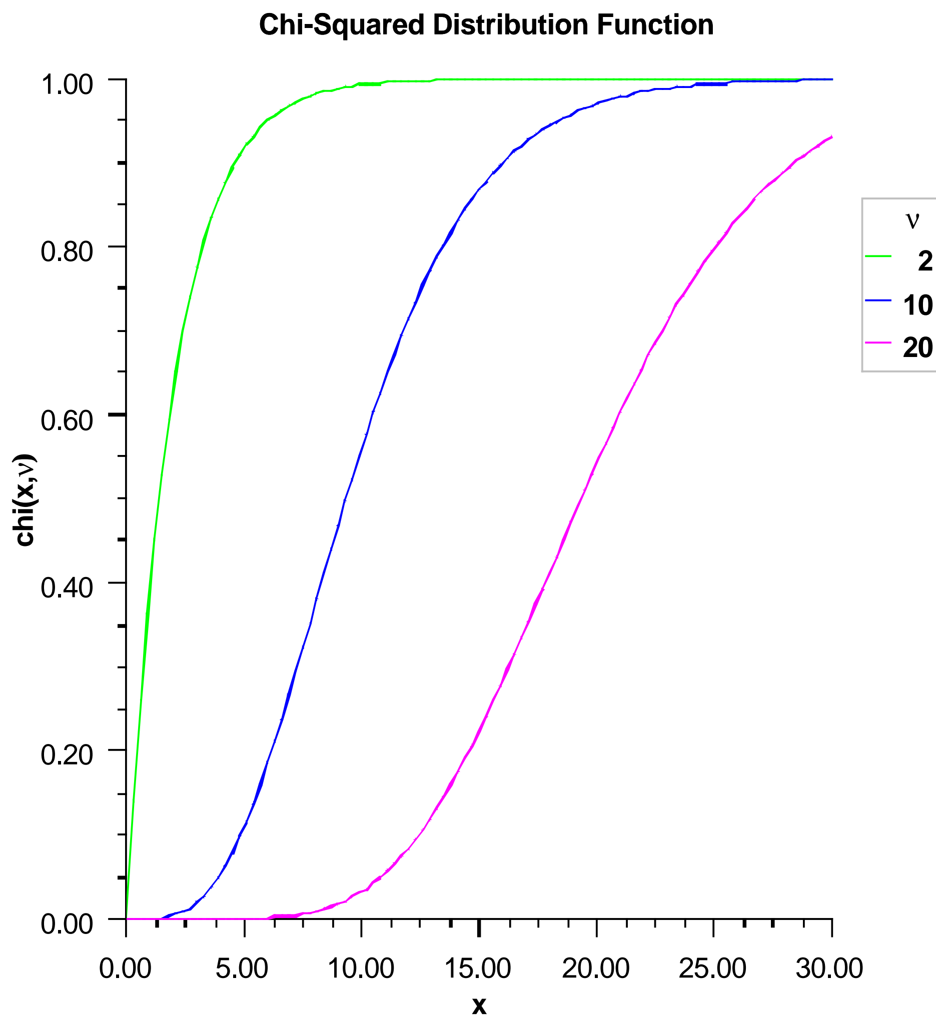
$$F(x,\nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} \int_0^x e^{-t/2} t^{\nu/2-1} dt$$

where $\Gamma(\cdot)$ is the gamma function. The value $p = F(x,\nu)$ is the probability that the random variable takes a value less than or equal to x.

For $\nu > 343$, `chi` uses the Wilson-Hilferty approximation (Abramowitz and Stegun [A&S] 1964, equation 26.4.17) for p in terms of the normal CDF, which is evaluated using method `normal`.

For $\nu \leq 343$, `chi` uses series expansions to evaluate p: for $x < \nu$, `chi` calculates p using A&S equation 6.5.29, and for $x \geq \nu$, `chi` calculates p using the continued fraction (CF) expansion of the incomplete gamma function given in A&S equation 6.5.31 and implemented using Lentz's algorithm (Lentz, W.J., 1976, Applied Optics, vol. 15, pp. 668-671) as modified by Thompson & Barnett (Thompson, I.J., and Barnett, A.R., 1986, Journal of Computational Physics, vol. 64, pp. 490-509).

For greater right tail accuracy, see `com.imsl.stat.Cdf.complementaryChi` (p. 1250) .



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. `df` must be positive.

Returns

a `double` scalar value representing the probability that a chi-squared random variable takes a value less than or equal to `chsq`.

chiMean

```
static public double chiMean(double df)
```

Description

Evaluates the mean of the chi-squared cumulative probability distribution function

Parameter

`df` – a `double` scalar value representing the number of degrees of freedom. This must be at least 0.5.

Returns

a `double`, the mean of the chi-squared distribution function.

chiVariance

```
static public double chiVariance(double df)
```

Description

Evaluates the variance of the chi-squared cumulative probability distribution function

Parameter

`df` – a `double` scalar value representing the number of degrees of freedom. This must be at least 0.5.

Returns

a `double`, the variance of the chi-squared distribution function.

complementaryChi

```
static public double complementaryChi(double chsq, double df)
```

Description

Calculates the complement of the chi-squared cumulative distribution function.

Method `complementaryChi` evaluates the cumulative distribution function, $1-F$, of a chi-squared random variable with `df` degrees of freedom, that is, with $x =$ `chsq` and $\nu =$ `df`,

$$p = F(x,\nu) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)}\int_0^x e^{-t/2}t^{\nu/2-1}dt$$

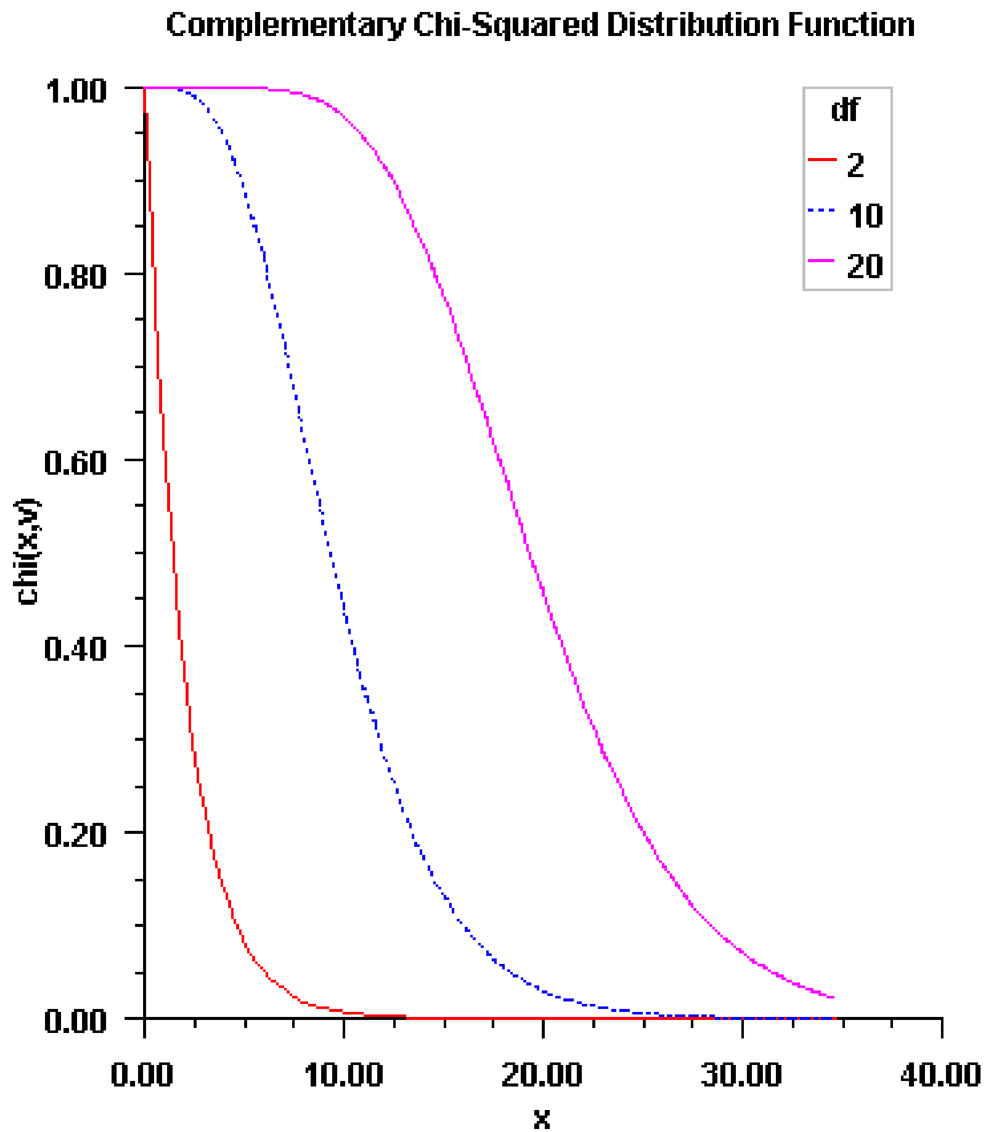
where $\Gamma(\cdot)$ is the gamma function. The value of the `complementaryChi` distribution function at the point x, $1-p$, is the probability that the random variable takes a value greater than x.

For $\nu > 343$, `complementaryChi` uses the Wilson-Hilferty approximation (Abramowitz and Stegun [A&S] 1964, equation 26.4.17) for p in terms of the normal CDF, which is evaluated using method `normal`.

For $\nu \leq 343$, `complementaryChi` uses series expansions to evaluate p: for $x < \nu$, `complementaryChi` calculates p using A&S series 6.5.29, and for $x \geq \nu$, `complementaryChi` calculates p using the continued fraction expansion of the incomplete gamma function given in A&S equation 6.5.31 and implemented using Lentz's algorithm (Lentz, W.J., 1976, Applied Optics, vol. 15,

pp. 668-671) as modified by Thompson & Barnett (Thompson, I.J., and Barnett, A.R., 1986, Journal of Computational Physics, vol. 64, pp. 490-509).

`complementaryChi` provides greater right tail accuracy for the Chi-squared distribution than does `1 - chi`.



Parameters

`chsq` – a `double` scalar value at which the function is to be evaluated.

`df` – a `double` scalar value representing the number of degrees of freedom. `df` must be positive.

Returns

a `double` scalar value representing the probability that a chi-squared random variable takes a value greater than `chsq`.

complementaryF

```
static public double complementaryF(double x, double dfn, double dfd)
```

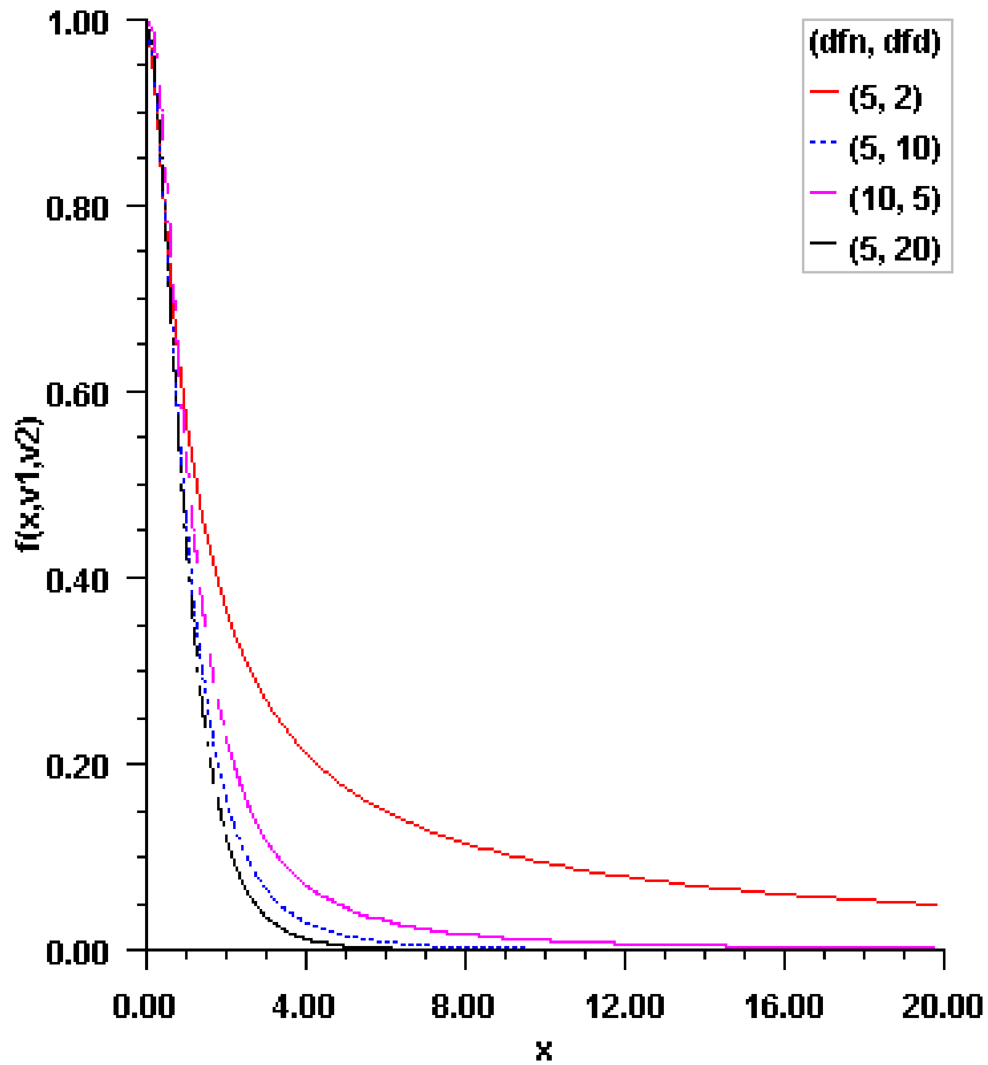
Description

Calculates the complement of the *F* distribution function.

`complementaryF` evaluates one minus 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 $q = v_2/2$. `complementaryF` also uses a relationship between *F* random variables that can be expressed as follows:

$$\mathrm{F}(X, dfn, dfd) = \mathrm{F}(1/X, dfd, dfn)$$

This function provides higher right tail accuracy for the *F* distribution.



Parameters

`x` – a `double`, the argument at which $Pr(x > F)$ 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 greater than `x`.

complementaryF2

```
static public double complementaryF2(double x, double dfn, double dfd)
```

complementaryNoncentralF

```
static public double complementaryNoncentralF(double f, double df1, double df2,
double lambda)
```

Description

Calculates the complement of the noncentral F cumulative distribution function.

The complementary noncentral F distribution is a generalization of the complementary F distribution. If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of X, then

$$F = \frac{(X/\nu_1)}{(Y/\nu_2)}$$

is a noncentral F-distributed random variable whose CDF is given by:

$$F(f,\nu_1,\nu_2,\lambda) = \sum_{j=0}^{\infty} c_j$$

where:

$$c_j = \omega_j \, I_x\left(\frac{\nu_1}{2}+j, \frac{\nu_2}{2}\right)$$

$$\omega_j = e^{-\lambda/2}\,\frac{(\lambda/2)^j}{j!} = \frac{\lambda}{2j}\,\omega_{j-1}$$

$$I_x(a,b) = \frac{\beta_x(a,b)}{\beta(a,b)}$$

$$\beta_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1}dt = x^a \sum_{j=0}^{\infty} \frac{\Gamma(j+1-b)}{(a+j)\,\Gamma(1-b)\,j!}\,x^j$$

$$\beta(a,b) = \beta_1(a,b) = \frac{\Gamma(a)\,\Gamma(b)}{\Gamma(a+b)}$$

$$I_x(a+1,b) = I_x(a,b) - T_x(a,b)$$

$$T_x(a,b) = \frac{\Gamma(a+b)}{\Gamma(a+1)\,\Gamma(b)} x^a(1-x)^b = T_x(a-1,b)\,\frac{a-1+b}{a}\,x$$

$$x = \frac{\nu_1 f}{\nu_2 + \nu_1 f}$$

and $\Gamma(\cdot)$ is the gamma function, ν_1 = df1, ν_2 = df2, λ = lambda, and f = f. The above series expansion for the noncentral F was taken from Butler and Paolella (1999) (see), with the correction for the recursion relation given below:

$$I_x(a+1,b) = I_x(a,b) - T_x(a,b)$$

extracted from the AS 63 algorithm for calculating the incomplete beta function as described by Majumder and Bhattacharjee (1973).The series approximation of the complementary (cmp) noncentral F *CDF*, denoted by $F(.)$, is obtainable by using the following identities:

$$\sum_{j=0}^{\infty} \omega_j = 1$$

$$I_{1-x}(b,a) = 1 - I_x(a,b)$$

$$\begin{aligned} I_{1-x}(b,a+1) &= 1 - I_x(a+1,b) = 1 - I_x(a,b) + T_x(a,b) \\ &= I_{1-x}(b,a) + T_x(a,b) \end{aligned}$$

Thus:

$$\begin{aligned} \overline{F}(f,\nu_1,\nu_2,\lambda) &= 1 - \sum_{j=0}^{\infty} c_j = \sum_{j=0}^{\infty} \omega_j [1 - I_x(\frac{\nu_1}{2}+j, \frac{\nu_2}{2})] \\ &= \sum_{j=0}^{\infty} \omega_j I_{1-x}(\frac{\nu_2}{2}, \frac{\nu_1}{2}+j) \end{aligned}$$

We can use the above expansion of $\overline{F}(f,\nu_1,\nu_2,\lambda)$ and the identities:

$$I_{1-x}(b,a+1) = I_{1-x}(b,a) + T_x(a,b)$$

$$T_x(a,b) = \frac{\Gamma(a+b)}{\Gamma(a+1)\,\Gamma(b)} x^a (1-x)^b = T_x(a-1,b)\,\frac{a-1+b}{a}\,x$$

to recursively calculate $\overline{F}(f,\nu_1,\nu_2,\lambda)$.

With a noncentrality parameter of zero, the noncentral *F* distribution is the same as the *F* distribution.

Parameters

f – a double value representing the argument at which the function is to be evaluated. f must be nonnegative.

df1 – a double value representing the number of numerator degrees of freedom. df1 must be positive.

df2 – a double value representing the number of denominator degrees of freedom. df2 must be positive.

lambda – a double value representing the noncentrality parameter. lambda must be nonnegative.

Returns

a double scalar value representing the probability that a noncentral *F* random variable takes a value greater than f.

complementaryStudentsT

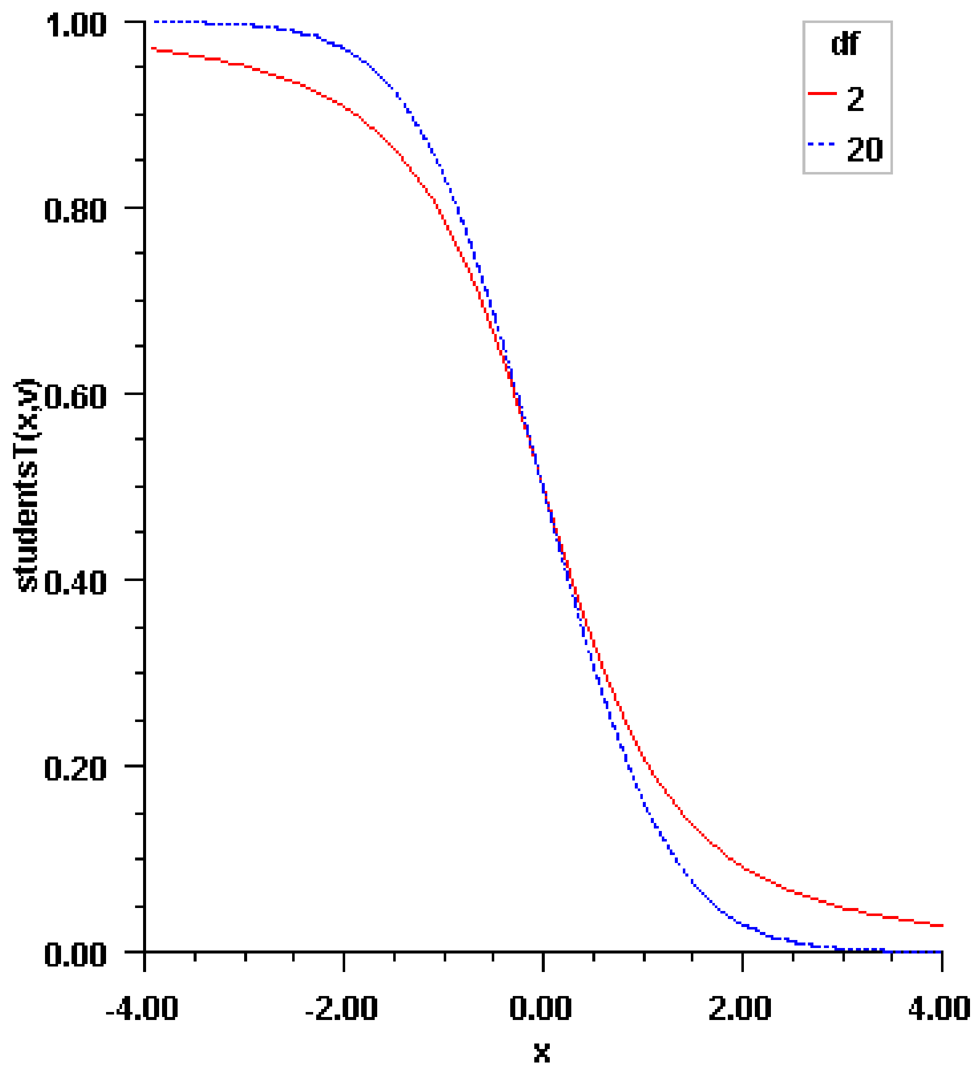
```
static public double complementaryStudentsT(double t, double df)
```

Description

Calculates the complement of the Student's *t* distribution.

Method `complementaryStudentsT` evaluates one minus 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.

This function provides higher right tail accuracy for the Student's *t* distribution.



Parameters

`t` – a `double` scalar value for which $Pr(x > t)$ 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 greater than `t`.

discreteUniform

```
static public double discreteUniform(int x, int n)
```

Description

Evaluates the discrete uniform cumulative probability distribution function.

Parameters

`x` – an `int` scalar value representing the argument at which the function is to be evaluated. `x` should be a value between the lower limit 0 and upper limit `n`

`n` – an `int` scalar value representing the upper limit of the discrete uniform distribution.

Returns

a `double` scalar value representing the probability that a discrete uniform random variable takes a value less than or equal to `x`.

exponential

```
static public double exponential(double x, double scale)
```

Description

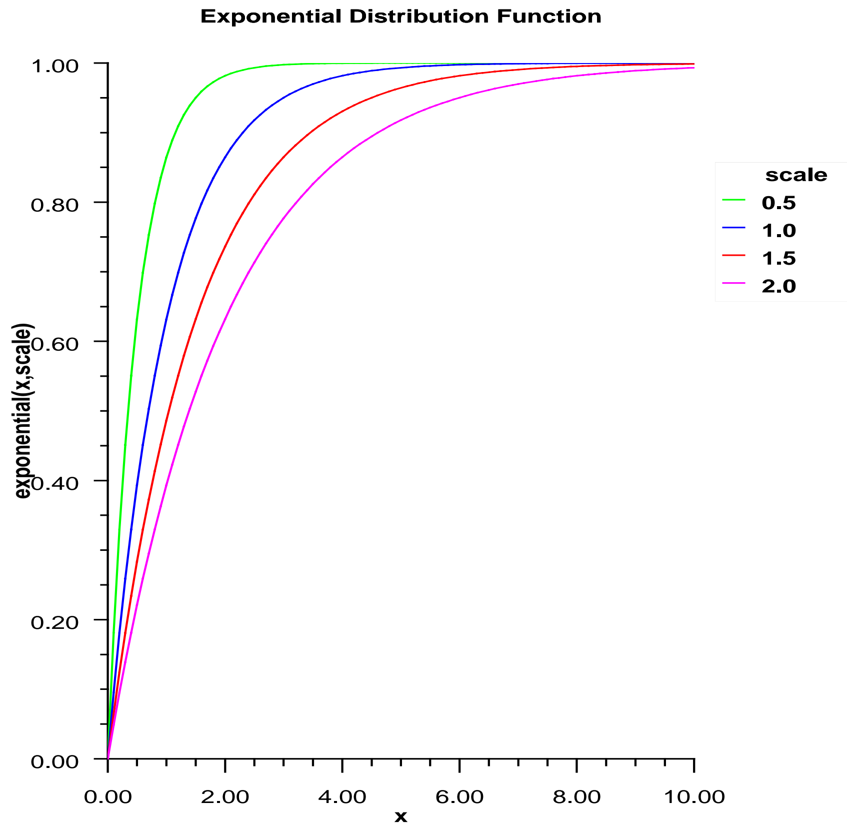
Evaluates the exponential cumulative probability distribution function.

Method `exponential` is a special case of the gamma distribution function, which evaluates the distribution function, F, with scale parameter b and shape parameter a, used in the gamma distribution function equal to 1.0. That is,

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t/b} dt$$

where $\Gamma(\cdot)$ is the gamma function. (The gamma function is the integral from 0 to ∞ 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.

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.

`scale` – a `double` scalar value representing the scale parameter, b .

Returns

a `double` scalar value representing the probability that an exponential random variable takes on a value less than or equal to `x`.

extremeValue

```
static public double extremeValue(double x, double mu, double beta)
```

Description

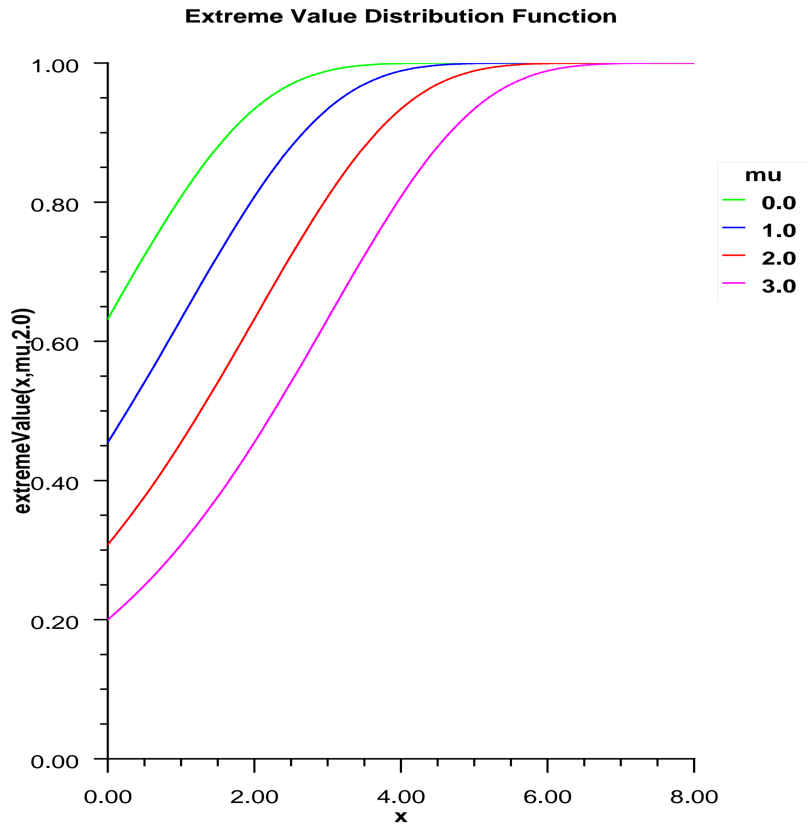
Evaluates the extreme value cumulative probability distribution function.

Method `extremeValue`, also known as the Gumbel minimum distribution, evaluates the extreme value distribution function, F, of a uniform random variable with location parameter μ and shape parameter β; that is,

$$F(x) = \int_0^x 1 - e^{-e^{\frac{x-\mu}{\beta}}} dt$$

The case where $\mu = 0$ and $\beta = 1$ is called the standard Gumbel distribution.

Random numbers are generated by evaluating uniform variates u_i, equating the continuous distribution function, and then solving for x_i by first computing $\frac{x_i-\mu}{\beta} = log(-log(1-u_i))$.



Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mu` – a `double` scalar value representing the location parameter, μ.

`beta` – a `double` scalar value representing the scale parameter, β

Returns

a `double` scalar value representing the probability that an extreme value random variable takes on a value less than or equal to `x`.

gamma

```
static public double gamma(double x, double a)
```

Description

Evaluates the gamma cumulative probability distribution function.

Method `gamma` evaluates the distribution function, F, of a gamma random variable with shape parameter a; that is,

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1} dt$$

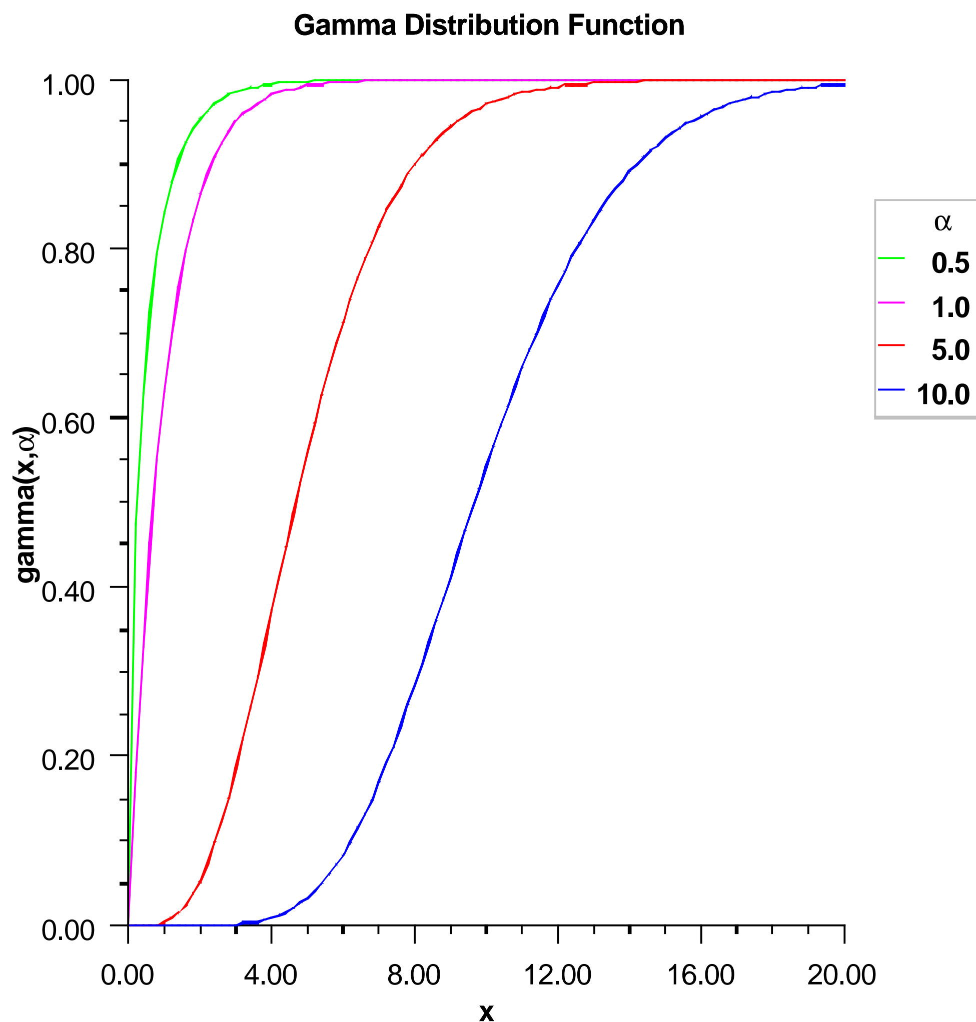
where $\Gamma(\cdot)$ is the gamma function. (The gamma function is the integral from 0 to ∞ of the same integrand as above). The value of the distribution function at the point x is the probability that the random variable takes a value less than or equal to x.

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, ∞) 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.

geometric

```
static public double geometric(int x, double pin)
```

Description

Evaluates the discrete geometric cumulative probability distribution function.

Parameters

`x` – an `int` scalar value representing the argument at which the function is to be evaluated

`pin` – an `double` scalar value representing the probability parameter for each independent trial (the probability of success for each independent trial).

Returns

a `double` scalar value representing the probability that a geometric random variable takes a value less than or equal to `x`. The return value is the probability that up to `x` trials would be observed before observing a success.

hypergeometric

```
static public double hypergeometric(int k, int sampleSize, int defectivesInLot,
int lotSize)
```

Description

Evaluates the hypergeometric cumulative probability 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`.

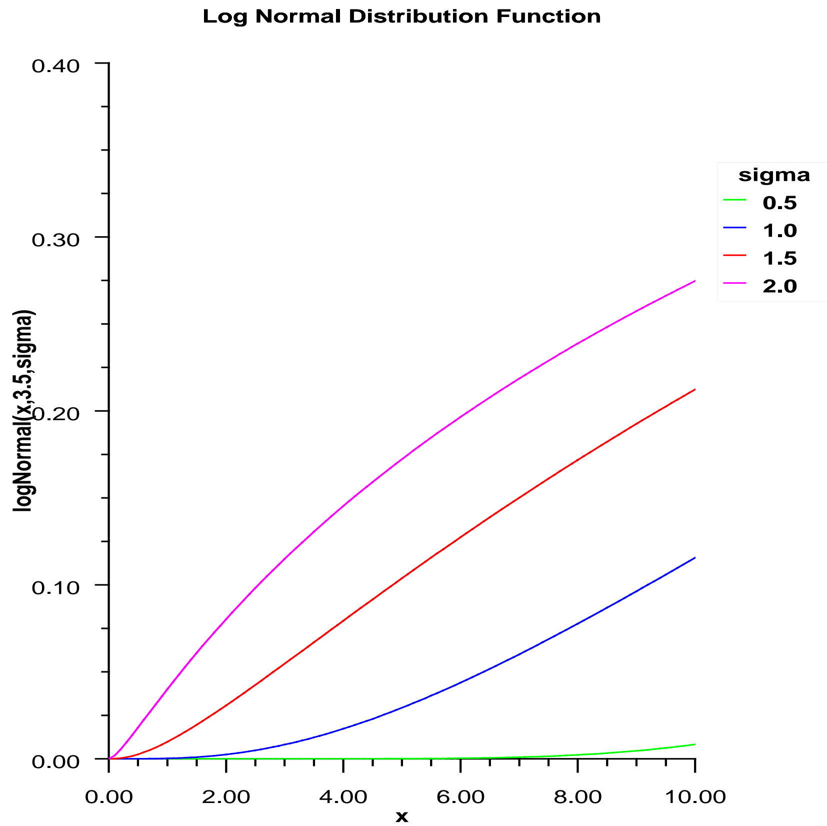
logNormal

```
static public double logNormal(double x, double mu, double sigma)
```

Description

Evaluates the standard lognormal cumulative probability distribution function.

$$F(x) = \frac{1}{x^{\sigma}\sqrt{2\pi}} \int \frac{1}{t} e^{-\frac{\ln t - \mu^2}{2\sigma^2}}$$



Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mu` – a `double` scalar value representing the location parameter.

`sigma` – a `double` scalar value representing the shape parameter. `sigma` must be a positive.

Returns

a `double` scalar value representing the probability that a standard lognormal random variable takes a value less than or equal to `x`.

logistic

```
static public double logistic(double x, double mu, double s)
```

Description

Evaluates the logistic cumulative probability distribution function.

Method `logistic` evaluates the distribution function, F, of a logistic random variable with location parameter μ and scale parameter s. It is given by

$$F(x,\mu,s) = \frac{1}{1+e^{-(x-\mu)/s}}$$

where $s > 0$.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mu` – a `double` scalar value representing the location parameter, μ.

`s` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value representing the probability that a logistic random variable takes a value less than or equal to `x`.

noncentralBeta

```
static public double noncentralBeta(double x, double shape1, double shape2,
double lambda)
```

Description

Evaluates the noncentral beta cumulative distribution function (*CDF*).

The noncentral beta distribution is a generalization of the beta distribution. If Z is a noncentral chi-square random variable with noncentrality parameter λ and $2\alpha_1$ degrees of freedom, and Y is a chi-square random variable with $2\alpha_2$ degrees of freedom which is statistically independent of Z, then

$$X = \frac{Z}{Z+Y} = \frac{\alpha_1 F}{\alpha_1 F + \alpha_2}$$

is a noncentral beta-distributed random variable and

$$F = \frac{\alpha_2 Z}{\alpha_1 Y} = \frac{\alpha_2 X}{\alpha_1 (1 - X)}$$

is a noncentral F-distributed random variable. The CDF for noncentral beta variable X can thus be simply defined in terms of the noncentral F CDF:

$$CDF_{nc\beta}(x,\ \alpha_1,\ \alpha_2,\ \lambda)\ =\ CDF_{ncF}(f,\ 2\alpha_1,\ 2\alpha_2,\ \lambda)$$

where $CDF_{nc\beta}(x,\ \alpha_1,\ \alpha_2,\ \lambda)$ is the noncentral beta CDF with x = `x`, α_1 = `shape1`, α_2 = `shape2`, and noncentrality parameter λ = `lambda`; $CDF_{ncF}(f,\ 2\alpha_1,\ 2\alpha_2,\ \lambda)$ is the noncentral F CDF with argument f, numerator and denominator degrees of freedom $2\alpha_1$ and $2\alpha_2$ respectively, and noncentrality parameter λ; and:

$$f\ =\ \frac{\alpha_2 x}{\alpha_1(1\ -\ x)};\ x\ =\ \frac{\alpha_1 f}{\alpha_1 f\ +\ \alpha_2}$$

(See documentation for class `Cdf` method `noncentralF` for a discussion of how the noncentral F CDF is defined and calculated.)

With a noncentrality parameter of zero, the noncentral beta distribution is the same as the beta distribution.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated. `x` must be nonnegative and less than or equal to 1.

`shape1` – a `double` scalar value representing the first shape parameter. `shape1` must be positive.

`shape2` – a `double` scalar value representing the second shape parameter. `shape2` must be positive.

`lambda` – a `double` scalar value representing the noncentrality parameter. `lambda` must nonnegative.

Returns

a `double` scalar value representing the probability that a noncentral beta random variable takes a value less than or equal to `x`.

noncentralF

```
static public double noncentralF(double f, double df1, double df2, double
lambda)
```

Description

Evaluates the noncentral F cumulative distribution function.

The noncentral F distribution is a generalization of the F distribution. If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of X, then

$$F\ =\ \frac{(X/\nu_1)}{(Y/\nu_2)}$$

is a noncentral F-distributed random variable whose CDF is given by:

$$CDF(f,\nu_1,\nu_2,\lambda)\ =\ \sum_{j=0}^{\infty} c_j$$

where:

$$c_j = \omega_j \, I_x\left(\frac{\nu_1}{2} + j, \frac{\nu_2}{2}\right)$$

$$\omega_j = e^{-\lambda/2} \frac{(\lambda/2)^j}{j!} = \frac{\lambda}{2j} \, \omega_{j-1}$$

$$I_x(a,b) = \frac{\beta_x(a,b)}{\beta(a,b)}$$

$$\beta_x(a,b) = \int_0^x t^{a-1}(1-t)^{b-1}dt = x^a \sum_{j=0}^{\infty} \frac{\Gamma(j+1-b)}{(a+j)\,\Gamma(1-b)\,j!} x^j$$

$$\beta(a,b) = \beta_1(a,b) = \frac{\Gamma(a)\,\Gamma(b)}{\Gamma(a+b)}$$

$$I_x(a+1,b) = I_x(a,b) - T_x(a,b)$$

$$T_x(a,b) = \frac{\Gamma(a+b)}{\Gamma(a+1)\,\Gamma(b)} x^a(1-x)^b = T_x(a-1,b)\,\frac{a-1+b}{a}\,x$$

$$x = \frac{\nu_1 f}{\nu_2 + \nu_1 f}$$

and $\Gamma(\cdot)$ is the gamma function, ν_1 = `df1`, ν_2 = `df2`, λ = `lambda`, and f = `f`.

With a noncentrality parameter of zero, the noncentral *F* distribution is the same as the *F* distribution.

Parameters

`f` – a `double` value representing the argument at which the function is to be evaluated. `f` must be nonnegative.

`df1` – a `double` value representing the number of numerator degrees of freedom. `df1` must be positive.

`df2` – a `double` value representing the number of denominator degrees of freedom. `df2` must be positive.

`lambda` – a `double` value representing the noncentrality parameter. `lambda` must be nonnegative.

Returns

a `double` scalar value representing the probability that a noncentral *F* random variable takes a value less than or equal to `f`.

noncentralchi

```
static public double noncentralchi(double chsq, double df, double alam)
```

Description

Evaluates the noncentral chi-squared cumulative probability distribution function.

Method `noncentralchi` evaluates the distribution function, F, of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `alam`, that is, with $\nu = \text{df}$, $\lambda = \text{alam}$, and $\chi = \text{chsq}$,

$$F(x) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} \int_0^x \frac{t^{(\nu+2i)/2-1} e^{-t/2}}{2^{(\nu+2i)/2}\Gamma\left(\frac{\nu+2i}{2}\right)} dt$$

where $\Gamma(\cdot)$ is the gamma function. This is a series of central chi-squared distribution functions with Poisson weights. 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 noncentral chi-squared random variable can be defined by the distribution function above, or alternatively and equivalently, as the sum of squares of independent normal random variables. If the Y_i have independent normal distributions with means μ_i and variances equal to one and

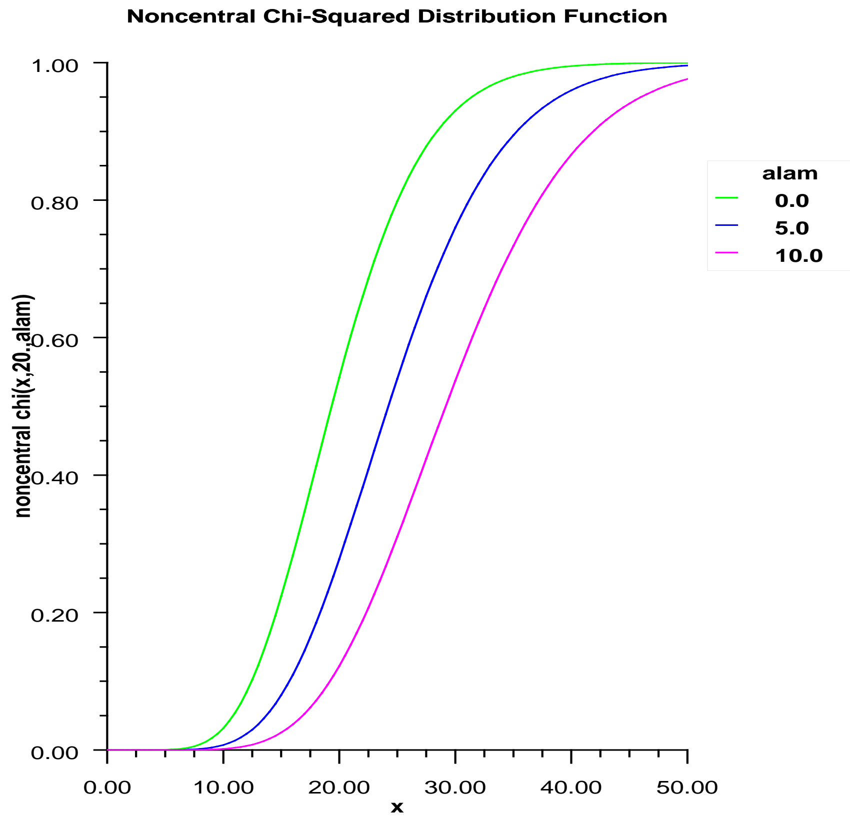
$$X = \sum_{i=1}^{n} Y_i^2$$

then X has a noncentral chi-squared distribution with n degrees of freedom and noncentrality parameter equal to

$$\sum_{i=1}^{n} \mu_i^2$$

With a noncentrality parameter of zero, the noncentral chi-squared distribution is the same as the chi-squared distribution.

`noncentralchi` determines the point at which the Poisson weight is greatest, and then sums forward and backward from that point, terminating when the additional terms are sufficiently small or when a maximum of 1000 terms have been accumulated. The recurrence relation 26.4.8 of Abramowitz and Stegun (1964) is used to speed the evaluation of the central chi-squared distribution functions.



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. `df` must be positive.

`alam` – a `double` scalar value representing the noncentrality parameter. This must be nonnegative, and `alam + df` must be less than or equal to 200,000.

Returns

a `double` scalar value representing the probability that a chi-squared random variable takes a value less than or equal to `chsq`.

noncentralstudentsT

```
static public double noncentralstudentsT(double t, int idf, double delta)
```

Description

Evaluates the noncentral Student's t cumulative probability distribution function.

Method `noncentralstudentsT` evaluates the distribution function `F` of a noncentral t random variable with `idf` degrees of freedom and noncentrality parameter `delta`; that is, with $\nu = idf$, $\delta = delta$, and $t_0 = t$,

$$F(t_0) = \int_{-\infty}^{t_0} \frac{\nu^{\nu/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(\nu/2)(\nu+x^2)^{(\nu+1)/2}} \sum_{i=0}^{\infty} \Gamma((\nu+i+1)/2) \left(\frac{\delta^i}{i!}\right) \left(\frac{2x^2}{\nu+x^2}\right)^{i/2} dx$$

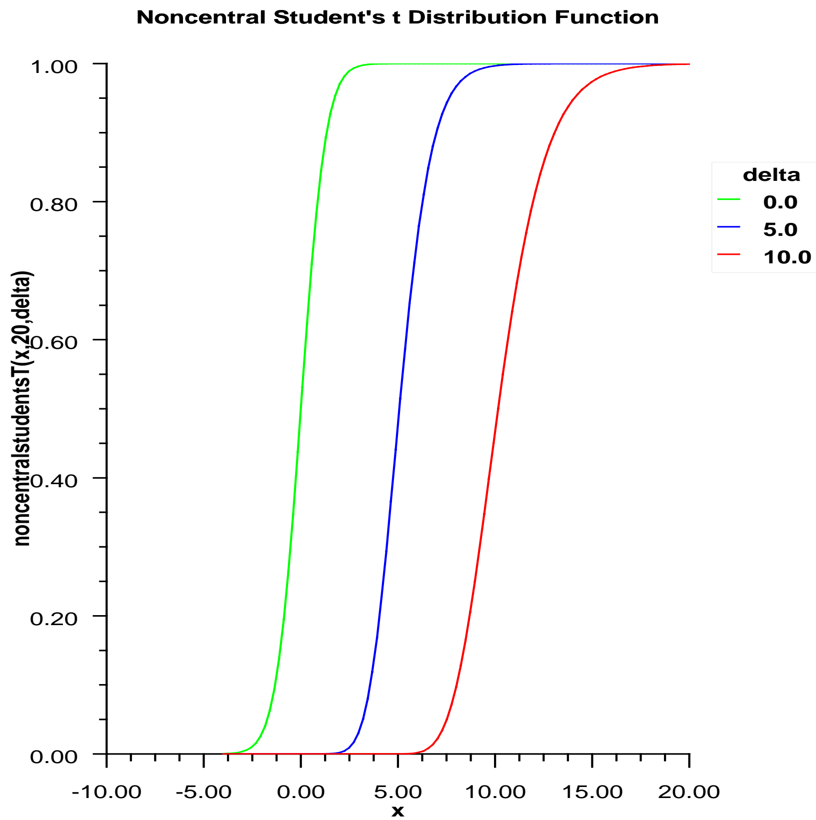
where $\Gamma(\cdot)$ is the gamma function. The value of the distribution function at the point t_0 is the probability that the random variable takes a value less than or equal to t_0.

The noncentral t random variable can be defined by the distribution function above, or alternatively and equivalently, as the ratio of a normal random variable and an independent chi-squared random variable. If w has a normal distribution with mean δ and variance equal to one, u has an independent chi-squared distribution with ν degrees of freedom, and

$$x = w/\sqrt{u/\nu}$$

then x has a noncentral t distribution with ν degrees of freedom and noncentrality parameter δ.

The distribution function of the noncentral t can also be expressed as a double integral involving a normal density function (see, for example, Owen 1962, page 108). The method `noncentralstudentsT` uses the method of Owen (1962, 1965), which uses repeated integration by parts on that alternate expression for the distribution function.



Parameters

`t` – a `double` scalar value representing the argument at which the function is to be evaluated.

`idf` – an `int` scalar value representing the number of degrees of freedom. This must be positive.

`delta` – a `double` scalar value representing the noncentrality parameter.

Returns

a `double` scalar value representing the probability that a noncentral Student's t random variable takes a

value less than or equal to `t`.

normal

```
static public double normal(double x)
```

Description

Evaluates the normal (Gaussian) cumulative probability distribution function.

Method `normal` evaluates the distribution function, Φ, of a standard normal (Gaussian) random variable, that is,

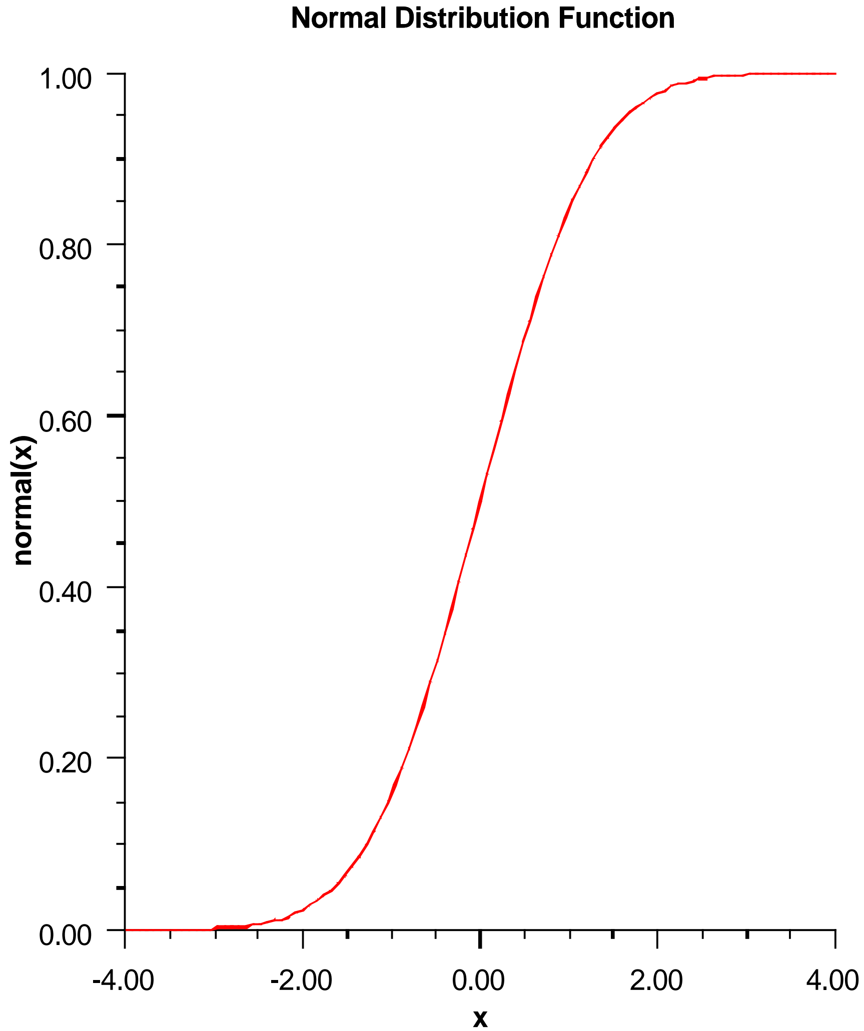
$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$$

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 μ and variance σ^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:

$$\Phi(x) = \text{erfc}(-x/\sqrt{2.0})/2$$



Parameter

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

```
static public double poisson(int k, double theta)
```

Description

Evaluates the Poisson cumulative probability 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

$$f(x+1) = f(x)\,(\theta/(x+1)),\ for x = 0, 1, 2, \ldots k-1$$

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*.

studentsT

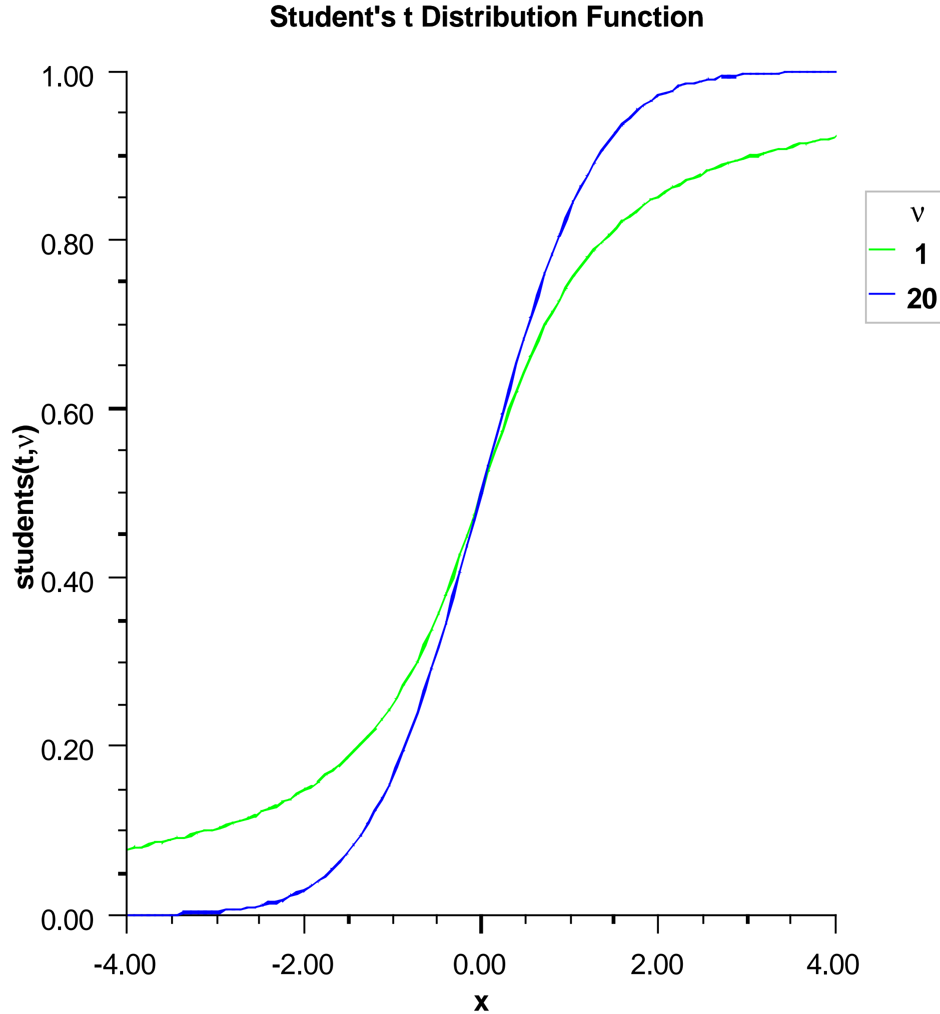
```
static public double studentsT(double t, double df)
```

Description

Evaluates the Student's t cumulative probability 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.

For greater right tail accuracy, see `com.imsl.stat.Cdf.complementaryStudentsT` (p. 1255) .



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`.

uniform

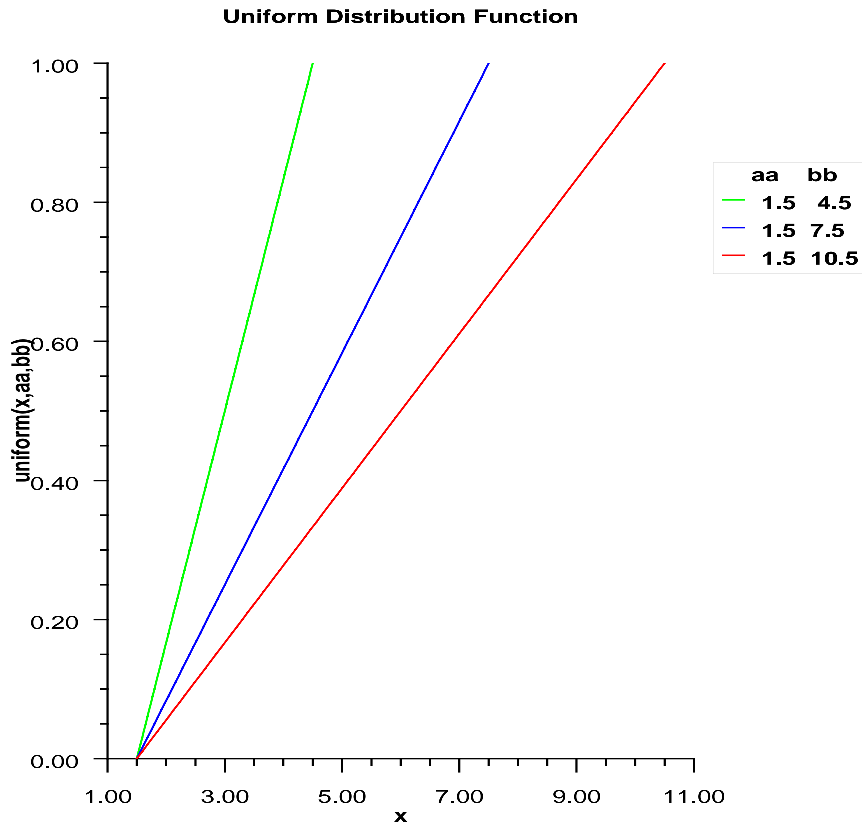
```
static public double uniform(double x, double aa, double bb)
```

Description

Evaluates the uniform cumulative probability distribution function.

Method `uniform` evaluates the distribution function, F, of a uniform random variable with location parameter aa and scale parameter bb; that is,

$$f(x) = \begin{cases} 0 & \text{if } x < aa \\ \frac{x-aa}{bb-aa} & \text{if } aa \le x \le bb \\ 1 & \text{if } x > bb \end{cases}$$



Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`aa` – a `double` scalar value representing the location parameter.

`bb` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value representing the probability that a uniform 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, result, pin, qin;
        int k, n;
        // Beta
        x = .5;
        pin = 12.;
        qin = 12.;
        result = Cdf.beta(x, pin, qin);
        System.out.println("beta(.5, 12., 12.) is " + result);

        // binomial
        k = 3;
        n = 5;
        pin = .95;
        result = Cdf.binomial(k, n, pin);
        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);
    }
}
```

Output

```
beta(.5, 12., 12.) is 0.5000000000000016
binomial(3, 5, .95) is 0.02259250000000004
chi(.15, 2) is 0.07225651367144709
```

Pdf class

```
public final class com.imsl.stat.Pdf
```

Probability density functions.

Methods

F

```
static public double F(double x, double dfn, double dfd)
```

Description

Evaluates the F probability density function.

The probability density function of the F distribution is

$$f(x,dfn,dfd) = \frac{\Gamma(\frac{v_1+v_2}{2})(\frac{v_1}{v_2})^{\frac{v_1}{2}}x^{\frac{v_1}{2}}}{\Gamma(\frac{v_1}{2})\Gamma(\frac{v_2}{2})(1+\frac{v_1 x}{v_2})^{\frac{v_1+v_2}{2}}}$$

where v_1 and v_2 are the shape parameters `dfn` and `dfd` and Γ is the gamma function,

$$\Gamma(a) = \int_0^\infty t^{a-1}e^{-t}dt$$

.

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 value of the probability density function at `x`.

Pareto

```
static public double Pareto(double x, double xm, double k)
```

Description

Evaluates the Pareto probability density function.

The probability density function of the Pareto distribution is

$$f(x,x_m,k) = 1 - \frac{kx_m^k}{x^{k+1}}$$

where the scale parameter $x_m > 0$ and the shape parameter $k > 0$. The function is only defined for $x \geq x_m$

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`xm` – a `double` scalar value representing the scale parameter, x_m.

`k` – a `double` scalar value representing the shape parameter.

Returns

a `double` scalar value representing the probability density function at `x`.

Rayleigh

```
static public double Rayleigh(double x, double alpha)
```

Description

Evaluates the Rayleigh probability density function.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated. It must be non-negative.

`alpha` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value representing the probability density function at `x`.

Weibull

```
static public double Weibull(double x, double gamma, double alpha)
```

Description

Evaluates the Weibull probability density 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, the probability density function at `x`.

beta

```
static public double beta(double x, double pin, double qin)
```

Description

Evaluates the beta probability density function.

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 value of the probability density function at `x`.

binomial

```
static public double binomial(int k, int n, double pin)
```

Description

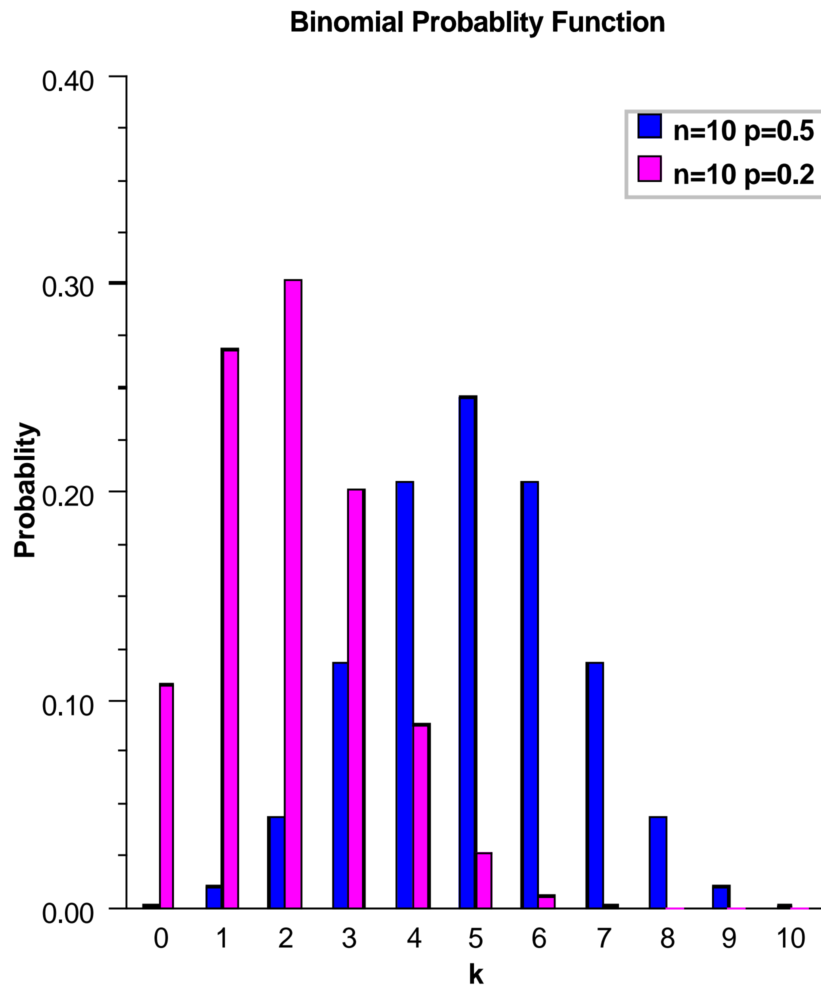
Evaluates the binomial probability density function.

Method `binomial` evaluates the probability that a binomial random variable with parameters n and p with p=pin 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

$$\Pr(X = j) = \frac{(n+1-j)\,p}{j\,(1-p)}\Pr(X = j-1)$$

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, ε, 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 $p = 0$, `binomial` is set to 0 if k is greater than 0 and to 1 otherwise; and for the case $p = 1$, `binomial` 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.

`pin` – a `double` scalar value representing the probability of success on each independent trial.

Returns

a `double` scalar value representing the probability that a binomial random variable takes a value equal to

k.

chi

```
static public double chi(double chsq, double df)
```

Description

Evaluates the chi-squared probability density function

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. `df` must be positive.

Returns

a `double` scalar value, the value of the probability density function at `chsq`.

discreteUniform

```
static public double discreteUniform(int x, int n)
```

Description

Evaluates the discrete uniform probability density function.

Parameters

`x` – an `int` argument for which the discrete uniform probability density function is to be evaluated. `x` should be a value between the lower limit 0 and upper limit `n`

`n` – an `int` scalar value representing the upper limit of the discrete uniform distribution.

Returns

a `double` scalar value representing the probability that a discrete uniform random variable takes a value equal to `x`.

exponential

```
static public double exponential(double x, double scale)
```

Description

Evaluates the exponential probability density function

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`scale` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value, the value of the probability density function at `x`.

extremeValue

```
static public double extremeValue(double x, double mu, double beta)
```

Description

Evaluates the extreme value probability density function.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mu` – a `double` scalar value representing the location parameter.

`beta` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value representing the probability density function at `x`.

gamma

```
static public double gamma(double x, double a, double b)
```

Description

Evaluates the gamma probability density function. The probability density function of the gamma distribution is

$$f(x;a,b) = x^{a-1} \frac{1}{b^a \Gamma(a)} e^{-x/b}$$

where `a` is the shape parameter and `b` is the scale parameter.

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.

`b` – a `double` scalar value representing the scale parameter. This must be positive.

Returns

a `double` scalar value, the probability density function at `x`.

geometric

```
static public double geometric(int x, double pin)
```

Description

Evaluates the discrete geometric probability density function.

Method `geometric` evaluates the geometric distribution for the number of trials before the first success.

Parameters

`x` – the `int` argument for which the geometric probability function is to be evaluated

`pin` – a `double` scalar value representing the probability parameter of the geometric distribution (the probability of success for each independent trial)

Returns

a `double` scalar value representing the probability that a geometric random variable takes a value equal to `x`.

hypergeometric

```
static public double hypergeometric(int k, int sampleSize, int defectivesInLot,
int lotSize)
```

Description

Evaluates the hypergeometric probability density function.

Method `hypergeometric` evaluates the probability density 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 density 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 $i = max(0, n - l + m)$. `hypergeometric` 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`.

logNormal

```
static public double logNormal(double x, double mu, double sigma)
```

Description

Evaluates the standard lognormal probability density function.

$$F(x)=\frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$$

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mu` – a `double` scalar value representing the location parameter.

`sigma` – a `double` scalar value representing the shape parameter. `sigma` must be a positive.

Returns

a `double` scalar value representing the probability density function at `x`.

logistic

```
static public double logistic(double x, double mu, double s)
```

Description

Evaluates the logistic probability density function.

The probability density function of the logistic distribution is

$$f(x,\mu,s) = \frac{e^{-(x-\mu)/s}}{s\left(1+e^{-(x-\mu)/s}\right)^2}$$

where μ is the location parameter and the scale parameter $s > 0$.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mu` – a `double` scalar value representing the location parameter.

`s` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value representing the probability density function at `x`.

noncentralBeta

```
static public double noncentralBeta(double x, double shape1, double shape2,
double lambda)
```

Description

Evaluates the noncentral beta probability density function (PDF).

The noncentral beta distribution is a generalization of the beta distribution. If Z is a noncentral chi-square random variable with noncentrality parameter λ and $2\alpha_1$ degrees of freedom, and Y is a chi-square random variable with $2\alpha_2$ degrees of freedom which is statistically independent of Z, then

$$X = \frac{Z}{Z+Y} = \frac{\alpha_1 F}{\alpha_1 F + \alpha_2}$$

is a noncentral beta-distributed random variable and

$$F = \frac{\alpha_2 Z}{\alpha_1 Y} = \frac{\alpha_2 X}{\alpha_1(1-X)}$$

is a noncentral F-distributed random variable. The PDF for noncentral beta variable X can thus be simply defined in terms of the noncentral F PDF:

$$PDF_{nc\beta}(x,\, \alpha_1,\, \alpha_2,\, \lambda) = PDF_{ncF}(f,\, 2\alpha_1,\, 2\alpha_2,\, \lambda)\, \frac{df}{dx}$$

where $PDF_{nc\beta}(x, \alpha_1, \alpha_2, \lambda)$ is the noncentral beta PDF with x = x, α_1 = shape1, α_2 = shape2, and noncentrality parameter λ = lambda; $PDF_{ncF}(f, 2\alpha_1, 2\alpha_2, \lambda)$ is the noncentral F PDF with argument f, numerator and denominator degrees of freedom $2\alpha_1$ and $2\alpha_2$ respectively, and noncentrality parameter λ; and

$$\frac{df}{dx} = \frac{(\alpha_1 f + \alpha_2)^2}{\alpha_1 \alpha_2} = \frac{\alpha_2}{\alpha_1 (1 - x)^2}$$

where

$$f = \frac{\alpha_2 x}{\alpha_1 (1 - x)}$$

and

$$x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2}$$

(See documentation for class `Cdf` method `noncentralF` for a discussion of how the noncentral F PDF is defined and calculated.)

With a noncentrality parameter of zero, the noncentral beta distribution is the same as the beta distribution.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated. `x` must be nonnegative and less than or equal to 1.

`shape1` – a `double` scalar value representing the first shape parameter. `shape1` must be positive.

`shape2` – a `double` scalar value representing the second shape parameter. `shape2` must be positive.

`lambda` – a `double` scalar value representing the noncentrality parameter. `lambda` must nonnegative.

Returns

a `double` scalar value representing the probability density associated with a noncentral beta random variable with value `x`.

noncentralChi

```
static public double noncentralChi(double chsq, double df, double alam)
```

Description

Evaluates the noncentral chi-squared probability density function (PDF).

The noncentral chi-squared distribution is a generalization of the chi-squared distribution. If $\{X_i\}$ are k independent, normally distributed random variables with means μ_i and variances σ_i^2, then the random variable

$$X = \sum_{i=1}^{k} \left(\frac{X_i}{\sigma_i} \right)^2$$

is distributed according to the noncentral chi-squared distribution. The noncentral chi-squared distribution has two parameters, k which specifies the number of degrees of freedom (i.e. the number of X_i), and λ which is related to the mean of the random variables X_i by

$$\lambda = \sum_{i=1}^{k} \left(\frac{\mu_i}{\sigma_i} \right)^2$$

The noncentral chi-squared distribution is equivalent to a (central) chi-squared distribution with $k+2i$ degrees of freedom, where i is the value of a Poisson distributed random variable with parameter $\lambda/2$. Thus, the probability density function is given by:

$$F(x,k,\lambda) = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2}(\lambda/2)^i}{i!} f(x,k+2i)$$

where the (central) chi-squared PDF $f(x,k)$ is given by:

$$f(x,k) = \frac{(x/2)^{k/2}\, e^{-x/2}}{x\,\Gamma(k/2)} \quad for\ x > 0,\ else\ 0$$

where $\Gamma(\cdot)$ is the gamma function. The above representation of $F(x,k,\lambda)$ can be shown to be equivalent to the representation:

$$F(x,k,\lambda) = \frac{e^{-(\lambda+x)/2}\,(x/2)^{k/2}}{x} \sum_{i=0}^{\infty} \phi_i$$

$$\phi_i = \frac{(\lambda x/4)^i}{i!\,\Gamma(k/2\ +\ i)}$$

Method `noncentralChi` evaluates the probability density function, $F(x,k,\lambda)$, of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `alam`, corresponding to k = `df`, λ = `alam`, and x = `chsq`.

Method `noncentralChi` evaluates the cumulative distribution function incorporating the above probability density function.

With a noncentrality parameter of zero, the noncentral chi-squared distribution is the same as the central chi-squared distribution.

Parameters

`chsq` – a `double` scalar value at which the function is to be evaluated. `chsq` must be nonnegative.

`df` – a `double` scalar value representing the number of degrees of freedom. `df` must be positive.

`alam` – a `double` scalar value representing the noncentrality parameter. `alam` must be nonnegative.

Returns

a `double` scalar value representing the probability density associated with a noncentral chi-squared random variable with value `chsq`.

noncentralF

```
static public double noncentralF(double f, double df1, double df2, double
lambda)
```

Description

Evaluates the noncentral *F* probability density function (*PDF*).

The noncentral *F* distribution is a generalization of the *F* distribution. If *x* is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and *y* is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of *X*, then

$$F = (x/\nu_1)/(y/\nu_2)$$

is a noncentral *F*-distributed random variable whose PDF is given by:

$$PDF(f, \nu_1, \nu_2, \lambda) = \Psi \sum_{k=0}^{\infty} \Phi_k$$

where

$$\Psi = \frac{e^{-\lambda/2}(\nu_1 f)^{\nu_1/2}(\nu_2)^{\nu_2/2}}{f\,(\nu_1 f + \nu_2)^{(\nu_1+\nu_2)/2}\,\Gamma(\nu_2/2)}$$

$$\Phi_k = \frac{R^k\,\Gamma(\frac{\nu_1+\nu_2}{2} + k)}{k!\,\Gamma(\frac{\nu_1}{2} + k)}$$

$$R = \frac{\lambda \nu_1 f}{2(\nu_1 f + \nu_2)}$$

where $\Gamma(\cdot)$ is the gamma function, ν_1 = `df1`, ν_2 = `df2`, λ = `lambda`, and f = `f`.

With a noncentrality parameter of zero, the noncentral *F* distribution is the same as the *F* distribution.

The efficiency of the calculation of the above series is enhanced by:

1. calculating each term Φ_k in the series recursively in terms of either the term Φ_{k-1} preceding it or the term Φ_{k+1} following it, and
2. initializing the sum with the largest series term and adding the subsequent terms in order of decreasing magnitude.

Special cases:

For $R = \lambda f = 0$:

$$PDF(f, \nu_1, \nu_2, \lambda) = \Psi\,\Phi_0 = \Psi \frac{\Gamma([\nu_1 + \nu_2]/2)}{\Gamma(\nu_1/2)}$$

For $\lambda = 0$:

$$PDF(f, \nu_1, \nu_2, \lambda) = \frac{(\nu_1 f)^{\nu_1/2}\,(\nu_2)^{\nu_2/2}\,\Gamma([\nu_1 + \nu_2]/2)}{f\,(\nu_1 f + \nu_2)^{(\nu_1+\nu_2)/2}\,\Gamma(\nu_1/2)\,\Gamma(\nu_2/2)}$$

For $f = 0$:

$$PDF(f, \nu_1, \nu_2, \lambda) = \frac{e^{-\lambda/2}\,f^{\nu_1/2\,-\,1}\,(\nu_1/\nu_2)^{\nu_1/2}\,\Gamma([\nu_1 + \nu_2]/2)}{\Gamma(\nu_1/2)\,\Gamma(\nu_2/2)}$$

$$PDF(f, \nu_1, \nu_2, \lambda) = \begin{cases} 0 & \text{if } \nu_1 > 2; \\ e^{-\lambda/2} & \text{if } \nu_1 = 2; \\ \infty & \text{if } \nu_1 < 2 \end{cases}$$

Parameters

`f` – a `double` value representing the argument at which the function is to be evaluated. `f` must be nonnegative.

`df1` – a `double` value representing the number of numerator degrees of freedom. `df1` must be positive.

`df2` – a `double` value representing the number of denominator degrees of freedom. `df2` must be positive.

`lambda` – a `double` value representing the noncentrality parameter. `lambda` must be nonnegative.

Returns

a `double` value representing the probability density associated with a noncentral F random variable with value `f`.

noncentralStudentsT

```
static public double noncentralStudentsT(double t, double df, double delta)
throws Pdf.AltSeriesAccuracyLossException
```

Description

Evaluates the noncentral Student's t probability density function.

The noncentral Student's t-distribution is a generalization of the Student's t-distribution. If w is a normally distributed random variable with unit variance and mean δ and u is a chi-square random variable with ν degrees of freedom that is statistically independent of w, then

$$T = w/\sqrt{u/\nu}$$

is a noncentral t-distributed random variable with ν degrees of freedom and noncentrality parameter δ, that is, with ν = `df`, and δ = `delta`. The probability density function for the noncentral t-distribution is:

$$f(t,\nu,\delta) = \frac{\nu^{\nu/2}\, e^{-\delta^2/2}}{\sqrt{\pi}\,\Gamma(\nu/2)\,(\nu+t^2)^{(\nu+1)/2}} \sum_{i=0}^{\infty} \Phi_i$$

where

$$\Phi_i = \frac{\Gamma((\nu+i+1)/2)}{i!}\,[\delta t]^i \left(\frac{2}{\nu+t^2}\right)^{i/2}$$

and t = `t`.

For noncentrality parameter $\delta = 0$, the PDF reduces to the (central) Student's t PDF:

$$f(t,\nu,0) = \frac{\Gamma((\nu+1)/2)\,\left(1 + (t^2/\nu)\right)^{-(\nu+1)/2}}{\sqrt{\nu\pi}\,\Gamma(\nu/2)}$$

and, for $t = 0$, the PDF becomes:

$$f(0,\nu,\delta) = \frac{\Gamma((\nu+1)/2)\, e^{-\delta^2/2}}{\sqrt{\nu\pi}\,\Gamma(\nu/2)}$$

Method `noncentralStudentsT` evaluates the cumulative distribution function incorporating the above probability density function.

Parameters

`t` – a `double` value representing the argument at which the function is to be evaluated.

`df` – a `double` value representing the number of degrees of freedom. `df` must be positive.

`delta` – a `double` value representing the noncentrality parameter.

Returns

a `double` value representing the probability density associated with a noncentral Student's *t* random variable with value `t`.

Exception

`AltSeriesAccuracyLossException` is thrown when the magnitude of alternating series sum is too small relative to the sum of positive terms to permit a reliable accuracy.

normal

```
static public double normal(double x, double mean, double stdev)
```

Description

Evaluates the normal (Gaussian) probability density function.

The probability density function for a normal distribution is given by

$$\frac{1}{\sigma\sqrt{2\pi}}e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

where μ and σ are the conditional mean and standard deviation.

Parameters

`x` – a `double` scalar value representing the argument at which the function is to be evaluated.

`mean` – a `double` scalar value containing the mean.

`stdev` – a `double` scalar value containing the standard deviation.

Returns

a `double` containing the value of the probability density function at `x`

poisson

```
static public double poisson(int k, double theta)
```

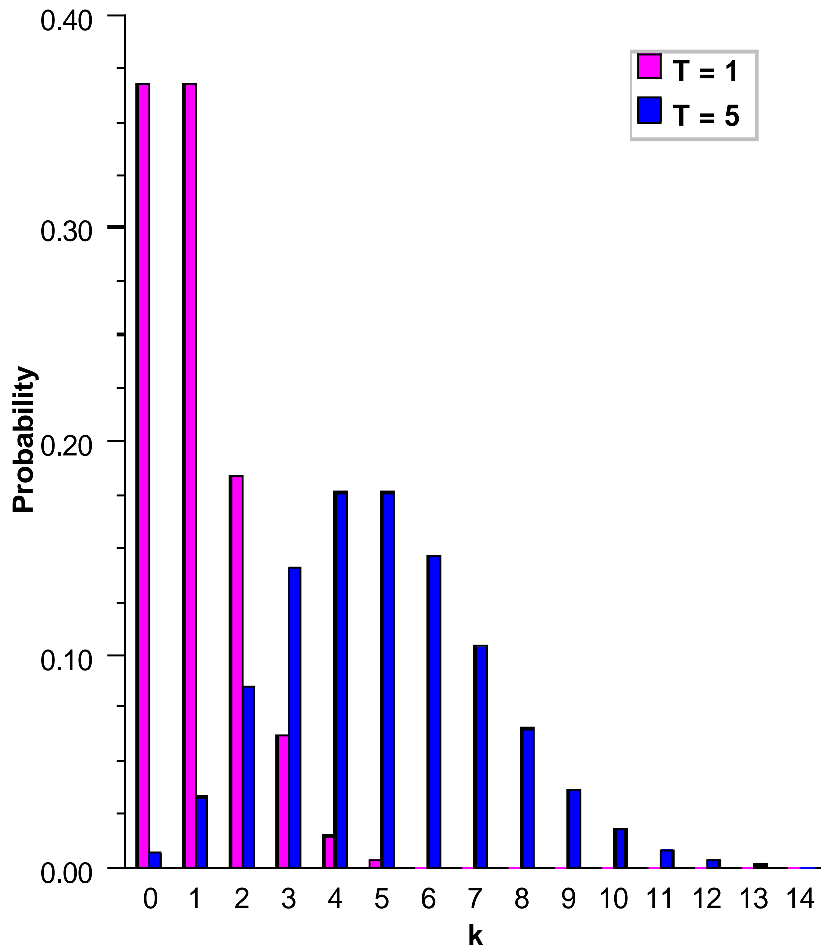
Description

Evaluates the Poisson probability density function.

Method `poisson` evaluates the probability density 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

$$f(x) = e^{-\theta}\,\theta^k/k!, \quad for k = 0,\ 1,\ 2, \ldots$$

`poisson` 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.

Example: The Probability Density Functions

Various probability density functions are exercised. Their use in this example typifies the manner in which other functions in the Pdf class would be used.

```
import com.imsl.stat.Pdf;

public class PdfEx1 {

    public static void main(String args[]) {
        //F Probability density function
        double result = Pdf.F(1.0, 100.0, 100.0);
        System.out.println("Pdf.F(1.0, 100.0, 100.0) is " + result);

        result = Pdf.normal(0.0, 0.0, 5.0);
        System.out.println("Pdf.normal(0.0, 0.0, 5.0) is " + result);
    }
}
```

Output

```
Pdf.F(1.0, 100.0, 100.0) is 1.9897309346795427
Pdf.normal(0.0, 0.0, 5.0) is 0.07978845608028655
```

Pdf.AltSeriesAccuracyLossException class

`static public class com.imsl.stat.Pdf.AltSeriesAccuracyLossException extends com.imsl.IMSLException`

The magnitude of alternating series sum is too small relative to the sum of positive terms to permit a reliable accuracy.

Constructors

Pdf.AltSeriesAccuracyLossException

`public Pdf.AltSeriesAccuracyLossException(String message)`

Description

The magnitude of alternating series sum is too small relative to the sum of positive terms to permit a reliable accuracy.

Parameter

`message` – a `String` containing the message explaining the cause of the exception.

Pdf.AltSeriesAccuracyLossException

```
public Pdf.AltSeriesAccuracyLossException(String key, Object[] arguments)
```

Description

The magnitude of alternating series sum is too small relative to the sum of positive terms to permit a reliable accuracy.

Parameters

`key` – a `String` containing the message explaining the cause of the exception.

`arguments` – an `Object` containing arguments for message `String key`.

InvCdf class

```
public final class com.imsl.stat.InvCdf
```

Inverse cumulative probability distribution functions.

Methods

F

```
static public double F(double p, double dfn, double dfd)
```

Description

Returns the inverse of the F cumulative probability distribution function.

Method `F` 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 `beta`. If X is an F variate with v_1 and v_2 degrees of freedom and $Y = v_1 X/(v_2 + v_1 X)$, then Y is a beta variate with parameters $p = v_1/2$ and $q = v_2/2$. If $P \leq 0.5$, `F` 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 returned value is `p`.

Pareto

```
static public double Pareto(double p, double xm, double k)
```

Description

Returns the inverse of the Pareto cumulative probability density function.

Parameters

`p` – a `double` scalar value representing the probability for which the inverse Pareto function is to be evaluated.

`xm` – a `double` scalar value representing the scale parameter, x_m.

`k` – a `double` scalar value representing the shape parameter.

Returns

a `double` scalar value. The probability that a Pareto random variable takes on a value less than or equal to this returned value is `p`.

Rayleigh

```
static public double Rayleigh(double p, double alpha)
```

Description

Returns the inverse of the Rayleigh cumulative probability distribution function.

Parameters

`p` – a `double` scalar value representing the probability for which the inverse Rayleigh function is to be evaluated.

`alpha` – a `double` scalar value representing the scale parameter.

Returns

a `double` scalar value. The probability that a Rayleigh random variable takes a value less than or equal to this returned value is `p`.

Weibull

```
static public double Weibull(double p, double gamma, double alpha)
```

Description

Returns the inverse of the Weibull cumulative probability distribution function.

Parameters

p – a double scalar value representing the probability for which the inverse Weibull function is to be evaluated.

gamma – a double scalar value representing the shape parameter.

alpha – a double scalar value representing the scale parameter.

Returns

a double scalar value. The probability that a Weibull random variable takes a value less than or equal to this returned value is p.

beta

```
static public double beta(double p, double pin, double qin)
```

Description

Evaluates the inverse of the beta cumulative probability distribution function.

Method beta 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 beta (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 returned value is p.

chi

```
static public double chi(double p, double df)
```

Description

Evaluates the inverse of the chi-squared cumulative probability distribution function.

Method chi evaluates the inverse distribution function of a chi-squared random variable with df degrees of freedom, that is, with $P = p$ and $\nu = df$, it determines x (equal to chi(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 $v < 40$, `chi` 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 `normal` 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. The probability that a chi-squared random variable takes a value less than or equal to this returned value is `p`.

discreteUniform

```
static public int discreteUniform(double p, int n)
```

Description

Returns the inverse of the discrete uniform cumulative probability distribution function.

Parameters

`p` – a `double` scalar value representing the probability for which the inverse discrete uniform function is to be evaluated

`n` – an `int` scalar value representing the upper limit of the discrete uniform distribution

Returns

an `int` scalar value. The probability that a discrete uniform random variable takes a value less than or equal to this returned value is `p`.

exponential

```
static public double exponential(double p, double scale)
```

Description

Evaluates the inverse of the exponential cumulative probability distribution function.

Method `exponential` evaluates the inverse distribution function of a gamma random variable with scale parameter =b and shape parameter a=1.0, that is, it determines $x = \text{exponential}(p, 1.0))$, such that

$$P = \frac{1}{\Gamma(a)} \int_o^x e^{-t/b} dt$$

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.

exponential 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.

scale – a double scalar value representing the scale parameter.

Returns

a double scalar value. The probability that an exponential random variable takes a value less than or equal to this returned value is p.

extremeValue

```
static public double extremeValue(double p, double mu, double beta)
```

Description

Returns the inverse of the extreme value cumulative probability distribution function.

Parameters

p – a double scalar value representing the probability for which the inverse extreme value function is to be evaluated.

mu – a double scalar value representing the location parameter.

beta – a double scalar value representing the scale parameter.

Returns

a double scalar value. The probability that an extreme value random variable takes a value less than or equal to this returned value is p.

gamma

```
static public double gamma(double p, double a)
```

Description

Evaluates the inverse of the gamma cumulative probability distribution function.

Method gamma evaluates the inverse distribution function of a gamma random variable with shape parameter a, that is, it determines $x = \text{gamma}(p,a))$, 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.

gamma 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. The probability that a gamma random variable takes a value less than or equal to this returned value is p.

geometric

```
static public double geometric(double r, double pin)
```

Description

Returns the inverse of the discrete geometric cumulative probability distribution function.

Parameters

r – a double scalar value representing the probability for which the inverse geometric function is to be evaluated

pin – an int scalar value representing the probability parameter for each independent trial (the probability of success for each independent trial).

Returns

a double scalar value. The probability that a geometric random variable takes a value less than or equal to this returned value is the input probability, r.

logNormal

```
static public double logNormal(double p, double mu, double sigma)
```

Description

Returns the inverse of the standard lognormal cumulative probability distribution function.

Parameters

p – a double scalar value representing the probability for which the inverse lognormal function is to be evaluated.

mu – a double scalar value representing the location parameter.

sigma – a double scalar value representing the shape parameter. sigma must be a positive.

Returns

a double scalar value. The probability that a standard lognormal random variable takes a value less than or equal to this returned value is p.

logistic

```
static public double logistic(double p, double mu, double s)
```

Description

Returns the inverse of the logistic cumulative probability distribution function.

Parameters

p – a double scalar value representing the probability for which the inverse logistic function is to be evaluated.

mu – a double scalar value representing the location parameter, μ.

s – a double scalar value representing the scale parameter.

Returns

a double scalar value. The probability that a logistic random variable takes a value less than or equal to this returned value is p.

noncentralBeta

```
static public double noncentralBeta(double p, double shape1, double shape2,
double lambda)
```

Description

Evaluates the inverse of the noncentral beta cumulative distribution function (*CDF*).

If Z is a noncentral chi-square random variable with noncentrality parameter λ and $2\alpha_1$ degrees of freedom, and Y is a chi-square random variable with $2\alpha_2$ degrees of freedom which is statistically independent of Z, then

$$X = \frac{Z}{Z + Y} = \frac{\alpha_1 F}{\alpha_1 F + \alpha_2}$$

is a noncentral beta-distributed random variable and

$$F = \frac{\alpha_2 Z}{\alpha_1 Y} = \frac{\alpha_2 X}{\alpha_1 (1 - X)}$$

is a noncentral F-distributed random variable. The CDF for noncentral beta variable X can thus be simply defined in terms of the noncentral F CDF:

$$CDF_{nc\beta}(x, \alpha_1, \alpha_2, \lambda) = CDF_{ncF}(f, 2\alpha_1, 2\alpha_2, \lambda)$$

where $CDF_{nc\beta}(x, \alpha_1, \alpha_2, \lambda)$ is the noncentral beta CDF with x = x, α_1 = shape1, α_2 = shape2, and noncentrality parameter λ = lambda; $CDF_{ncF}(f, 2\alpha_1, 2\alpha_2, \lambda)$ is the noncentral F CDF with argument f, numerator and denominator degrees of freedom $2\alpha_1$ and $2\alpha_2$ respectively, and noncentrality parameter λ; and:

$$f = \frac{\alpha_2 x}{\alpha_1 (1 - x)}; \; x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2}$$

(See documentation for class Cdf method noncentralF for a discussion of how the noncentral F CDF is defined and calculated.)

Method noncentralBeta evaluates

$$x = CDF_{nc\beta}^{-1}(p, \alpha_1, \alpha_2, \lambda)$$

by first evaluating:

$$f = CDF_{ncF}^{-1}(p,\ 2\alpha_1,\ 2\alpha_2,\ \lambda)$$

and then solving for x using $x = \frac{\alpha_1 f}{\alpha_1 f + \alpha_2}$. (See documentation for class `Cdf` method `noncentralF` for a discussion of how the inverse noncentral F CDF is calculated.)

Parameters

`p` – a `double` scalar value representing the probability for which the inverse of the noncentral beta cumulative distribution function is to be evaluated. `p` must be non-negative and less than or equal to one.

`shape1` – a `double` scalar value representing the first shape parameter. `shape1` must be positive.

`shape2` – a `double` scalar value representing the second shape parameter. `shape2` must be positive.

`lambda` – a `double` scalar value representing the noncentrality parameter. `lambda` must be nonnegative.

Returns

a `double` scalar value representing the inverse of the noncentral beta distribution function evaluated at `p`. The probability that a noncentral beta random variable takes a value less than or equal to `noncentralBeta` is `p`.

noncentralF

```
static public double noncentralF(double p, double dfn, double dfd, double
lambda)
```

Description

Evaluates the inverse of the noncentral F cumulative distribution function (*CDF*).

If X is a noncentral chi-square random variable with noncentrality parameter λ and ν_1 degrees of freedom, and Y is a chi-square random variable with ν_2 degrees of freedom which is statistically independent of X, then

$$F = (X/\nu_1)/(Y/\nu_2)$$

is a noncentral F-distributed random variable whose CDF is given by:

$$CDF(f,\nu_1,\nu_2,\lambda) = \int_0^f PDF(x,\nu_1,\nu_2,\lambda)dx$$

where the probability density function $PDF(x,\nu_1,\nu_2,\lambda)$ is given by:

$$PDF(x,\nu_1,\nu_2,\lambda) = \Psi \sum_{k=0}^{\infty} \Phi_k$$

$$\Psi = \frac{e^{-\lambda/2}(\nu_1 x)^{\nu_1/2}(\nu_2)^{\nu_2/2}}{x\,(\nu_1 x + \nu_2)^{(\nu_1+\nu_2)/2}\,\Gamma(\nu_2/2)}$$

$$\Phi_k = \frac{R^k\,\Gamma(\frac{\nu_1+\nu_2}{2} + k)}{k!\,\Gamma(\frac{\nu_1}{2} + k)}$$

$$R = \frac{\lambda \nu_1 x}{2(\nu_1 x + \nu_2)}$$

where $\Gamma(\cdot)$ is the gamma function, ν_1 = `dfn`, ν_2 = `dfd`, λ = `lambda`, and $p = CDF(f, \nu_1, \nu_2, \lambda)$ is the probability that $F \le f$.

Method `noncentralF` evaluates

$$f = CDF^{-1}(p, \nu_1, \nu_2, \lambda)$$

Method `noncentralF` uses bisection and modified regula falsi search algorithms to invert the distribution function $CDF(f, \nu_1, \nu_2, \lambda)$, which is evaluated using method `noncentralF`. For sufficiently small p, an accurate approximation of $CDF^{-1}(p, \nu_1, \nu_2, \lambda)$ can be used which requires no such inverse search algorithms.

Parameters

`p` – a `double` scalar value representing the probability for which the inverse of the noncentral *F* cumulative distribution function is to be evaluated. `p` must be non-negative and less than one.

`dfn` – a `double` scalar value representing the number of numerator degrees of freedom. `dfn` must be positive.

`dfd` – a `double` scalar value representing the number of denominator degrees of freedom. `dfd` must be positive.

`lambda` – a `double` scalar value representing the noncentrality parameter. `lambda` must nonnegative.

Returns

a `double` scalar value representing the inverse of the noncentral *F* distribution function evaluated at *p*. The probability that a noncentral *F* random variable takes a value less than or equal to `noncentralF(p, dfn, dfd, lambda)` is `p`.

noncentralchi

```
static public double noncentralchi(double p, double df, double alam)
```

Description

Evaluates the inverse of the noncentral chi-squared cumulative probability distribution function.

Method `noncentralchi` evaluates the inverse distribution function of a noncentral chi-squared random variable with `df` degrees of freedom and noncentrality parameter `alam`, that is, with $P = p$, $\nu =$ df, and $\lambda =$ alam, it determines $c_0 =$ `noncentralchi(p, df, alam)`), such that

$$P = \sum_{i=0}^{\infty} \frac{e^{-\lambda/2} (\lambda/2)^i}{i!} \int_0^{c_0} \frac{x^{(\nu+2i)/2-1} e^{-x/2}}{2^{(\nu+2i)/2} \Gamma\left(\frac{\nu+2i}{2}\right)} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to c_0 is P.

Method `noncentralchi` uses bisection and modified regula falsi to invert the distribution function, which is evaluated using `noncentralchi`. See `noncentralchi` for an alternative definition of the noncentral chi-squared random variable in terms of normal random variables.

Parameters

p – a double scalar value representing the probability for which the inverse noncentral chi-squared distribution function is to be evaluated. p must be in the open interval (0.0, 1.0).

df – a double scalar value representing the number of degrees of freedom. This must be at least 0.5. but less than or equal to 200,000.

alam – a double scalar value representing the noncentrality parameter. This must be nonnegative, and alam + df must be less than or equal to 200,000.

Returns

a double scalar value. The probability that a noncentral chi-squared random variable takes a value less than or equal to this returned value is p.

noncentralstudentsT

```
static public double noncentralstudentsT(double p, int idf, double delta)
```

Description

Evaluates the inverse of the noncentral Student's t cumulative probability distribution function.

Method noncentralstudentsT evaluates the inverse distribution function of a noncentral t random variable with idf degrees of freedom and noncentrality parameter delta; that is, with $P = p$, $\nu = idf$, $\delta = delta$, it determines $t_0 =$ noncentralstudentsT(p, idf, delta)), such that

$$P = \int_{-\infty}^{t_0} \frac{\nu^{\nu/2} e^{-\delta^2/2}}{\sqrt{\pi}\Gamma(\nu/2)(\nu+x^2)^{(\nu+1)/2}} \sum_{i=0}^{\infty} \Gamma((\nu+i+1)/2) \left(\frac{\delta^i}{i!}\right) \left(\frac{2x^2}{\nu+x^2}\right)^{i/2} dx$$

where $\Gamma(\cdot)$ is the gamma function. The probability that the random variable takes a value less than or equal to t_0 is P. See noncentralstudentsT for an alternative definition in terms of normal and chi-squared random variables. The method noncentralstudentsT uses bisection and modified regula falsi to invert the distribution function, which is evaluated using noncentralstudentsT.

Parameters

p – a double scalar value representing the probability for which the function is to be evaluated.

idf – an int scalar value representing the number of degrees of freedom. This must be positive.

delta – a double scalar value representing the noncentrality parameter.

Returns

a double scalar value. The probability that a noncentral Student's t random variable takes a value less than or equal to this returned value is p.

normal

```
static public double normal(double p)
```

Description

Evaluates the inverse of the normal (Gaussian) cumulative probability distribution function.

Method `normal` evaluates the inverse of the distribution function, Φ, of a standard normal (Gaussian) random variable, that is, `normal` (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.

Parameter

`p` – a `double` scalar value representing the probability at which the function is to be evaluated.

Returns

a `double` scalar value. The probability that a standard normal random variable takes a value less than or equal to this returned value is `p`.

studentsT

```
static public double studentsT(double p, double df)
```

Description

Returns the inverse of the Student's t cumulative probability distribution function.

`studentsT` 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 `beta` 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. The probability that a Student's t random variable takes a value less than or equal to this returned value is `p`.

uniform

```
static public double uniform(double p, double aa, double bb)
```

Description

Returns the inverse of the uniform cumulative probability distribution function.

Parameters

p – a double scalar value representing the probability for which the inverse uniform function is to be evaluated.

aa – a double scalar value representing the minimum value.

bb – a double scalar value representing the maximum value.

Returns

a double scalar value. The probability that a uniform random variable takes a value less than or equal to this returned value is p.

Example: The Inverse Cumulative Distribution Functions

Various inverse cumulative distribution functions are exercised. Their use in this example typifies the manner in which other functions in the InvCdf class would be used.

```
import com.imsl.stat.InvCdf;

public class InvCdfEx1 {

    public static void main(String args[]) {

        // Inverse Beta
        double x = .5;
        double pin = 12.;
        double qin = 12.;
        double result = InvCdf.beta(x, pin, qin);
        System.out.println("InvCdf.beta(.5, 12., 12.) is " + result);

        // Inverse Chi
        double prob = .99;
        int n = 2;
        result = InvCdf.chi(prob, n);
        System.out.println("InvCdf.chi(.99, 2) is " + result);
    }
}
```

Output

```
InvCdf.beta(.5, 12., 12.) is 0.4999999999999991
InvCdf.chi(.99, 2) is 9.210340371976173
```

CdfFunction interface

```
public interface com.imsl.stat.CdfFunction
```

Public interface for the user-supplied cumulative distribution function to be used by InverseCdf and ChiSquaredTest.

Method

cdf

`public double cdf(double p)`

Description

Public interface for the user-supplied cumulative distribution function to be used by InverseCdf.

Parameter

`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.

InverseCdf class

`public class com.imsl.stat.InverseCdf implements Serializable`

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.

Constructor

InverseCdf

`public InverseCdf(CdfFunction cdf)`

Description

Constructor for the inverse of a user-supplied cummulative distribution function.

Parameter

`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 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.

Parameter

`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 Exception {
```

```
        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
```

InverseCdf.DidNotConvergeException class

`static public class com.imsl.stat.InverseCdf.DidNotConvergeException extends com.imsl.IMSLException`

The iteration did not converge

Constructors

InverseCdf.DidNotConvergeException

`public InverseCdf.DidNotConvergeException(String message)`

Description

Constructs a `DidNotConvergeException` object.

Parameter

`message` – a `String` containing the error message

InverseCdf.DidNotConvergeException

`public InverseCdf.DidNotConvergeException(String key, Object[] arguments)`

Description

Constructs a `DidNotConvergeException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

Distribution interface

`public interface com.imsl.stat.Distribution`

Public interface for the user-supplied distribution function.

The purpose of this interface is to fit the probability distribution to a given set of data and return the probability density at each value of the given set of data.

Method

eval

`public double[] eval(double[] xData)`

Description

Evaluation method to fit the user-supplied probability density function to input data

Parameter

`xData` – a `double` array representing the points at which the probability density function is to be evaluated.

Returns

a `double` array representing the probability density at each value of `xData`

ProbabilityDistribution interface

`public interface com.imsl.stat.ProbabilityDistribution implements com.imsl.stat.Distribution, Serializable`

Public interface for a user-supplied probability distribution.

The purpose of this interface is to evaluate the probability density of a given set of data by either fitting the probability density function to the data or by evaluating the probability density function with supplied parameters. Both `eval` methods return the probability density at each value of the given set of data. After the probability distribution is fitted to the data, the `GetParameters` method can be used to return the distribution parameters used to fit the probability density function to the data.

The `DataMining` package class `NaiveBayesClassifier` uses an implementation of `ProbabilityDistribution` to train continuous data.

Methods

eval

```
public double eval(double xData, Object[] parameters)
```

Description

Evaluation method for the user-supplied distribution function and parameters. Evaluates the user-supplied probability density at xData using the supplied probability distribution parameters.

Parameters

xData – a double scalar value containing the point the distribution function is to evaluate

parameters – an Object array containing the probability distribution parameters to be used in evaluating xData. See method getParameters.

Returns

a double scalar value representing the probability density at xData

eval

```
public double[] eval(double[] xData, Object[] parameters)
```

Description

Evaluates the user-supplied probability density of each value in xData using the supplied probability distribution parameters.

Parameters

xData – a double array containing the points at which the probability density function is to be evaluated

parameters – an Object array containing the probability distribution parameters to be used in evaluating xData, see method getParameters

Returns

a double array representing the probability density of each value in xData

getParameters

```
public Object[] getParameters()
```

Description

Returns the current parameters of the probability density function.

Returns

an Object array containing the parameters resulting from the last invocation of the (Distribution) eval method with the following signature, double[] eval(double[] xData). This Object array can be used as input to the eval methods that require an Object array of distribution parameters as input.

NormalDistribution class

```
public class com.imsl.stat.NormalDistribution implements
com.imsl.stat.ProbabilityDistribution, Serializable
```

Evaluates the normal (Gaussian) probability density for a given set of data.

`NormalDistribution` evaluates the normal probability density of a given set of data, `xData`. If parameters are not supplied, the `eval` method fits the normal probability density function to the data by first calculating the mean and standard deviation of `xData`. The normal probability density function is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

where μ and σ are the mean and standard deviation.

The `DataMining` package class `NaiveBayesClassifier` uses `NormalDistribution` as the default method to train continuous data.

Constructor

NormalDistribution

```
public NormalDistribution()
```

Methods

eval

```
public double[] eval(double[] xData)
```

Description

Fits a normal (Gaussian) probability distribution to `xData` and returns the probability density at each value.

Parameter

`xData` – a `double` array representing the points at which the normal probability distribution function is to be evaluated

Returns

a `double` array representing the normal probability density at each value in `xData`

eval

```
public double eval(double xData, Object[] parameters)
```

Description

Evaluates a normal (Gaussian) probability density at a given point `xData`.

Parameters

`xData` – a `double` containing the point at which the normal probability density function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the normal probability density, see method `getParameters`

Returns

a `double` representing the normal probability density at `xData`

eval

```
public double[] eval(double[] xData, Object[] parameters)
```

Description

Evaluates a normal (Gaussian) probability distribution with the given parameters at each point in `xData` and returns the probability density at each value.

Parameters

`xData` – a `double` array representing the points at which the normal probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the normal probability density function, see method `getParameters`

Returns

a `double` array representing the normal probability density of each value in `xData`

getMean

```
public double getMean()
```

Description

Returns the population mean of `xData`.

Returns

a `double` representing the population mean of `xData`

getParameters

```
public Object[] getParameters()
```

Description

Returns the current parameters of the normal probability density function.

Returns

an `Object` array containing the parameters resulting from the last invocation of the (`Distribution`) `eval` method with the following signature, `double[] eval(double[] xData)`. This `Object` array can be used as input to the `eval` methods that require an `Object` array of distribution parameters as input.

getStandardDeviation

```
public double getStandardDeviation()
```

Description

Returns the population standard deviation.

Returns

a `double` representing the population standard devation of `xData`

GammaDistribution class

```
public class com.imsl.stat.GammaDistribution implements
com.imsl.stat.ProbabilityDistribution, Serializable
```

Evaluates a gamma probability density for a given set of data.

`GammaDistribution` evaluates the gamma density of a given set of data, `xData`. If parameters are not supplied, the `Eval` method fits the gamma probability density function to the data by first calculating the shape and scale parameters using an MLE technique for a best fit. The gamma probability density function is defined as:

$$f(x) = x^{a-1}\frac{e^{-\frac{x}{b}}}{b^a\Gamma(a)},\ x > 0,\ a > 0 \text{ and } b > 0\ ,$$

where a and b are the scale and shape parameters.

The `DataMining` package class `NaiveBayesClassifier` uses `GammaDistribution` as a method to train continuous data.

Constructor

GammaDistribution

```
public GammaDistribution()
```

Methods

eval

`public double[] eval(double[] xData)`

Description

Fits a gamma probability distribution to `xData` and returns the probability density at each value.

Parameter

`xData` – a `double` array representing the points at which the gamma probability distribution function is to be evaluated

Returns

a `double` array representing the gamma probability density at each value of `xData`

eval

`public double eval(double xData, Object[] parameters)`

Description

Evaluates a gamma probability density at a given point `xData`.

Parameters

`xData` – a `double` representing the point at which the gamma probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the gamma distribution, see method `getParameters`

Returns

a `double` representing the gamma probability density at `xData`

eval

`public double[] eval(double[] xData, Object[] parameters)`

Description

Evaluates a gamma probability distribution with a given set of parameters at each point in `xData` and returns the probability density at each value.

Parameters

`xData` – a `double` array representing the points at which the gamma probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the gamma distribution, see method `getParameters`

Returns

a `double` array representing the gamma probability density at each value of `xData`

getParameters

`public Object[] getParameters()`

Description

Returns the current parameters of the gamma probability density function.

Returns

an `Object` array containing the parameters resulting from the last invocation of the (`Distribution`) `eval` method with the following signature, `double[] eval(double[] xData)`. This `Object` array can be used as input to the `eval` methods that require an `Object` array of distribution parameters as input.

getScaleParameter

```
public double getScaleParameter()
```

Description

Returns the maximum-likelihood estimate found for the gamma scale parameter.

Returns

a `double` representing the maximum-likelihood estimate found for the gamma scale parameter

getShapeParameter

```
public double getShapeParameter()
```

Description

Returns the maximum-likelihood estimate found for the gamma shape parameter.

Returns

a `double` representing the maximum-likelihood estimate found for the gamma shape parameter

LogNormalDistribution class

```
public class com.imsl.stat.LogNormalDistribution implements
com.imsl.stat.ProbabilityDistribution, Serializable
```

Evaluates a lognormal probability density for a given set of data.

`LogNormalDistribution` evaluates the lognormal probability density of a given set of data, `xData`. If parameters are not supplied, the `eval` method fits the lognormal probability density function to the data by first calculating the mean and standard deviation. The lognormal probability density function is defined as:

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}}$$

where μ and σ are the mean and standard deviation.

The `DataMining` package class `NaiveBayesClassifier` uses `LognormalDistribution` as a method to train continuous data.

Constructor

LogNormalDistribution

`public LogNormalDistribution()`

Methods

eval

`public double[] eval(double[] xData)`

Description

Fits a lognormal probability distribution to `xData` and returns the probability density at each value.

Parameter

`xData` – a `double` array representing the points at which the lognormal probability distribution function is to be evaluated

Returns

a `double` array representing the lognormal probability density at each value of `xData`

eval

`public double eval(double xData, Object[] parameters)`

Description

Evaluates a lognormal probability density function at a given point `xData`.

Parameters

`xData` – a `double` representing the point at which the lognormal probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the lognormal distribution, see method `getParameters`

Returns

a `double` representing the lognormal probability density at `xData`

eval

`public double[] eval(double[] xData, Object[] parameters)`

Description

Evaluates a lognormal probability distribution with a given set of parameters at each point in `xData` and returns the probability density at each value.

Parameters

`xData` – a `double` array representing the points at which the lognormal probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the lognormal distribution, see method `getParameters`

Returns

a `double` array representing the lognormal probability density at each value of `xData`

getMean

```
public double getMean()
```

Description

Returns the lognormal probability distribution mean parameter.

Returns

a `double` representing the mean parameter

getParameters

```
public Object[] getParameters()
```

Description

Returns the current parameters of the lognormal probability density function

Returns

an `Object` array containing the parameters resulting from the last invocation of the (`Distribution`) `eval` method with the following signature, `double[] eval(double[] xData)`. This `Object` array can be used as input to the `eval` methods that require an `Object` array of distribution parameters as input.

getStandardDeviation

```
public double getStandardDeviation()
```

Description

Returns the lognormal probability distribution standard deviation parameter.

Returns

a `double` representing the standard deviation parameter

PoissonDistribution class

```
public class com.imsl.stat.PoissonDistribution implements
com.imsl.stat.ProbabilityDistribution, Serializable
```

Evaluates a Poisson probability density of a given set of data.

The `poissonDistribution` evaluates the Poisson probability density of a given set of data, `xData`. If parameters are not supplied, the `eval` method fits the Poisson probability density function by first calculating *theta*, θ. The Poisson probability density function is defined as:

$$f(x) = \frac{\theta^x e^{-\theta}}{x!}, x \geq 0 \text{ and } \theta > 0.$$

The `DataMining` package class `NaiveBayesClassifier` uses `PoissonDistribution` as a method to train continuous data.

Constructor

PoissonDistribution

```
public PoissonDistribution()
```

Methods

eval

```
public double[] eval(double[] xData)
```

Description

Fits a Poisson probability distribution to `xData` and returns the probability density at each value.

Parameter

`xData` – a `double` array representing the points at which the Poisson probability distribution function is to be evaluated

Returns

a `double` array representing the Poisson probability density at each value of `xData`

eval

```
public double eval(double xData, Object[] parameters)
```

Description

Evaluates a Poisson probability density function at a given point `xData`.

Parameters

`xData` – a `double` representing the point at which the Poisson probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the Poisson distribution, see method `getParameters`

Returns

a `double` representing the Poisson probability density at `xData`

eval

`public double[] eval(double[] xData, Object[] parameters)`

Description

Evaluates a Poisson probability distribution with a given set of parameters at each point in `xData` and returns the probability density at each value.

Parameters

`xData` – a `double` array representing the points at which the Poisson probability distribution function is to be evaluated

`parameters` – an `Object` array representing the parameters used to evaluate the Poisson distribution, see method getParameters.

Returns

a `double` array representing the Poisson probability density at each value of `xData`

getParameters

`public Object[] getParameters()`

Description

Returns the current parameters of the Poisson probability density function.

Returns

an `Object` array containing the parameters resulting from the last invocation of the (`Distribution`) `eval` method with the following signature, `double[] eval(double[] xData)`. This `Object` array can be used as input to the `eval` methods that require an `Object` array of distribution parameters as input.

getTheta

`public double getTheta()`

Description

Returns the mean number of successes in a given time period of the Poisson probability distribution.

Returns

a `double` representing the mean number of successes in a given time period of the Poisson probability distribution

Chapter 23: Random Number Generation

Types

class Random 1324
class FaureSequence 1349
class MersenneTwister 1353
class MersenneTwister64 1357
interface RandomSequence 1361
class RandomSamples 1362

Usage Notes

Overview of Random Number Generation

This chapter describes functions for the generation of random numbers that are useful for applications in simulation studies.

In the following discussions, the phrases *random numbers*, *random deviates*, *deviates*, and *variates* are used interchangeably. The phrase *pseudorandom* is sometimes used to emphasize that the numbers generated are really not *random* since they result from a deterministic process. The usefulness of pseudorandom numbers is derived from the similarity, in a statistical sense, of samples of the pseudorandom numbers to samples of observations from the specified distributions. In short, while the pseudorandom numbers are completely deterministic and repeatable, they simulate the realizations of independent and identically distributed random variables.

Class `com.imsl.stat.Random` extends `java.util.Random`. It adds the ability to generate random numbers with a number of different non-uniform distributions. The non-uniform random number generators use a uniform random number generator. The uniform random number generator in `java.util.Random` can be used. `com.imsl.stat.Random` adds a linear congruential generator. It is also possible to use another uniform generator as long as it implements the `Random.BaseGenerator` interface.

The `MersenneTwister` and `MersenneTwister64` uniform random number generators generate random number series with very long periods. They both implement the `Random.BaseGenerator` interface and so can be used as the base uniform generator in `com.imsl.stat.Random`.

The class `FaureSequence` generates the *low-discrepancy* Faure sequence. This is also called a *quasi-random* generator. The sequence is a series of points in *n*-dimensions, which are close to being as equally spaced as possible. Low-discrepancy refers to the difference from actually being as equally spaced as possible.

The `FaureSequence` implements the `RandomSequence` interface. This interface defines a sequence of points in *n*-dimensions. It is used by the class `com.imsl.math.HyperRectangleQuadrature` to evaluate *n*-dimensional integrals.

Random class

```
public class com.imsl.stat.Random extends java.util.Random implements
Serializable, Cloneable
```

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).

Alternatively, one can select a 32-bit or 64-bit Mersenne Twister generator by first instantiating `com.imsl.stat.MersenneTwister` (p. 1353) or `com.imsl.stat.MersenneTwister64` (p. 1357) . These generators have a period of $2^{19937}-1$ and a 623-dimensional equidistribution property. See Matsumoto et al. 1998 for details.

The generation of uniform (0,1) numbers is done by the method nextDouble.

Nonuniform random numbers are generated using a variety of transformation procedures. All of the transformations used are exact (mathematically). The most straightforward transformation is the *invers e CDF technique*, but it is often less efficient than others involving *acceptance/rejection* and mixtures. See Kennedy and Gentle(1980) for discussion of these and other techniques.Many of the nonuniform generators use different algorithms depending on the values of the parameters of the distributions. This is particularly true of the generators for discrete distributions. Schmeiser (1983) gives an overview of techniques for generating deviates from discrete distributions.

Extensive empirical tests of some of the uniform random number generators available in the `Random` class are reported by Fishman and Moore (1982 and 1986). Results of tests on the generator using the multiplier 16807 are reported by Learmonth and Lewis (1973). If the user wishes to perform additional tests, the routines in Chapter 17, *Tests of Goodness of Fit*, may be of use. Often in Monte Carlo applications, it is appropriate to construct an ad hoc test that is sensitive to departures that are important in the given application. For example, in using Monte Carlo methods to evaluate a one-dimensional integral, autocorrelations of order one may not be harmful, but they may be disastrous in evaluating a two-dimensional integral. Although generally the routines in this chapter for generating random deviates from nonuniform distributions use exact methods, and, hence, their quality depends almost solely on the quality of the underlying uniform generator, it is often advisable to employ an ad hoc test of goodness of fit for the transformations that are to be applied to the deviates from the nonuniform generator.

Three methods are associated with copulas. A *copula* is a multivariate cumulative probability distribution (CDF) whose arguments are random variables uniformly distributed on the interval [0,1] corresponding to the probabilities (variates) associated with arbitrarily distributed marginal deviates. The copula structure allows the multivariate CDF to be partitioned into the copula, which has associated with it information characterizing the dependence among the marginal variables, and the set of separate marginal deviates, each of which has its own distribution structure.

Two methods, `nextGaussianCopula` and `nextStudentsTCopula`, allow the user to specify a correlation structure (in the form of a Cholesky matrix) which can be used to imprint correlation information on a sequence of multivariate random vectors. Each call to one of these methods returns a random vector whose elements (variates) are each uniformly distributed on the interval [0,1] and correlated according to a user-specified Cholesky matrix. These variate vector sequences may then be inverted to marginal deviate sequences whose distributions and imprinted correlations are user-specified.

Method `nextGaussianCopula` generates a random Gaussian copula sequence by inverting uniform [0,1] random numbers to N(0,1) deviates vectors, imprinting each vector with the correlation information by multiplying it with the Cholesky matrix, and then using the N(0,1) CDF to map the imprinted deviates back to uniform [0,1] variates.

Method `nextStudentsTCopula` inverts a vector of uniform [0, 1] random numbers to a N(0,1) deviate vector, imprints the vector with correlation information by multiplying it with the Cholesky matrix, transforms the imprinted N(0,1) vector to an imprinted Student's t vector (where each element is Student's t distributed with ν degrees of freedom) by dividing each element of the imprinted N(0,1) vector by $\sqrt{\frac{s}{\nu}}$, where s is a random deviate taken from a chi-squared distribution with ν degrees of freedom, and finally maps each element of the resulting imprinted Student's t vector back to a uniform [0, 1] distributed variate using the Student's t CDF.

The third copula method, `canonicalCorrelation`, extracts a correlation matrix from a sequence of multivariate deviate vectors whose component marginals are arbitrarily distributed. This is accomplished by first extracting the empirical CDF from each of the marginal deviates and then using this CDF to map the deviates to uniform [0,1] variates which are then inverted to Normal (0,1) deviates. Each element C_{ij} of the correlation matrix can then be extracted by averaging the products $z_{it}z_{jt}$ of deviates i and j over the t-indexed sequence. The utility of method `canonicalCorrelation` is that because the correlation matrix is derived from N(0,1) deviates, the correlation is unbiased, i.e. undistorted by the arbitrary marginal distribution structures of the original deviate vector sequences. This is important in such financial applications as portfolio optimization, where correlation is used to estimate and minimize risk.

The use of these routines is illustrated with RandomEx2.java, which first uses method `nextGaussianCopula` to create a correlation imprinted sequence of random deviate vectors and then uses method `canonicalCorrelation` to extract the correlation matrix from the imprinted sequence of vectors.

Constructors

Random

`public Random()`

Description

Constructor for the Random number generator class.

Random

`public Random(Random.BaseGenerator baseGenerator)`

Description

Constructor for the Random number generator class with an alternate basic number generator.

Parameter

`baseGenerator` – is used to override the method `next`.

Random

`public Random(long seed)`

Description

Constructor for the Random number generator class with supplied seed.

Parameter

`seed` – a `long` which represents the random number generator seed in the range of -2,147,483,647 to +2,147,483,647

Methods

canonicalCorrelation

`public double[][] canonicalCorrelation(double[][] deviate)`

Description

Method `canonicalCorrelation` generates a canonical correlation matrix from an arbitrarily distributed multivariate deviate sequence with `nvar` deviate variables, `nseq` steps in the sequence, and a Gaussian Copula dependence structure.

Method `canonicalCorrelation` first maps each of the `j=1..nvar` input deviate sequences `deviate[k=1..nseq][j]` into a corresponding sequence of variates, say `variate[k][j]` (where

variates are values of the empirical cumulative probability function, $CDF(x)$, defined as the probability that random deviate variable $X \leq x$, and where `nseq = deviate.length` and `nvar = deviate[0].length`). The variate matrix `variate[k][j]` is then mapped into Normal(0,1) distributed deviates z_{kj} using the method `Cdf.inverseNormal(variate[k][j])` and then the standard covariance estimator

$$C_{ij} = \frac{1}{n_{seq}} \sum_{k=1}^{n_{seq}} z_{ki}\, z_{kj}$$

is used to calculate the canonical correlation matrix `correlation = canonicalCorrelation(deviate)`, where C_{ij} = `correlation[i][j]` and n_{seq} = `nseq`.

If a multivariate distribution has Gaussian marginal distributions, then the standard "empirical" correlation matrix given above is "unbiased", i.e. an accurate measure of dependence among the variables. But when the marginal distributions depart significantly from Gaussian, i.e. are skewed or flattened, then the empirical correlation may become biased. One way to remove such bias from dependence measures is to map the non-Gaussian-distributed marginal deviates to Gaussian N(0,1) deviates (by mapping the non-Gaussian marginal deviates to empirically derived marginal CDF variate values, then inverting the variates to N(0,1) deviates as described above), and calculating the standard empirical correlation matrix from these N(0,1) deviates as in the equation above. The resulting "(Gaussian) canonical correlation" matrix thereby avoids the bias that would occur if the empirical correlation matrix were extracted from the non-Gaussian marginal distributions directly.

The canonical correlation matrix may be of value in such applications as Markowitz porfolio optimization, where an unbiased measure of dependence is required to evaluate portfolio risk, defined in terms of the portfolio variance which is in turn defined in terms of the correlation among the component portfolio instruments.

The utility of the canonical correlation derives from the observation that a "copula" multivariate distribution with uniformly-distributed deviates (corresponding to the CDF probabilities associated with the marginal deviates) may be mapped to arbitrarily distributed marginals, so that an unbiased dependence estimator derived from one set of marginals (N(0,1) distributed marginals) can be used to represent the dependence associated with arbitrarily-distributed marginals. The "Gaussian Copula" (whose variate arguments are derived from N(0,1) marginal deviates) is a particularly useful structure for representing multivariate dependence.

This is demonstrated in Example 2 where method `Random.nextGaussianCopula(CholeskyMtrx)` (where `CholeskyMtrx` is a Cholesky object derived from a user-specified covariance matrix) is used to imprint correlation information on otherwise arbitrarily distributed and independent random sequences. Method `Random.canonicalCorrelation` is then used to extract an unbiased correlation matrix from these imprinted deviate sequences.

Parameter

`deviate` – is the double `nseq` by `nvar` array of input deviate values.

next

`protected int next(int bits)`

Description

Generates the next pseudorandom number. If an alternate base generator was set in the constructor, its `next` method is used. If the `multiplier` is set then the multiplicative congruential method is used. Otherwise, `super.next(bits)` is used.

Parameter

`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 ε, where ε 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.

Parameter

`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.

nextDiscrete

```
public int nextDiscrete(int imin, double[] probabilities)
```

Description

Generate a pseudorandom number from a general discrete distribution using an alias method.

Method `nextDiscrete` generates a pseudorandom number from a discrete distribution with probability function given in the vector `probabilities`; that is

$$\Pr(X = i) = p_j$$

for $i = i_{min}, i_{min}+1, \ldots, i_{min}+n_m-1$, where $j = i - i_{min} + 1$, $p_j =$ `probabilities[j-1]`, $i_{min} =$ `imin`,$n_m =$ `nmass` and `probabilities.length` is the number of mass points.

The algorithm is the alias method, due to Walker (1974), with modifications suggested by Kronmal and Peterson (1979). On the first call with a set of probabilities, the method performs an initial setup after which the number generation phase is very fast. To increase efficiency, the code skips the setup phase on subsequent calls with the same inputs.

Parameters

`imin` – an `int` which specifies the smallest value the random deviate can assume. This is the value corresponding to the probability in `probabilities[0]`.

`probabilities` – a `double` array containing the probabilities associated with the individual mass points. The elements of `probabilities` must be nonnegative and must sum to 1.0. The length of `probabilities` muse be greater than 1.

Returns

an `int` which contains the random discrete deviate.

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 θ can be generated by using `nextExponential` and then multiplying the result by θ.

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

$$f(x) = \frac{p}{\theta} e^{-x/\theta_1} + \frac{1-p}{\theta_2} e^{-x/\theta_2} \ for x > 0$$

where $p = $ 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 θ_1, and with probability *1 - p* generates an exponential with mean θ_2. When p is greater than 1, but less than $\theta_1/(\theta_1 - \theta_2)$, then either an exponential deviate with mean θ_2 or the sum of two exponentials with means θ_1 and θ_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 $0 \leq p \leq \text{theta1}/(\text{theta1} - \text{theta2})$.

Returns

a double, a pseudorandom number from a mixture of the two exponential distributions.

nextExtremeValue

```
public double nextExtremeValue(double mu, double beta)
```

Description

Generate a pseudorandom number from an extreme value distribution.

Parameters

mu – a double scalar value representing the location parameter.

beta – a double scalar value representing the scale parameter.

Returns

a double pseudorandom number from an extreme value distribution

nextF

```
public double nextF(double dfn, double dfd)
```

Description

Generate a pseudorandom number from the F distribution.

Parameters

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, a pseudorandom number from an F distribution

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.

Parameter

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

nextGaussianCopula

```
public double[] nextGaussianCopula(Cholesky chol)
```

Description

Generate pseudorandom numbers from a Gaussian Copula distribution.

`nextGaussianCopula` generates pseudorandom numbers from a multivariate Gaussian Copula distribution which are uniformly distributed on the interval (0,1) representing the probabilities associated with N(0,1) deviates imprinted with correlation information from input Cholesky object `chol`. Cholesky matrix `R` is defined as the "square root" of a user-defined correlation matrix, that is `R` is a lower triangular matrix such that `R` times the transpose of `R` is the correlation matrix. First, a length k vector of independent random normal deviates with mean 0 and variance 1 is generated, and then this deviate vector is pre-multiplied by Cholesky matrix `R`. Finally, the Cholesky-imprinted random N(0,1) deviates are mapped to output probabilities using the N(0,1) cumulative distribution function (CDF).

Random deviates from arbitrary marginal distributions which are imprinted with the correlation information contained in Cholesky matrix `R` can then be generated by inverting the output probabilities using user-specified inverse CDF functions.

Parameter

`chol` – is the `Cholesky` object containing the Cholesky factorization of the correlation matrix of order k.

Returns

a `double` array which contains the pseudorandom numbers from a multivariate Gaussian Copula 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.

Parameter

`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.

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

nextLogarithmic

```
public int nextLogarithmic(double a)
```

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

$$f(x) = -\frac{a^x}{x\ln(1-a)}$$

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.

Parameter

`a` – a `double` which specifies the parameter of the logarithmic distribution, $0 < a < 1.0$.

Returns

an `int`, a pseudorandom number from a logarithmic distribution.

nextMultivariateNormal

```
public double[] nextMultivariateNormal(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 a lower triangular matrix such that `matrix` times the transpose of `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 pre-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.

Parameter

`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

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 θ = 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.

Parameter

`theta` – a `double` which specifies the mean of the Poisson distribution, theta > 0

Returns

an `int`, a pseudorandom number from a Poisson distribution

nextRayleigh

`public double nextRayleigh(double alpha)`

Description

Generate a pseudorandom number from a Rayleigh distribution.

Method `nextRayleigh` generates pseudorandom numbers from a Rayleigh distribution with scale parameter *alpha*.

Parameter

`alpha` – a `double` which specifies the scale parameter of the Rayleigh distribution

Returns

a `double`, a pseudorandom number from a Rayleigh 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`.

Parameter

`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

nextStudentsTCopula

`public double[] nextStudentsTCopula(double df, Cholesky chol)`

Description

Generate pseudorandom numbers from a Student's t Copula distribution.

`nextStudentsTCopula` generates pseudorandom numbers from a multivariate Student's *t* Copula distribution which are uniformly distributed on the interval (0,1) representing the probabilities associated with Student's *t* deviates with `df` degrees of freedom imprinted with correlation information from the input Cholesky object `chol`. Cholesky matrix `R` is defined as the "square root" of a user-defined correlation matrix, i.e. `R` is a lower triangular matrix such that `R` times the transpose of `R` is the correlation matrix. First, a length k vector of independent random normal deviates with mean 0 and variance 1 is generated, and then this deviate vector is pre-multiplied by Cholesky matrix `R`. Each of the k elements of the resulting vector of Cholesky-imprinted random deviates is then divided by $\sqrt{\frac{s}{\nu}}$, where ν = `df` and s is a random deviate taken from a chi-squared distribution with `df` degrees of freedom. Each element of the Cholesky-imprinted N(0,1) vector is a linear combination of normally distributed random numbers and is therefore itself normal, and the division of each element by $\sqrt{\frac{s}{\nu}}$ therefore insures that each element of the resulting vector is Student's *t* distributed. Finally each element of the Cholesky-imprinted Student's *t* vector is mapped to an output probability using the Student's *t* cumulative distribution function (CDF) with `df` degrees of freedom.

Random deviates from arbitrary marginal distributions which are imprinted with the correlation information contained in Cholesky matrix `R` can then be generated by inverting the output probabilities using user-specified inverse CDF functions.

Parameters

`df` – a `double` which specifies the degrees of freedom parameter.

`chol` – the `Cholesky` object containing the Cholesky factorization of the correlation matrix of order k.

Returns

a `double` array which contains the pseudorandom numbers from a multivariate Students t Copula distribution with `df` degrees of freedom.

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 $0 \leq x \leq .5$, and $f(x) = 4(1-x)$, for $.5 < x \leq 1$. `nextTriangular` uses an inverse CDF technique.

Returns

a `double`, a pseudorandom number from a triangular distribution on the interval (0,1)

nextUniformDiscrete

```
public int nextUniformDiscrete(int k)
```

Description

Generate a pseudorandom number from a discrete uniform distribution.

`nextUniformDiscrete` generates pseudorandom numbers from a discrete uniform distribution with parameter k. The integers $i = 1, \ldots, k$ occur with equal probability. A random integer is generated by multiplying k by a uniform (0,1) random number, adding 1.0, and truncating the result to an integer. This, of course, is equivalent to sampling with replacement from a finite population of size k.

Parameter

`k` – Parameter of the discrete uniform distribution. The integers $1, \ldots, k$ occur with equal probability. Parameter k must be positive.

Returns

an `int`, a pseudorandom number from a discrete uniform distribution.

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

$$f(x) = \frac{1}{2\pi I_0(c)} \exp\left[c \cos(x)\right] \, for - \pi < x < \pi$$

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).

Parameter

`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

$$f(x) = Ax^{A-1}e^{-x^A} \, for x \geq 0$$

`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.

Parameter

`a` – a `double` which specifies the shape parameter of the Weibull distribution, a > 0

Returns

a `double`, a pseudorandom number from a Weibull distribution

nextZigguratNormalAR

```
public double nextZigguratNormalAR()
```

Description

Generates pseudorandom numbers using the Ziggurat method.

The `nextZigguratNormalAR` method cuts the density into many small pieces. For each random number generated, an interval is chosen at random and a random normal is generated from the choosen interval. In this implementation, the density is cut into 256 pieces, but symmetry is used so that only 128 pieces are needed by the computation. Following Doornik (2005), different uniform random deviates are used to determine which slice to use and to determine the normal deviate from the slice.

Returns

a `double` containing the random normal deviate.

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.

Parameter

`multiplier` – an `int` which represents the random number generator multiplier

setSeed

`public void setSeed(long seed)`

Description

Sets the seed.

Parameter

`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.

Parameter

`n` – is the number of random deviates to skip.

Example 1: 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 Exception {
        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 0.5518855965158241
```

Example 2: Using Copulas to imprint and extract correlation information

This example uses method `Random.nextGaussianCopula` to generate a multivariate sequence `GCdevt[k=0..nseq-1][j=0..nvar-1]` whose marginal distributions are user-defined and imprinted with a user-specified correlation matrix `CorrMtrxIn[i=0..nvar-1][j=0..nvar-1]` and then uses method `Random.canonicalCorrelation` to extract from this multivariate random sequence a canonical correlation matrix `CorrMtrx[i=0..nvar-1][j=0..nvar-1]`.

This example illustrates two useful copula related procedures. The first procedure generates a random multivariate sequence with arbitrary user-defined marginal deviates whose dependence is specified by a user-defined correlation matrix. The second procedue is the inverse of the first: an arbitary multivariate deviate input sequence is first mapped to a corresponding sequence of empirically derived variates, i.e. cumulative distribution function values representing the probability that each random variable has a value less than or equal to the input deviate. The variates are then inverted, using the inverse Normal(0,1) function, to N(0,1) deviates; and finally, a canonical covariance matrix is extracted from the multivariate N(0,1) sequence using the standard sum of products.

This example demonstrates that the `nextGaussianCopula` method correctly imbeds the user-defined correlation information into an arbitrary marginal distribution sequence by extracting the canonical correlation from these sequences and showing that they differ from the original correlation matrix by a small relative error, which generally decreases as the number of multivariate sequence vectors increases.

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

public class RandomEx2 {

    static Random IMSLRandom() {
        Random r = new Random();
        r.setSeed(123457);
        r.setMultiplier(16807);
        return r;
    }

    public static void main(String args[]) throws com.imsl.IMSLException {
        double CorrMtrxIn[][] = {
            {1., -0.9486832980505138, 0.8164965809277261},
            {-0.9486832980505138, 1., -0.6454972243679028},
            {0.8164965809277261, -0.6454972243679028, 1.}
        };
```

```
int nvar = 3;

System.out.println("Random Example 2:");
System.out.println("");

for (int i = 0; i < nvar; i++) {
    for (int j = 0; j < i; j++) {
        System.out.println("CorrMtrxIn[" + i + "]["
                + j + "] = " + CorrMtrxIn[i][j]);
    }
}

PrintMatrixFormat pmf = new PrintMatrixFormat();
pmf.setNumberFormat(new java.text.DecimalFormat("0.000000000"));

new PrintMatrix("Input Correlation Matrix: ").print(pmf, CorrMtrxIn);
System.out.println("Correlation Matrices calculated from");
System.out.println(" Gaussian Copula imprinted multivariate sequence:");
System.out.println("");

// Compute the Cholesky factorization of CorrMtrxIn
Cholesky CholMtrx = new Cholesky(CorrMtrxIn);

for (int kmax = 500; kmax < 1000000; kmax *= 10) {

    System.out.println("# vectors in multivariate sequence: " + kmax);

    double GCvart[][] = new double[kmax][];
    double GCdevt[][] = new double[kmax][nvar];
    Random r = IMSLRandom();
    for (int k = 0; k < kmax; k++) {
        GCvart[k] = r.nextGaussianCopula(CholMtrx);  //probs
        for (int j = 0; j < nvar; j++) {
            /*
             *  invert Gaussian Copula probabilities to deviates using
             *  variable-specific inversions:  j = 0: Chi Square;
             *  1:  F; 2: Normal(0,1);
             *  will end up with deviate sequences ready for mapping to
             *  canonical correlation matrix:
             */
            if (j == 0) {
                //convert probs into ChiSquare(df=10) deviates:
                GCdevt[k][j] = InvCdf.chi(GCvart[k][j], 10.);
            } else if (j == 1) {
                //convert probs into F(dfn=15,dfd=10) deviates:
                GCdevt[k][j] = InvCdf.F(GCvart[k][j], 15., 10.);
            } else {
                //convert probs into Normal(mean=0,variance=1) deviates:
                GCdevt[k][j] = InvCdf.normal(GCvart[k][j]);
            }
        }
    }
    /*
     *  extract Canonical Correlation matrix from arbitrarily
     *  distributed deviate sequences GCdevt[k=0..kmax-1][j=0..nvar-1]
     *  which have been imprinted with CorrMtrxIn[i=1..nvar][j=1..nvar]
```

```
             *  above:
             */
            double CorrMtrx[][] = r.canonicalCorrelation(GCdevt);
            double relerr;
            for (int i = 0; i < nvar; i++) {
                for (int j = 0; j < i; j++) {
                    relerr = Math.abs(1. - (CorrMtrx[i][j] / CorrMtrxIn[i][j]));
                    System.out.println("CorrMtrx[" + i + "][" + j + "] = "
                            + CorrMtrx[i][j] + "; relerr = " + relerr);
                }
            }
            new PrintMatrix("Correlation Matrix: ").print(pmf, CorrMtrx);
        }
    }
}
```

Output

```
Random Example 2:

CorrMtrxIn[1][0] = -0.9486832980505138
CorrMtrxIn[2][0] = 0.8164965809277261
CorrMtrxIn[2][1] = -0.6454972243679028
         Input Correlation Matrix:
        0            1            2
0   1.000000000  -0.948683298   0.816496581
1  -0.948683298   1.000000000  -0.645497224
2   0.816496581  -0.645497224   1.000000000

Correlation Matrices calculated from
 Gaussian Copula imprinted multivariate sequence:

# vectors in multivariate sequence: 500
CorrMtrx[1][0] = -0.9502956556814602; relerr = 0.0016995741721812507
CorrMtrx[2][0] = 0.8052605146732508; relerr = 0.013761314519784795
CorrMtrx[2][1] = -0.6402027401666432; relerr = 0.008202179655294572
            Correlation Matrix:
        0            1            2
0   1.000000000  -0.950295656   0.805260515
1  -0.950295656   1.000000000  -0.640202740
2   0.805260515  -0.640202740   1.000000000

# vectors in multivariate sequence: 5000
CorrMtrx[1][0] = -0.9486118836203709; relerr = 7.527741901824925E-5
CorrMtrx[2][0] = 0.815532446740145; relerr = 0.001180818401573247
CorrMtrx[2][1] = -0.6462558369400558; relerr = 0.001175237543268981
            Correlation Matrix:
        0            1            2
0   1.000000000  -0.948611884   0.815532447
1  -0.948611884   1.000000000  -0.646255837
2   0.815532447  -0.646255837   1.000000000

# vectors in multivariate sequence: 50000
CorrMtrx[1][0] = -0.9483147908254509; relerr = 3.8844072180899136E-4
CorrMtrx[2][0] = 0.8178271707817083; relerr = 0.0016296330995904107
```

```
CorrMtrx[2][1] = -0.6466691282330974; relerr = 0.0018155056613018417
            Correlation Matrix:
        0            1            2
0   1.000000000  -0.948314791   0.817827171
1  -0.948314791   1.000000000  -0.646669128
2   0.817827171  -0.646669128   1.000000000

# vectors in multivariate sequence: 500000
CorrMtrx[1][0] = -0.9486520691467761; relerr = 3.291815488037919E-5
CorrMtrx[2][0] = 0.8165491846305497; relerr = 6.442611524937192E-5
CorrMtrx[2][1] = -0.6459060179737274; relerr = 6.333003309577645E-4
            Correlation Matrix:
        0            1            2
0   1.000000000  -0.948652069   0.816549185
1  -0.948652069   1.000000000  -0.645906018
2   0.816549185  -0.645906018   1.000000000
```

Example 3: General Discrete Random Numbers

In this example, `nextDiscrete` is used to generate 2 groups of five pseudorandom variates from the discrete distribution:

$$\Pr(X=1)=0.05$$
$$\Pr(X=2)=0.45$$
$$\Pr(X=3)=0.31$$
$$\Pr(X=4)=0.04$$
$$\Pr(X=5)=0.15$$

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

public class RandomEx3 {

    static Random IMSLRandom() {
        Random r = new Random();
        r.setSeed(123457);
        r.setMultiplier(16807);
        return r;
    }

    public static void main(String args[]) {
        int[] deviates = new int[5];
        double[] prob1 = {.05, .45, .31, .04, .15};
        PrintMatrix pm = new PrintMatrix("Random deviates");
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        Random r = IMSLRandom();

        for (int i = 0; i < 5; i++) {
            deviates[i] = r.nextDiscrete(1, prob1);
        }

        pmf.setNoColumnLabels();
        pmf.setNoRowLabels();
```

```
        pm.print(pmf, deviates);

        for (int i = 0; i < 5; i++) {
            deviates[i] = r.nextDiscrete(1, prob1);
        }

        pm.print(pmf, deviates);
    }
}
```

Output

```
Random deviates

3
2
2
3
5

Random deviates

1
3
4
5
3
```

Example 4: Discrete Uniform Random Numbers

In this example, nextDiscrete and nextUniformDiscrete are each used to generate discrete uniform pseudorandom numbers. Note that the probabilities across the possible values must be equal, in order to generate uniformly from nextDiscrete. Also note that since the same random seed is used, the methods produce the same results.

$$\Pr(X = 1) = 0.20$$
$$\Pr(X = 2) = 0.20$$
$$\Pr(X = 3) = 0.20$$
$$\Pr(X = 4) = 0.20$$
$$\Pr(X = 5) = 0.20$$

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

public class RandomEx4 {

    static Random IMSLRandom() {
        Random r = new Random();
        r.setSeed(123457);
        r.setMultiplier(16807);
        return r;
```

```
    }

    public static void main(String args[]) {
        int[] deviates = new int[5];
        int[] deviatesUniform = new int[5];
        double[] prob1 = {.20, .20, .20, .20, .20};
        PrintMatrix pm = new PrintMatrix("Random deviates");
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        Random r = IMSLRandom();

        for (int i = 0; i < 5; i++) {
            deviates[i] = r.nextDiscrete(1, prob1);
        }

        pmf.setNoColumnLabels();
        pmf.setNoRowLabels();
        pm.print(pmf, deviates);

        Random rUniform = IMSLRandom();

        for (int i = 0; i < 5; i++) {
            deviatesUniform[i] = rUniform.nextUniformDiscrete(5);
        }

        pm.print(pmf, deviatesUniform);
    }
}
```

Output

```
Random deviates

5
2
4
3
5

Random deviates

5
2
4
3
5
```

Random.BaseGenerator interface

```
public interface com.imsl.stat.Random.BaseGenerator
```

Base pseudorandom number.

Method

next

```
public int next(int bits)
```

Description

Generates the next pseudorandom number.

Parameter

`bits` – random bits

Returns

the next pseudorandom value from this random number generator's sequence.

FaureSequence class

```
public class com.imsl.stat.FaureSequence implements Serializable,
com.imsl.stat.RandomSequence, Cloneable
```

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

$$E = [0,t_1) \times \cdots \times [0,t_d)\,,\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d,$$

λ 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

$$D_n^{(d)} \le c(d)\frac{(\log n)^d}{n}$$

for all $n > 1$.

Generalized Faure sequences can be defined for any prime base $b \ge d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \ge d$, 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, $0 \le a_j(n) < b$.

The *j*-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)} a_d(n) b^{-k-1},\ 1 \le j \le d$$

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.

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.

Parameter

`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 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

```
static public int nextPrime(int n)
```

Description

Returns the smallest prime greater than or equal to n.

Parameter

`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
```

MersenneTwister class

```
public class com.imsl.stat.MersenneTwister implements
com.imsl.stat.Random.BaseGenerator, Cloneable, Serializable
```

A 32-bit Mersenne Twister generator. `MersenneTwister` generates uniform pseudorandom 32-bit numbers with a period of $2^{19937} - 1$ and a 623-dimensional equidistribution property. See Matsumoto et al. 1998 for details. The series of random numbers can be generated using a seed for initialization or by using an array of type `int`. One can also save the state of the generator at initialization to be re-used later. This generator can be used to generate non-uniform distributions by creating an `com.imsl.stat.Random` (p. 1324) object using an instance of this class as an argument to the constructor.

This Java code was translated from the the following C program.

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the

distribution.

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

http://www.math.sci.hiroshima-u.ac.jp/m̃-mat/MT/emt.html

email: m-mat@math.sci.hiroshima-u.ac.jp

Constructors

MersenneTwister

`public MersenneTwister(int seed)`

Description

Constructor for the MersenneTwister class with supplied seed.

Parameter

`seed` – an `int` which represents the seed used to initialize the 32-bit Mersenne Twister generator

MersenneTwister

`public MersenneTwister(int[] key)`

Description

Constructor for the MersenneTwister class with supplied array.

Parameter

`key` – an `int` array used to initialize the 32-bit Mersenne Twister generator

Methods

clone

`public Object clone()`

Description

Returns a clone of this object.

Returns

an `Object` which is a clone of this `MersenneTwister` object

next

`public int next(int bits)`

Description

Generates the next pseudorandom number.

Parameter

`bits` – is the number of random bits required.

Returns

the next pseudorandom value from this random number generator's sequence

nextDouble

`public double nextDouble()`

Description

Generates the next pseudorandom, uniformly distributed `double` value from this random number generator's sequence. Only the first 32 bits of the `double` are pseudorandom.

Returns

the next pseudorandom, uniformly distributed `double` value from this random number generator's sequence

nextFloat

`public float nextFloat()`

Description

Generates the next pseudorandom, uniformly distributed `float` value from this random number generator's sequence.

Returns

the next pseudorandom, uniformly distributed `float` value from this random number generator's sequence

nextInt

`public int nextInt()`

Description

Generates the next pseudorandom number.

Returns

the next pseudorandom value from this random number generator's sequence. They are uniformly distributed among all 32-bit integer values, both positive and negative.

Example: Mersenne Twister Random Number Generation

In this example, four simulation streams are generated. The first series is generated with the seed used for initialization. The second series is generated using an array for initialization. The third series is obtained by resetting the generator back to the state it had at the beginning of the second stream. Therefore, the second and third streams are identical. The fourth stream is obtained by resetting the generator back to its original, uninitialized state, and having it reinitialize using the seed. The first and fourth streams are therefore the same.

```
import com.imsl.stat.*;
import java.io.*;

public class MersenneTwisterEx1 {

    public static void main(String args[]) throws Exception {
        int nr = 4;
        double[] r = new double[nr];
        int s = 123457;
        /* Initialize MersenneTwister with a seed */
        MersenneTwister mt1 = new MersenneTwister(s);
        MersenneTwister mt2 = (MersenneTwister) mt1.clone();
        /* Save the state of MersenneTwister */
        FileOutputStream fos = new FileOutputStream("mt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(mt1);
        oos.close();
        fos.close();
        Random rndm = new Random(mt1);

        /* Get the next five random numbers */
        for (int k = 0; k < nr; k++) {
            r[k] = rndm.nextDouble();
        }

        System.out.println("              First Stream Output");
        System.out.println(r[0] + "     " + r[1] + "     " + r[2]
                + "     " + r[3]);

        /* Check the cloned copy against the original */
        Random rndm2 = new Random(mt2);
        for (int k = 0; k < nr; k++) {
            r[k] = rndm2.nextDouble();
        }

        System.out.println("\n              Clone Stream Output");
        System.out.println(r[0] + "     " + r[1] + "     " + r[2]
```

```
                + "     " + r[3]);

        /* Check the serialized copy against the original */
        FileInputStream fis = new FileInputStream("mt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        mt2 = (MersenneTwister) ois.readObject();
        Random rndm3 = new Random(mt2);
        for (int k = 0; k < nr; k++) {
            r[k] = rndm3.nextDouble();
        }
        System.out.println("\n              Serialized Stream Output");
        System.out.println(r[0] + "     " + r[1] + "     " + r[2]
                + "     " + r[3]);
    }
}
```

Output

```
              First Stream Output
0.43474506366564114     0.013851109283287921     0.49560038426424047     0.7012807898922319

              Clone Stream Output
0.43474506366564114     0.013851109283287921     0.49560038426424047     0.7012807898922319

              Serialized Stream Output
0.43474506366564114     0.013851109283287921     0.49560038426424047     0.7012807898922319
```

MersenneTwister64 class

`public class com.imsl.stat.MersenneTwister64 implements com.imsl.stat.Random.BaseGenerator, Cloneable, Serializable`

A 64-bit Mersenne Twister generator. `MersenneTwister64` generates uniform pseudorandom 64-bit numbers with a period of $2^{19937} - 1$ and a 623-dimensional equidistribution property. See Matsumoto et al. 1998 for details. Since 64-bit numbers are generated, all of the bits of both `nextFloat` and `nextDouble` are pseudorandom. The series of random numbers can be generated using a seed for initialization or by using an array of type `long`. One can also save the state of the generator at initialization to be re-used later. This generator can be used to generate non-uniform distributions by creating an `com.imsl.stat.Random` (p. 1324) object using an instance of this class as an argument to the constructor.

This Java code was translated from the the following C program.

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

http://www.math.sci.hiroshima-u.ac.jp/m̃-mat/MT/emt.html

email: m-mat@math.sci.hiroshima-u.ac.jp

Constructors

MersenneTwister64

`public MersenneTwister64(long seed)`

Description

Constructor for the MersenneTwister64 class with supplied seed.

Parameter

`seed` – a `long` which represents the seed used to initialize the 64-bit Mersenne Twister generator.

MersenneTwister64

`public MersenneTwister64(long[] key)`

Description

Constructor for the MersenneTwister64 class with supplied array.

Parameter

`key` – a `long` array used to initialize the 64-bit Mersenne Twister generator.

Methods

clone

```
public Object clone()
```

Description

Returns a clone of this object.

Returns

an `Object` which is a clone of this `MersenneTwister64` object

next

```
public int next(int bits)
```

Description

Generates the next pseudorandom number.

Parameter

`bits` – is the number of random bits required.

Returns

the next pseudorandom value from this random number generator's sequence.

nextDouble

```
public double nextDouble()
```

Description

Generates the next pseudorandom, uniformly distributed `double` value from this random number generator's sequence.

Returns

the next pseudorandom, uniformly distributed `double` value from this random number generator's sequence.

nextFloat

```
public float nextFloat()
```

Description

Generates the next pseudorandom, uniformly distributed `float` value from this random number generator's sequence.

Returns

the next pseudorandom, uniformly distributed `float` value from this random number generator's sequence.

nextLong

`public long nextLong()`

Description

Generates the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence.

Returns

the next pseudorandom, uniformly distributed `long` value from this random number generator's sequence.

Example: Mersenne Twister Random Number Generation

In this example, four simulation streams are generated. The first series is generated with the seed used for initialization. The second series is generated using an array for initialization. The third series is obtained by resetting the generator back to the state it had at the beginning of the second stream. Therefore, the second and third streams are identical. The fourth stream is obtained by resetting the generator back to its original, uninitialized state, and having it reinitialize using the seed. The first and fourth streams are therefore the same.

```
import com.imsl.stat.*;
import java.io.*;

public class MersenneTwister64Ex1 {

    public static void main(String args[]) throws Exception {
        long key[] = {0x123L, 0x234L, 0x345L, 0x456L};

        int nr = 4;
        double[] r = new double[nr];
        long s = 123457;
        /* Initialize MersenneTwister64 with a seed */
        MersenneTwister64 mt1 = new MersenneTwister64(s);
        MersenneTwister64 mt2 = (MersenneTwister64) mt1.clone();
        /* Save the state of MersenneTwister64 */
        FileOutputStream fos = new FileOutputStream("mt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(mt1);
        oos.close();
        fos.close();
        Random rndm = new Random(mt1);

        /* Get the next five random numbers */
        for (int k = 0; k < nr; k++) {
            r[k] = rndm.nextDouble();
        }
```

```
        System.out.println("              First Stream Output");
        System.out.println(r[0] + "     " + r[1] + "     " + r[2]
                + "     " + r[3]);

        /* Check the cloned copy against the original */
        Random rndm2 = new Random(mt2);
        for (int k = 0; k < nr; k++) {
            r[k] = rndm2.nextDouble();
        }

        System.out.println("\n              Clone Stream Output");
        System.out.println(+r[0] + "     " + r[1] + "     " + r[2]
                + "     " + r[3]);

        /* Check the serialized copy against the original */
        FileInputStream fis = new FileInputStream("mt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        mt2 = (MersenneTwister64) ois.readObject();
        Random rndm3 = new Random(mt2);
        for (int k = 0; k < nr; k++) {
            r[k] = rndm3.nextDouble();
        }
        System.out.println("\n              Serialized Stream Output");
        System.out.println(r[0] + "     " + r[1] + "     " + r[2]
                + "     " + r[3]);
    }
}
```

Output

```
              First Stream Output
0.5799165508168153     0.7101593787073387     0.5456686378667656     0.516359030432273

              Clone Stream Output
0.5799165508168153     0.7101593787073387     0.5456686378667656     0.516359030432273

              Serialized Stream Output
0.5799165508168153     0.7101593787073387     0.5456686378667656     0.516359030432273
```

RandomSequence interface

`public interface com.imsl.stat.RandomSequence`

Interface implemented by generators of random or quasi-random multidimensional sequences.

Methods

getDimension

```
public int getDimension()
```

Description

Returns the dimension of the sequence.

nextPoint

```
public double[] nextPoint()
```

Description

Returns the next multidimensional point in the sequence.

Returns

a `double` array of length *dimension*.

RandomSamples class

```
public class com.imsl.stat.RandomSamples
```

Generates a simple pseudorandom sample from a finite population, a sample of indices, or a permutation of an array of indices.

`getPermutation(int k)` generates a pseudorandom permutation of the integers from 1 to `k`. It begins by filling a vector of length `k` with the consecutive integers 1 to `k`. Then, with `M` initially equal to `k`, a random index `j` between 1 and `M` is generated. The element of the vector with the index `M-1` and the element with index `j` swap places in the vector. `M` is then decremented by 1 and the process repeats until `M=1`.

`getSampleIndices(int nSamp, int nPop)` generates the indices of a pseudorandom sample, without replacement, of size `nSamp` numbers from a population of size `nPop`. If `nSamp` is greater than `nPop/2`, the integers from 1 to `nPop` are selected sequentially with a probability conditional on the number selected and the number still to be considered. If, when the i-th population index is considered and j items have been included in the sample, then the index i is included with probability `(nSamp - j)/(nPop + 1 - i)`.

If `nSamp` is not greater than `nPop/2`, a O (`nSamp`) algorithm due to Ahrens and Dieter (1985) is used. Of the methods discussed by Ahrens and Dieter, SG* is used in `getSampleIndices(int nSamp, int nPop)`. It involves a preliminary selection of q indices using a geometric distribution for the distances between each index and the next one. If the preliminary sample size q is less than `nSamp`, a new preliminary sample is chosen, and this is continued until a preliminary sample greater in size than `nSamp` is chosen. This preliminary sample is then thinned using the same sampling method as that in which the sample size is greater than half of the population size.

`getSamples()` generates a pseudorandom sample from a given population, without replacement, using an algorithm due to McLeod and Bellhouse (1983). The first `nSamp` items in the population are included in the sample. Then, for each successive item from the population, a random item in the sample is replaced by that item from the population with probability equal to the sample size divided by the number of population items that have been encountered at that time.

To retrieve the random indices, use `getIndices()` after calling `getSamples()`.

Constructors

RandomSamples

```
public RandomSamples()
```

Description

Constructor for the RandomSamples class.

RandomSamples

```
public RandomSamples(Random r)
```

Description

Constructor for the RandomSamples class.

This constructor sets the random number generator to be used by the `RandomSamples` class.

Parameter

`r` – a `Random` that is the random number generator

Methods

getIndices

```
public int[] getIndices()
```

Description

Returns the indices computed from a call to `getSamples()`.

Returns

an `int` array of length `nSamp` containing the indices of the sample

getPermutation

```
public int[] getPermutation(int k)
```

Description

Returns a permutation array of integers.

Parameter

`k` – an `int` which represents the number of integers to be permuted. The integers to permute are the numbers 1, ..., k.

Returns

an `int` array containing the permuted integers

getSampleIndices

```
public int[] getSampleIndices(int nSamp, int nPop)
```

Description

Computes and returns an array of sampled indices. This is a random sample (without replacement) of the integers from 0 to `nPop-1`, in increasing order.

Parameters

`nSamp` – an `int` indicating the sample size desired

`nPop` – an `int` indicating the number of items in the population

Returns

an `int` array of length `nSamp` containing the indices of the sample

getSamples

```
public double[] getSamples(double[] population, int nSamp)
```

Description

Generates a pseudorandom sample from a given population array, without replacement.

Parameters

`population` – a `double` array containing the population to be sampled

`nSamp` – an `int` indicating the sample size desired

Returns

a `double` array containing the samples

getSamples

```
public double[][] getSamples(double[][] population, int nSamp)
```

Description

Generates a pseudorandom sample from a given population matrix, without replacement.

`getSamples()` implements an algorithm due to McLeod and Bellhouse (1983).

The first `nSamp` items in the population are included in the sample. Then, for each successive item from the population, a random item in the sample is replaced by that item from the population with probability equal to the sample size divided by the number of population items that have been encountered at that time.

To retrieve the random indices, use `getIndices()` after calling `getSamples()`.

Parameters

`population` – a `double` matrix containing the population to be sampled

`nSamp` – an `int` indicating the sample size desired

Returns

a `double` matrix containing the samples

setRandomObject

```
public void setRandomObject(Random r)
```

Description

Sets the seed for the random number generator.

Parameter

`r` – a `Random` that is the random number generator

Example 1: Permutation

In this example, the `getPermutations()` method is used to produce a pseudorandom permutation of the integers from 1 to 10.

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

public class RandomSamplesEx1 {

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

        RandomSamples rs = new RandomSamples();
        Random r = new Random(123457L);
        r.setMultiplier(16807);
        rs.setRandomObject(r);
        int[] idx = rs.getPermutation(k);

        PrintMatrix pm = new PrintMatrix("Random permutation of the integers "
                + "from 1 to 10");
        pm.print(idx);
    }
}
```

Output

```
Random permutation of the integers from 1 to 10
   0
0   5
1   9
2   2
3   8
4   1
```

```
5   6
6   4
7   7
8   3
9  10
```

Example 2: Sample Indices

In this example, the `getSampleIndices()` method is used to generate the indices of a pseudorandom sample of size 5 from a population of size 100.

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

public class RandomSamplesEx2 {

    public static void main(String[] args) {
        int nSamp = 5;
        int nPop = 100;

        RandomSamples rs = new RandomSamples();
        Random r = new Random(123457L);
        r.setMultiplier(16807);
        rs.setRandomObject(r);
        int[] idx = rs.getSampleIndices(nSamp, nPop);

        PrintMatrix pm = new PrintMatrix("Random Sample");
        pm.print(idx);
    }
}
```

Output

```
Random Sample
   0
0   2
1  22
2  53
3  61
4  79
```

Example 3: Samples

Consider the Wolfer Sunspot Data (Anderson 1971, p. 660) consisting of the number of sunspots observed each year from 1749 through 1924. In this example, the `getSamples()` method is used to generate five pseudorandom samples from the set of 176 observations of Wolfer sunspot data. The sampled indices are returned via `getIndices()`.

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

```
import com.imsl.math.PrintMatrix;
import com.imsl.math.PrintMatrixFormat;

public class RandomSamplesEx3 {

    public static void main(String[] args) {
        double[][] x = {
            {1749.0, 80.9}, {1750.0, 83.4},
            {1751.0, 47.7}, {1752.0, 47.8},
            {1753.0, 30.7}, {1754.0, 12.2},
            {1755.0, 9.6}, {1756.0, 10.2},
            {1757.0, 32.4}, {1758.0, 47.6},
            {1759.0, 54.0}, {1760.0, 62.9},
            {1761.0, 85.9}, {1762.0, 61.20},
            {1763.0, 45.0}, {1764.0, 36.4},
            {1765.0, 20.9}, {1766.0, 11.4},
            {1767.0, 37.8}, {1768.0, 69.8},
            {1769.0, 106.1}, {1770.0, 100.8},
            {1771.0, 81.6}, {1772.0, 66.5},
            {1773.0, 34.8}, {1774.0, 30.6},
            {1775.0, 7.0}, {1776.0, 19.8},
            {1777.0, 92.5}, {1778.0, 154.4},
            {1779.0, 125.9}, {1780.0, 84.8},
            {1781.0, 68.1}, {1782.0, 38.5},
            {1783.0, 22.8}, {1784.0, 10.2},
            {1785.0, 24.1}, {1786.0, 82.9},
            {1787.0, 132.0}, {1788.0, 130.9},
            {1789.0, 118.1}, {1790.0, 89.9},
            {1791.0, 66.6}, {1792.0, 60.0},
            {1793.0, 46.9}, {1794.0, 41.0},
            {1795.0, 21.3}, {1796.0, 16.0},
            {1797.0, 6.4}, {1798.0, 4.1},
            {1799.0, 6.8}, {1800.0, 14.5},
            {1801.0, 34.0}, {1802.0, 45.0},
            {1803.0, 43.1}, {1804.0, 47.5},
            {1805.0, 42.2}, {1806.0, 28.1},
            {1807.0, 10.1}, {1808.0, 8.1},
            {1809.0, 2.5}, {1810.0, 0.0},
            {1811.0, 1.4}, {1812.0, 5.0},
            {1813.0, 12.2}, {1814.0, 13.9},
            {1815.0, 35.4}, {1816.0, 45.8},
            {1817.0, 41.1}, {1818.0, 30.4},
            {1819.0, 23.9}, {1820.0, 15.7},
            {1821.0, 6.6}, {1822.0, 4.0},
            {1823.0, 1.8}, {1824.0, 8.5},
            {1825.0, 16.6}, {1826.0, 36.3},
            {1827.0, 49.7}, {1828.0, 62.5},
            {1829.0, 67.0}, {1830.0, 71.0},
            {1831.0, 47.8}, {1832.0, 27.5},
            {1833.0, 8.5}, {1834.0, 13.2},
            {1835.0, 56.9}, {1836.0, 121.5},
            {1837.0, 138.3}, {1838.0, 103.2},
            {1839.0, 85.8}, {1840.0, 63.2},
            {1841.0, 36.8}, {1842.0, 24.2},
            {1843.0, 10.7}, {1844.0, 15.0},
            {1845.0, 40.1}, {1846.0, 61.5},
```

```
        {1847.0, 98.5}, {1848.0, 124.3},
        {1849.0, 95.9}, {1850.0, 66.5},
        {1851.0, 64.5}, {1852.0, 54.2},
        {1853.0, 39.0}, {1854.0, 20.6},
        {1855.0, 6.7}, {1856.0, 4.3},
        {1857.0, 22.8}, {1858.0, 54.8},
        {1859.0, 93.8}, {1860.0, 95.7},
        {1861.0, 77.2}, {1862.0, 59.1},
        {1863.0, 44.0}, {1864.0, 47.0},
        {1865.0, 30.5}, {1866.0, 16.3},
        {1867.0, 7.3}, {1868.0, 37.3},
        {1869.0, 73.9}, {1870.0, 139.1},
        {1871.0, 111.2}, {1872.0, 101.7},
        {1873.0, 66.3}, {1874.0, 44.7},
        {1875.0, 17.1}, {1876.0, 11.3},
        {1877.0, 12.3}, {1878.0, 3.4},
        {1879.0, 6.0}, {1880.0, 32.3},
        {1881.0, 54.3}, {1882.0, 59.7},
        {1883.0, 63.7}, {1884.0, 63.5},
        {1885.0, 52.2}, {1886.0, 25.4},
        {1887.0, 13.1}, {1888.0, 6.8},
        {1889.0, 6.3}, {1890.0, 7.1},
        {1891.0, 35.6}, {1892.0, 73.0},
        {1893.0, 84.9}, {1894.0, 78.0},
        {1895.0, 64.0}, {1896.0, 41.8},
        {1897.0, 26.2}, {1898.0, 26.7},
        {1899.0, 12.1}, {1900.0, 9.5},
        {1901.0, 2.7}, {1902.0, 5.0},
        {1903.0, 24.4}, {1904.0, 42.0},
        {1905.0, 63.5}, {1906.0, 53.8},
        {1907.0, 62.0}, {1908.0, 48.5},
        {1909.0, 43.9}, {1910.0, 18.6},
        {1911.0, 5.7}, {1912.0, 3.6},
        {1913.0, 1.4}, {1914.0, 9.60},
        {1915.0, 47.4}, {1916.0, 57.10},
        {1917.0, 103.9}, {1918.0, 80.6},
        {1919.0, 63.6}, {1920.0, 37.6},
        {1921.0, 26.1}, {1922.0, 14.2},
        {1923.0, 5.8}, {1924.0, 16.7}
    };
    int nSamp = 5;

    RandomSamples rs = new RandomSamples();
    Random r = new Random(123457L);
    r.setMultiplier(16807);
    rs.setRandomObject(r);
    double[][] samples = rs.getSamples(x, nSamp);
    int[] idx = rs.getIndices();

    PrintMatrix pm = new PrintMatrix("Random Samples");
    PrintMatrixFormat mf = new PrintMatrixFormat();
    mf.setNumberFormat(new java.text.DecimalFormat("0"));
    mf.setNoColumnLabels();
    pm.print(mf, samples);
    pm = new PrintMatrix("Sampled Indices");
    pm.print(mf, idx);
```

```
    }
}
```

Output

```
Random Samples

0  1764   36
1  1828   62
2  1923    6
3  1773   35
4  1769  106

Sampled Indices

0   15
1   79
2  174
3   24
4   20
```

Example 4: Combined Example

In this example, the `getPermutations()` and `getSampleIndices()` methods are used to sample ten values from a hypothetical vector of 1,000,000 values.

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

public class RandomSamplesEx4 {

    public static void main(String[] args) {
        int nsamp = 10;
        int xlength = 1000000;

        RandomSamples rs = new RandomSamples();
        Random r = new Random(123457L);
        r.setMultiplier(16807);
        rs.setRandomObject(r);
        int[] idx = rs.getSampleIndices(nsamp, xlength);
        int[] perm = rs.getPermutation(nsamp);

        int[] samp = new int[nsamp];
        for (int i = 0; i < nsamp; i++) {
            samp[i] = idx[perm[i] - 1];
        }

        PrintMatrix pm = new PrintMatrix("Random Sample");
        pm.print(samp);
    }
}
```

Output

```
Random Sample
      0
0  434,660
1  580,208
2  514,608
3  908,824
4  376,662
5  300,775
6  330,074
7   36,541
8  439,379
9  957,579
```

Example 5: Samples

In this example, the `getSamples()` method is used to generate five pseudorandom samples from the set of 150 observations of Fisher's iris data. The sampled indices are returned via `getIndices()`.

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

public class RandomSamplesEx5 {

    public static void main(String[] args) {
        double[][] x = {
            {5.1, 3.5, 1.4, 0.2}, {4.9, 3.0, 1.4, 0.2},
            {4.7, 3.2, 1.3, 0.2}, {4.6, 3.1, 1.5, 0.2},
            {5.0, 3.6, 1.4, 0.2}, {5.4, 3.9, 1.7, 0.4},
            {4.6, 3.4, 1.4, 0.3}, {5.0, 3.4, 1.5, 0.2},
            {4.4, 2.9, 1.4, 0.2}, {4.9, 3.1, 1.5, 0.1},
            {5.4, 3.7, 1.5, 0.2}, {4.8, 3.4, 1.6, 0.2},
            {4.8, 3.0, 1.4, 0.1}, {4.3, 3.0, 1.1, 0.1},
            {5.8, 4.0, 1.2, 0.2}, {5.7, 4.4, 1.5, 0.4},
            {5.4, 3.9, 1.3, 0.4}, {5.1, 3.5, 1.4, 0.3},
            {5.7, 3.8, 1.7, 0.3}, {5.1, 3.8, 1.5, 0.3},
            {5.4, 3.4, 1.7, 0.2}, {5.1, 3.7, 1.5, 0.4},
            {4.6, 3.6, 1.0, 0.2}, {5.1, 3.3, 1.7, 0.5},
            {4.8, 3.4, 1.9, 0.2}, {5.0, 3.0, 1.6, 0.2},
            {5.0, 3.4, 1.6, 0.4}, {5.2, 3.5, 1.5, 0.2},
            {5.2, 3.4, 1.4, 0.2}, {4.7, 3.2, 1.6, 0.2},
            {4.8, 3.1, 1.6, 0.2}, {5.4, 3.4, 1.5, 0.4},
            {5.2, 4.1, 1.5, 0.1}, {5.5, 4.2, 1.4, 0.2},
            {4.9, 3.1, 1.5, 0.2}, {5.0, 3.2, 1.2, 0.2},
            {5.5, 3.5, 1.3, 0.2}, {4.9, 3.6, 1.4, 0.1},
            {4.4, 3.0, 1.3, 0.2}, {5.1, 3.4, 1.5, 0.2},
            {5.0, 3.5, 1.3, 0.3}, {4.5, 2.3, 1.3, 0.3},
            {4.4, 3.2, 1.3, 0.2}, {5.0, 3.5, 1.6, 0.6},
            {5.1, 3.8, 1.9, 0.4}, {4.8, 3.0, 1.4, 0.3},
            {5.1, 3.8, 1.6, 0.2}, {4.6, 3.2, 1.4, 0.2},
            {5.3, 3.7, 1.5, 0.2}, {5.0, 3.3, 1.4, 0.2},
```

```
    {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}
};
int nSamp = 5;

RandomSamples rs = new RandomSamples();
Random r = new Random(123457L);
r.setMultiplier(16807);
```

```
        rs.setRandomObject(r);
        double[][] samples = rs.getSamples(x, nSamp);
        int[] idx = rs.getIndices();

        PrintMatrix pm = new PrintMatrix("Random Samples");
        PrintMatrixFormat mf = new PrintMatrixFormat();
        mf.setNumberFormat(new java.text.DecimalFormat("0.0"));
        mf.setNoColumnLabels();
        pm.print(mf, samples);
        pm = new PrintMatrix("Sampled Indices");
        pm.print(mf, idx);
    }
}
```

Output

```
    Random Samples

0  5.7  4.4  1.5  0.4
1  5.7  2.6  3.5  1.0
2  6.0  2.2  5.0  1.5
3  4.8  3.4  1.9  0.2
4  5.4  3.4  1.7  0.2

Sampled Indices

0   15.0
1   79.0
2  119.0
3   24.0
4   20.0
```

Chapter 24: Input/Output

Types

class AbstractFlatFile 1373
class FlatFile 1423
class Tokenizer 1456
class MPSReader 1457

Usage Notes

The class `AbstractFlatFile` is used to read a data file and return a `java.sql.ResultSet` object. The `ResultSet` object is normally used by JDBC to return the result of an SQL query.

Since `AbstractFlatFile` is abstract it cannot be used directly. The class `FlatFile` extends `AbstractFlatFile` and is used to read flat text files. Typical flat files are comma separated files (CSV files) and space separated files. The class `FlatFile` uses the class `Tokenizer` class to split tokens on a line. This class can be extended to handle other types of flat files.

The `AbstractFlatFile` can also be extended to read binary files. It provides the support for the required members of the `java.sql.ResultSet` interface, but the actual reading of the binary files must be done in user supplied code.

The class `MPSReader` is used to read linear programming problems written in the standard MPS file format.

AbstractFlatFile class

```
abstract public class com.imsl.io.AbstractFlatFile implements
java.sql.ResultSet
```

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.

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
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Moves the cursor to the given row number in this `ResultSet` object.

Parameter

`row` – an `int` that 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

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

afterLast

```
public void afterLast() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

beforeFirst

`public void beforeFirst() throws`
`AbstractFlatFile.FlatFileSQLFeatureNotSupportedException`

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.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

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`
`AbstractFlatFile.FlatFileSQLFeatureNotSupportedException`

Description

Cancels the updates made to the current row in this `ResultSet` object. Since updates are not allowed, this method always throws a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

clearWarnings

`public void clearWarnings() throws 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.

Exception

`SQLException` if a database access error occurs

close

`public void close() throws SQLException`

Description

Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

Exception

`SQLException` if a database access error occurs

deleteRow

```
public void deleteRow() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Deletes the current row from this `ResultSet` object and from the underlying database. Since updates are not allowed, this method always throws a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

doGetBytes

```
abstract protected byte[] doGetBytes(int columnIndex) throws SQLException
```

Description

Implements the actual `getBytes()`. The bytes represent the raw values returned by the driver.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `byte` array representation of the column value. If the value is SQL `null`, the value returned is `null`.

Exception

`SQLException` if a database access error occurs

doNext

```
abstract protected boolean doNext() throws 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

Exception

`SQLException` if a database access error occurs

findColumn

```
public int findColumn(String columnLabel) throws SQLException
```

Description

Maps the given `ResultSet` column name to its `ResultSet` column index.

Parameter

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

Returns

an int specifying the column index of the given column name

Exception

SQLException if the ResultSet object does not contain columnLabel or a database access error occurs

findColumnName

```
protected String findColumnName(int columnIndex) throws SQLException
```

Description

Maps the given columnIndex into its column name.

Parameter

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

Exception

SQLException if a database access error occurs

first

```
public boolean first() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented.

getArray

```
public Array getArray(String columnLabel) throws 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.

Parameter

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

Returns

an `Array` object representing the SQL `ARRAY` value in the specified column

Exception

`SQLException` if a database access error occurs

getAsciiStream

`public InputStream getAsciiStream(int columnIndex) throws 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 does 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

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`.

Exception

`SQLException` if a database access error occurs

getAsciiStream

`public InputStream getAsciiStream(String columnLabel) throws 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 does 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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`.

Exception

`SQLException` if a database access error occurs

getBigDecimal

`public BigDecimal getBigDecimal(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

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.

Exception

`SQLException` if a conversion or database access error occurs

getBigDecimal

`public BigDecimal getBigDecimal(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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.

Exception

`SQLException` if a database access error occurs

getBinaryStream

`public InputStream getBinaryStream(int columnIndex) throws 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.

Parameter

columnIndex – an int that specifies the column. The first column is 1, the second is 2, and so on.

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.

Exception

SQLException if a database access error occurs

getBinaryStream

```
public InputStream getBinaryStream(String columnLabel) throws 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.

Parameter

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the 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.

Exception

SQLException if a database access error occurs

getBlob

```
public Blob getBlob(int columnIndex) throws 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.

NOTE: java.sql.Blob.free() has not been implemented (they are empty methods) in the instance returned. java.sql.Blob.getBinaryStream(long, long) returns null and java.sql.Blob.setBinaryStream(long) throws a SQLException.

Parameter

columnIndex – an int that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a Blob object representing the SQL BLOB value in the specified column

Exception

`SQLException` if a database access error occurs

getBlob

```
public Blob getBlob(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `Blob` object representing the SQL `BLOB` value in the specified column

Exception

`SQLException` if a database access error occurs

getBoolean

```
public boolean getBoolean(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `boolean` representation of the column value. If the value is SQL `NULL`, the value returned is `false`.

Exception

`SQLException` if a conversion or database access error occurs

getBoolean

```
public boolean getBoolean(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `boolean` representation of the column value. If the value is SQL `NULL`, the value returned is `false`.

Exception

`SQLException` if a database access error occurs

getByte

`public byte getByte(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `byte` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a conversion or database access error occurs

getByte

`public byte getByte(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `byte` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a database access error occurs

getBytes

`public byte[] getBytes(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `byte` array representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a database access error occurs

getBytes

`public byte[] getBytes(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `byte` array representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a database access error occurs

getCharacterStream

`public Reader getCharacterStream(int columnIndex) throws SQLException`

Description

Gets the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

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.

Exception

`SQLException` if a database access error occurs

getCharacterStream

`public Reader getCharacterStream(String columnLabel) throws SQLException`

Description

Gets the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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.

Exception

`SQLException` if a database access error occurs

---

getClob

`public Clob getClob(int columnIndex) throws 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.

NOTE: `java.sql.Clob.free()` has not been implemented (they are empty methods) in the instance returned. `java.sql.Clob.getCharacterStream(long, long)` returns `null`. `java.sql.Clob.position(Clob, long)`, `java.sql.Clob.setAsciiStream(long)`, `java.sql.Clob.setCharacterStream(long)`, `java.sql.Clob.setString(long, String)`, and `java.sql.Clob.setString(long, String, int, int)` throw a `SQLException`.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `Clob` object representing an SQL `Clob` value in the specified column

Exception

`SQLException` if a database access error occurs

---

getClob

`public Clob getClob(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `Clob` object representing the SQL `CLOB` value in the specified column

Exception

`SQLException` if a database access error occurs

getColumnClass

```
public Class getColumnClass(int columnIndex) throws 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`.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `Class` object used to specify the class of the data in the column

Exception

`SQLException` if a database access error occurs

getColumnCount

```
abstract public int getColumnCount() throws SQLException
```

Description

Returns the number of columns in this `ResultSet` object.

Returns

an `int` that specifies the number of columns

Exception

`SQLException` if a database access error occurs

getConcurrency

```
public int getConcurrency() throws SQLException
```

Description

Returns the concurrency mode of this `ResultSet` object.

Returns

an `int` that specifies whether concurrency is read only or an update processes as well. Always returns `CONCUR_READ_ONLY`.

Exception

`SQLException` if a database access error occurs

getCursorName

```
public String getCursorName() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Gets the name of the SQL cursor used by this `ResultSet` object. The default implementation throws a `SQLException`.

Returns

a `String` that specifies the SQL name for this `ResultSet` object's cursor

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

getDate

```
public Date getDate(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `java.sql.Date` representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a conversion or database access error occurs

getDate

```
public Date getDate(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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`.

Exception

`SQLException` if a database access error occurs

getDate

```
public Date getDate(int columnIndex, Calendar cal) throws 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` that specifies the column. The first column is 1, the second is 2, and so on.

`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.

Exception

`SQLException` if a database access error occurs

---

getDate

```
public Date getDate(String columnLabel, Calendar cal) throws 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

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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.

Exception

`SQLException` if a database access error occurs

---

getDouble

```
public double getDouble(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `double` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a conversion or database access error occurs

getDouble

`public double getDouble(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `double` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a database access error occurs

getFetchDirection

`public int getFetchDirection() throws SQLException`

Description

Returns the fetch direction for this `ResultSet` object.

Returns

an `int` that specifies the current fetch direction for this `ResultSet` object. Always returns `FETCH_FORWARD`.

Exception

`SQLException` if a database access error occurs

getFetchSize

`public int getFetchSize() throws SQLException`

Description

Returns the fetch size for this `ResultSet` object.

Returns

an `int` that specifies the current fetch size for this `ResultSet` object

Exception

`SQLException` if a database access error occurs

getFloat

`public float getFloat(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `float` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a conversion or database access error occurs

getFloat

```
public float getFloat(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `float` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a database access error occurs

getInt

```
public int getInt(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

an `int` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a conversion or database access error occurs

getInt

```
public int getInt(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

an `int` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a database access error occurs

getLong

```
public long getLong(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `long` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a conversion or database access error occurs

getLong

```
public long getLong(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `long` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a database access error occurs

getMetaData

```
public ResultSetMetaData getMetaData() throws 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

Exception

`SQLException` if a database access error occurs

getObject

`abstract public Object getObject(int columnIndex) throws 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 returns the value of the given column as a Java object. The type of the Java object is the default Java object type corresponding to the column's SQL type, following the mapping for built-in types specified in the JDBC specification.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `java.lang.Object` representation of the column value

Exception

`SQLException` if a database access error occurs

getObject

`public Object getObject(String columnLabel) throws 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 returns the value of the given column as a Java object. The type of the Java object is 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())`.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `java.lang.Object` representation of the column value

Exception

`SQLException` if a database access error occurs

getObject

```
public Object getObject(int columnIndex, Class type) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Returns the value of the designated column in the current row of this `ResultSet` object converted to the requested data type in the Java programming language.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`type` – a Java data type to convert to

Returns

an `Object` representing the SQL value in the specified column

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

getObject

```
public Object getObject(int columnIndex, Map map) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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` that specifies the column. The first column is 1, the second is 2, and so on.

`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

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

getObject

```
public Object getObject(String columnLabel, Class type) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Returns the value of the designated column in the current row of this `ResultSet` object converted to the requested data type in the Java programming language.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`type` – a Java data type to convert to

Returns

an `Object` representing the SQL value in the specified column

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

getObject

```
public Object getObject(String columnLabel, Map map) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

getRef

```
public Ref getRef(int columnIndex) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `Ref` object representing the SQL `REF` value in the specified column

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

getRef

`public Ref getRef(String columnLabel) throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException`

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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `Ref` object representing the SQL `REF` value in the specified column

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

getRow

`public int getRow() throws SQLException`

Description

Retrieves the current row number. The first row is number 1, the second number 2, and so on.

Returns

an `int` that specifies the current row number; `0` if there is no current row

Exception

`SQLException` if a database access error occurs

---

getShort

`public short getShort(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `short` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a conversion or database access error occurs

getShort

`public short getShort(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `short` representation of the column value. If the value is SQL `NULL`, the value returned is `0`.

Exception

`SQLException` if a database access error occurs

getStatement

`public Statement getStatement() throws SQLException`

Description

Returns the `Statement` object that produced this `ResultSet` object. Since there is no `java.sqlStatement`, this method always throws a `SQLException`.

Returns

the `Statment` object that produced this `ResultSet` object or `null` if the result set was produced some other way

Exception

`SQLException` is always thrown since updates are not allowed

getString

`public String getString(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `String` representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a database access error occurs

getString

`public String getString(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `String` representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a database access error occurs

getTime

`public Time getTime(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `java.sql.Time` representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a conversion or database access error occurs

getTime

`public Time getTime(String columnLabel) throws 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.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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`.

Exception

`SQLException` if a database access error occurs

getTime

```
public Time getTime(int columnIndex, Calendar cal) throws 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` that specifies the column. The first column is 1, the second is 2, and so on.

`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.

Exception

`SQLException` if a database access error occurs

getTime

```
public Time getTime(String columnLabel, Calendar cal) throws 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

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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.

Exception

`SQLException` if a database access error occurs

getTimestamp

```
public Timestamp getTimestamp(int columnIndex) throws 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.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Returns

a `java.sql.Timestamp` representation of the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a conversion or database access error occurs

getTimestamp

```
public Timestamp getTimestamp(String columnLabel) throws SQLException
```

Description

Gets the value of the designated column in the current row of this `ResultSet` object as a `java.sql.Timestamp` object.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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`

Exception

`SQLException` if a database access error occurs

getTimestamp

```
public Timestamp getTimestamp(int columnIndex, Calendar cal) throws
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 that specifies the column. The first column is 1, the second is 2, and so on.

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.

Exception

SQLException if a database access error occurs

getTimestamp

```
public Timestamp getTimestamp(String columnLabel, Calendar cal) throws
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

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the 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.

Exception

SQLException if a database access error occurs

getType

```
public int getType() throws SQLException
```

Description

Returns the type of this ResultSet object. The type is determined by the Statement object that created the result set.

Returns

an int that specifies the type of this ResultSet object. Always returns TYPE_FORWARD_ONLY.

Exception

SQLException if a database access error occurs

getURL

```
public URL getURL(int columnIndex) throws SQLException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.net.URL` object.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

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.

Exception

`SQLException` if a conversion or database access error occurs

getURL

```
public URL getURL(String columnLabel) throws SQLException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.net.URL` object.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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.

Exception

`SQLException` if a database access error occurs

getWarnings

```
public SQLWarning getWarnings() throws SQLException
```

Description

Returns the first warning reported by calls on this `ResultSet` object. Subsequent warnings on this `ResultSet` object are 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) are chained on the `Statement` object.

Returns

the first `SQLWarning` object reported or `null`

Exception

`SQLException` if a database access error occurs

insertRow

```
public void insertRow() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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 a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

isAfterLast

```
public boolean isAfterLast() throws 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

Exception

`SQLException` if a database access error occurs

isBeforeFirst

```
public boolean isBeforeFirst() throws 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

Exception

`SQLException` if a database access error occurs

isFirst

```
public boolean isFirst() throws 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

Exception

`SQLException` if a database access error occurs

isLast

```
public boolean isLast() throws 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

Exception

`SQLException` if a database access error occurs

last

```
public boolean last() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

moveToCurrentRow

```
public void moveToCurrentRow() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Moves the cursor to the remembered cursor position, usually the current row. Since updates are not allowed, this method always throws a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

moveToInsertRow

```
public void moveToInsertRow() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Moves the cursor to the insert row. Since updates are not allowed, this method always throws a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

next

```
public boolean next() throws 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

Exception

`SQLException` if a database access error occurs or this method is called on a closed result set

---

previous

```
public boolean previous() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

refreshRow

```
public void refreshRow() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Refreshes the current row with its most recent value in the database. Since updates are not allowed, this method always throws a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

relative

```
public boolean relative(int rows) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Moves the cursor a relative number of rows, either positive or negative.

Parameter

`rows` – an `int` that 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

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

rowDeleted

```
public boolean rowDeleted() throws 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.

Exception

`SQLException` if a database access error occurs

rowInserted

```
public boolean rowInserted() throws 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.

Exception

`SQLException` if a database access error occurs

rowUpdated

```
public boolean rowUpdated() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

setColumnClass

`protected void setColumnClass(int columnIndex, 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, 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 SQLException`

Description

Gives a hint as to the direction in which the rows in this `ResultSet` object is processed.

Parameter

`direction` – an `int` that specifies the expected direction this `ResultSet` object is to be processed

Exception

`SQLException` if the fetch direction is not `FETCH_FORWARD`

setFetchSize

`public void setFetchSize(int rows) throws 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.

Parameter

`rows` – an `int` that specifies the number of rows to fetch

Exception

`SQLException` if a database access error occurs or the condition `0 = rows = this.getMaxRows()` is not satisfied

setWarning

```
protected void setWarning(SQLWarning warning)
```

Description

Sets a `SQLWarning`.

Parameter

`warning` – a `SQLWarning` that is to be added to this object.

updateArray

```
public void updateArray(int column, Array x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `Array` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`column` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.sql.Array` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateArray

```
public void updateArray(String columnLabel, Array x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `Array` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Array` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateAsciiStream

```
public void updateAsciiStream(int columnIndex, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – an `InputStream` that specifies the new column value

`length` – an `int` that specifies the stream length

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateAsciiStream

```
public void updateAsciiStream(String columnLabel, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – an `InputStream` that specifies the new column value

`length` – an `int` that specifies the stream length

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateBigDecimal

```
public void updateBigDecimal(int columnIndex, BigDecimal x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.math.BigDecimal` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.math.BigDecimal` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateBigDecimal

```
public void updateBigDecimal(String columnLabel, BigDecimal x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.BigDecimal` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.BigDecimal` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateBinaryStream

```
public void updateBinaryStream(int columnIndex, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – an `InputStream` that specifies the new column value

`length` – an `int` that specifies the stream length

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateBinaryStream

```
public void updateBinaryStream(String columnLabel, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

x – an InputStream that specifies the new column value

length – an int that specifies the stream length

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented.

updateBlob

```
public void updateBlob(int column, Blob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an java.sql.Blob value. Since updates are not allowed, this method always throws a SQLException.

Parameters

column – an int that specifies the column. The first column is 1, the second is 2, and so on.

x – a java.sql.Blob that specifies the new column value

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented.

updateBlob

```
public void updateBlob(String columnLabel, Blob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an java.sql.Blob value. Since updates are not allowed, this method always throws a SQLException.

Parameters

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

x – a java.sql.Blob that specifies the new column value

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented.

updateBoolean

```
public void updateBoolean(int columnIndex, boolean x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `boolean` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `boolean` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateBoolean

```
public void updateBoolean(String columnLabel, boolean x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `boolean` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `boolean` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateByte

```
public void updateByte(int columnIndex, byte x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `byte` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `byte` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateByte

```
public void updateByte(String columnLabel, byte x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `byte` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `byte` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateBytes

```
public void updateBytes(int columnIndex, byte[] x) throws SQLException
```

Description

Updates the designated column with a `byte` array value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `byte` array that specifies the new column value

Exception

`SQLException` is always thrown since updates are not allowed

updateBytes

```
public void updateBytes(String columnLabel, byte[] x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `byte` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `byte` array that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateCharacterStream

```
public void updateCharacterStream(int columnIndex, Reader x, int length) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `Reader` that specifies the new column value

`length` – an `int` that specifies the stream length

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateCharacterStream

```
public void updateCharacterStream(String columnLabel, Reader reader, int
length) throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

`length` – an `int` that specifies the stream length

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateClob

```
public void updateClob(int column, Clob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `java.sql.Clob` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`column` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.sql.Clob` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateClob

```
public void updateClob(String columnLabel, Clob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `java.sql.Clob` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Clob` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

updateDate

```
public void updateDate(int columnIndex, Date x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Date` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.sql.Date` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

updateDate

```
public void updateDate(String columnLabel, Date x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Date` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Date` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

updateDouble

```
public void updateDouble(int columnIndex, double x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `double` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `double` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateDouble

```
public void updateDouble(String columnLabel, double x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `double` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `double` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateFloat

```
public void updateFloat(int columnIndex, float x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `float` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `float` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateFloat

```
public void updateFloat(String columnLabel, float x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `float` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `float` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateInt

```
public void updateInt(int columnIndex, int x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `int` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – an `int` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateInt

```
public void updateInt(String columnLabel, int x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `int` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – an `int` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateLong

```
public void updateLong(int columnIndex, long x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `long` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `long` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateLong

```
public void updateLong(String columnLabel, long x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `long` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `long` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateNull

```
public void updateNull(int columnIndex) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Gives a nullable column a `null` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateNull

```
public void updateNull(String columnLabel) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `null` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateObject

```
public void updateObject(int columnIndex, Object x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – an `Object` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateObject

```
public void updateObject(String columnLabel, Object x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Object` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateObject

```
public void updateObject(int columnIndex, Object x, int scale) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – an `Object` that 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 is ignored.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateObject

```
public void updateObject(String columnLabel, Object x, int scale) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `Object` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – an `Object` that 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 is ignored.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateRef

```
public void updateRef(int column, Ref x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `java.sql.Ref` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`column` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.sql.Ref` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateRef

```
public void updateRef(String columnLabel, Ref x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an `java.sql.Ref` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Ref` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateRow

```
public void updateRow() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

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 a `SQLException`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateShort

```
public void updateShort(int columnIndex, short x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `short` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `short` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateShort

```
public void updateShort(String columnLabel, short x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `short` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `short` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateString

```
public void updateString(int columnIndex, String x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `String` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `String` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateString

```
public void updateString(String columnLabel, String x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `String` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `String` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

updateTime

```
public void updateTime(int columnIndex, Time x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Time` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.sql.Time` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

updateTime

```
public void updateTime(String columnLabel, Time x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Time` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Time` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

updateTimestamp

```
public void updateTimestamp(int columnIndex, Timestamp x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Timestamp` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second is 2, and so on.

`x` – a `java.sql.Timestamp` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

---

updateTimestamp

```
public void updateTimestamp(String columnLabel, Timestamp x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Timestamp` value. Since updates are not allowed, this method always throws a `SQLException`.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `java.sql.Timestamp` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented.

wasNull

```
public boolean wasNull() throws 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` otherwise `false`

Exception

`SQLException` if a database access error occurs

AbstractFlatFile.FlatFileSQLException class

```
static protected class com.imsl.io.AbstractFlatFile.FlatFileSQLException
extends java.sql.SQLException
```

A SQLException thrown by the AbstractFlatFile class.

AbstractFlatFile.FlatFileSQLFeatureNotSupportedException class

```
static protected class
com.imsl.io.AbstractFlatFile.FlatFileSQLFeatureNotSupportedException extends
java.sql.SQLFeatureNotSupportedException
```

A SQLFeatureNotSupportedException thrown by the AbstractFlatFile class.

FlatFile class

`public class com.imsl.io.FlatFile extends com.imsl.io.AbstractFlatFile`

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 `java.io.BufferedReader.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.

Fields

PARSE_BYTE

`static final public FlatFile.Parser PARSE_BYTE`

Implements a `Parser` that converts a `String` to a `Byte`.

PARSE_DOUBLE

`static final public FlatFile.Parser PARSE_DOUBLE`

Implements a `Parser` that converts a `String` to a `Double`.

PARSE_FLOAT

`static final public FlatFile.Parser PARSE_FLOAT`

Implements a `Parser` that converts a `String` to a `Float`.

PARSE_INTEGER

`static final public FlatFile.Parser PARSE_INTEGER`

Implements a `Parser` that converts a `String` to an `Integer`.

PARSE_LONG

`static final public FlatFile.Parser PARSE_LONG`

Implements a `Parser` that converts a `String` to a `Long`.

PARSE_SHORT

`static final public FlatFile.Parser PARSE_SHORT`

Implements a `Parser` that converts a `String` to a `Short`.

Constructors

FlatFile

`public FlatFile(BufferedReader reader) throws IOException`

Description

Creates a `FlatFile` with the CSV tokenizer.

The CSV tokenizer is for reading comma separated value files.

Parameter

`reader` – a `BufferedReader` that is the stream to be read

FlatFile

`public FlatFile(String filename) throws IOException`

Description

Creates a FlatFile from a CSV file.

A CSV file is a comma separated value file.

Parameter

`filename` – a `String` that specifies the name of the file to be read

FlatFile

`public FlatFile(BufferedReader reader, Tokenizer tokenizer)`

Description

Creates a FlatFile from a BufferedReader.

Parameters

`reader` – a `BufferedReader` that is the stream to be read

`tokenizer` – a `Tokenizer` that splits a text line into tokens, one per column

FlatFile

`public FlatFile(String filename, Tokenizer tokenizer) throws IOException`

Description

Creates a `FlatFile` from a file.

Parameters

`filename` – a `String` that specifies the name of the file to be read

`tokenizer` – a `Tokenizer` that splits a text line into tokens, one per column

Methods

doGetBytes

`protected byte[] doGetBytes(int columnIndex) throws SQLException`

Description

Gets the value of the designated column in the current row as a `byte` array.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

Returns

a `byte` array that contains the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`SQLException` if a database access error occurs

doNext

`protected boolean doNext() throws 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

a `boolean`, `true` if the new current row is valid, `false` if there are no more rows

Exception

`SQLException` if a database access error occurs

getColumnCount

`public int getColumnCount() throws SQLException`

Description

Returns the number of columns in this `ResultSet` object.

Returns

an `int` that specifies the number of columns

Exception

`SQLException` if a database access error occurs

getHoldability

```
public int getHoldability() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the holdability of this `ResultSet` object.

Returns

an `int` that is either `ResultSet.HOLD_CURSORS_OVER_COMMIT` or
`ResultSet.CLOSE_CURSORS_AT_COMMIT`

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

getNCharacterStream

```
public Reader getNCharacterStream(int columnIndex) throws
SQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object. It is intended for use when accessing `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

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.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

getNCharacterStream

```
public Reader getNCharacterStream(String columnLabel) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.io.Reader` object. It is intended for use when accessing `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the 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.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

getNClob

```
public NClob getNClob(int columnIndex) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `NClob` object in the Java programming language.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

Returns

an `NClob` object representing the SQL `NCLOB` value in the specified column

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

getNClob

```
public NClob getNClob(String columnLabel) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `NClob` object in the Java programming language.

Parameter

`columnLabel` – an `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

an `NClob` object representing the SQL `NCLOB` value in the specified column

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

getNString

```
public String getNString(int columnIndex) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `String` in the Java programming language. It is intended for use when accessing `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

Returns

a `String` that specifies the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

getNString

```
public String getNString(String columnLabel) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `String` in the Java programming language. It is intended for use when accessing `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

Parameter

`columnLabel` – an `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `String` that specifies the column value. If the value is SQL `NULL`, the value returned is `null`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

getObject

```
public Object getObject(int columnIndex) throws 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 returns the value of the given column as a Java object. The type of the Java object is 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())`.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

Returns

a `java.lang.Object` holding the column value

Exception

`SQLException` if a database access error occurs

getRowId

```
public RowId getRowId(int columnIndex) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.RowId` object in the Java programming language.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

Returns

a `RowId` that is the column value. If the value is a SQL `NULL` the value returned is `null`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

getRowId

```
public RowId getRowId(String columnLabel) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` object as a `java.sql.RowId` object in the Java programming language.

Parameter

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

a `RowId` that is the column value. If the value is a SQL `NULL` the value returned is `null`.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

getSQLXML

```
public SQLXML getSQLXML(int columnIndex) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` as a `java.sql.SQLXML` object in the Java programming language.

Parameter

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

Returns

an `SQLXML` object that maps an SQL XML value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

getSQLXML

```
public SQLXML getSQLXML(String columnLabel) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves the value of the designated column in the current row of this `ResultSet` as a `java.sql.SQLXML` object in the Java programming language.

Parameter

`columnLabel` – an `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

Returns

an `SQLXML` object that maps an SQL XML value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

isClosed

```
public boolean isClosed() throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Retrieves whether this `ResultSet` object has been closed. A `ResultSet` is closed if the method close has been called on it, or if it is automatically closed.

Returns

a `boolean`, `true` if this `ResultSet` object is closed, `false` if it is still open

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

isWrapperFor

```
public boolean isWrapperFor(Class iface) throws SQLException
```

Description

Returns `true` if this either implements the interface argument or is directly or indirectly a wrapper for an object that does. Returns `false` otherwise. If this implements the interface then return `true`, else if this is a wrapper then return the result of recursively calling `isWrapperFor` on the wrapped object. If this does not implement the interface and is not a wrapper, return `false`. This method should be implemented as a low-cost operation compared to unwrap so that callers can use this method to avoid expensive unwrap calls that may fail. If this method returns `true` then calling unwrap with the same argument should succeed.

Parameter

`iface` – a `Class` defining an interface

Returns

a `boolean`, `false`

Exception

`SQLException` if an error occurs while determining whether this is a wrapper for an object with the given interface

readLine

`protected String readLine() throws IOException`

Description

Reads and returns a line from the input.

Returns

a `String` that contains a line from the input

Exception

`IOException` thrown if an IO exception occurs

setColumnClass

`protected void setColumnClass(int columnIndex, Class columnClass)`

Description

Sets a column class.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`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` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`columnParser` – a `Parser` that is the parser to be used to parse entries in the specified column

setDateColumnParser

```
protected void setDateColumnParser(int columnIndex, String pattern, Locale
locale)
```

Description

Creates for a pattern string and sets the Parser for the specified column.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`pattern` – a `String` that specifies a pattern used to construct a `java.text.SimpleDateFormat` object used to parse the column

`locale` – a `Locale` that specifies the locale for the date format parser

unwrap

```
public Object unwrap(Class iface) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Returns an object that implements the given interface to allow access to non-standard methods, or standard methods not exposed by the proxy. If the receiver implements the interface then the result is the receiver or a proxy for the receiver. If the receiver is a wrapper and the wrapped object implements the interface then the result is the wrapped object or a proxy for the wrapped object. Otherwise returns the result of calling `unwrap` recursively on the wrapped object or a proxy for that result. If the receiver is not a wrapper and does not implement the interface, then an `SQLException` is thrown.

Parameter

`iface` – a `Class` defining an interface that the result must implement

Returns

an object that implements the interface. The object may be a proxy for the actual implementing object.

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateAsciiStream

```
public void updateAsciiStream(int columnIndex, InputStream x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

> `columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.
>
> `x` – a `InputStream` that specifies the new column value

Exception

> `FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateAsciiStream

```
public void updateAsciiStream(String columnLabel, InputStream x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

> `columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.
>
> `x` – a `InputStream` that specifies the new column value

Exception

> `FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateAsciiStream

```
public void updateAsciiStream(int columnIndex, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

> `columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.
>
> `x` – a `InputStream` that specifies the new column value
>
> `length` – an `int` that specifies the length of the stream

Exception

> `FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateAsciiStream

```
public void updateAsciiStream(int columnIndex, InputStream x, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `InputStream` that specifies the new column value

`length` – an `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateAsciiStream

```
public void updateAsciiStream(String columnLabel, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `InputStream` that specifies the new column value

`length` – an `int` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateAsciiStream

```
public void updateAsciiStream(String columnLabel, InputStream x, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with an ASCII stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

x – a InputStream that specifies the new column value

length – an long that specifies the length of the stream

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented

updateBinaryStream

```
public void updateBinaryStream(int columnIndex, InputStream x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the updateRow or insertRow methods are called to update the database.

Parameters

columnIndex – an int that specifies the column. The first column is 1, the second 2, and so on.

x – a InputStream that specifies the new column value

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented

updateBinaryStream

```
public void updateBinaryStream(String columnLabel, InputStream x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the updateRow or insertRow methods are called to update the database.

Parameters

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

x – a InputStream that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBinaryStream

```
public void updateBinaryStream(int columnIndex, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `InputStream` that specifies the new column value

`length` – an `int` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBinaryStream

```
public void updateBinaryStream(int columnIndex, InputStream x, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `InputStream` that specifies the new column value

`length` – an `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBinaryStream

```
public void updateBinaryStream(String columnLabel, InputStream x, int length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `InputStream` that specifies the new column value

`length` – an `int` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBinaryStream

```
public void updateBinaryStream(String columnLabel, InputStream x, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a binary stream value. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `InputStream` that specifies the new column value

`length` – an `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBlob

```
public void updateBlob(int columnIndex, InputStream inputStream) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given input stream. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`inputStream` – an `InputStream` that contains the data to set the parameter value to

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBlob

```
public void updateBlob(int columnIndex, Blob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Blob` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Blob` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBlob

```
public void updateBlob(String columnLabel, InputStream inputStream) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given input stream. The data will be read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`inputStream` – an `InputStream` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBlob

```
public void updateBlob(String columnLabel, Blob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Blob` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `Blob` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBlob

```
public void updateBlob(int columnIndex, InputStream inputStream, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given input stream, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`inputStream` – an `InputStream` that contains the data to set the parameter value to

`length` – an `long` that specifies the number of bytes in the parameter data

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateBlob

```
public void updateBlob(String columnLabel, InputStream inputStream, long
length) throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given input stream, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`inputStream` – an `InputStream` that specifies the new column value

`length` – an `long` that specifies the number of bytes in the parameter data

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateCharacterStream

```
public void updateCharacterStream(int columnIndex, Reader x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value. The data is read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Reader` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateCharacterStream

```
public void updateCharacterStream(String columnLabel, Reader reader) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value. The data is read from the stream as needed until end-of-stream is reached.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateCharacterStream

```
public void updateCharacterStream(int columnIndex, Reader x, int length) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Reader` that specifies the new column value

`length` – an `int` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateCharacterStream

```
public void updateCharacterStream(int columnIndex, Reader x, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Reader` that specifies the new column value

`length` – an `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateCharacterStream

```
public void updateCharacterStream(String columnLabel, Reader reader, int
length) throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

`length` – an `int` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateCharacterStream

```
public void updateCharacterStream(String columnLabel, Reader reader, long
length) throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value, which is the specified number of bytes.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

`length` – an `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateClob

```
public void updateClob(int columnIndex, Reader reader) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader` object. The data is read from the stream as needed until end-of-stream is reached. The JDBC driver does any necessary conversion from UNICODE to the database `char` format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`reader` – a `Reader` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

updateClob

```
public void updateClob(int columnIndex, Clob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Clob` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Clob` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

updateClob

```
public void updateClob(String columnLabel, Reader reader) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader` object. The data is read from the stream as needed until end-of-stream is reached. The JDBC driver does any necessary conversion from UNICODE to the database `char` format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

---

updateClob

```
public void updateClob(String columnLabel, Clob x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.Clob` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `Clob` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateClob

```
public void updateClob(int columnIndex, Reader reader, long length) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader` object, which is the given number of characters long. When a very large UNICODE value is input to a `LONGVARCHAR` parameter, it may be more practical to send it via a `java.io.Reader` object. The JDBC driver does any necessary conversion from UNICODE to the database `char` format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`reader` – a `Reader` that specifies the new column value

`length` – a `long` that specifies the number of characters in the parameter data

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateClob

```
public void updateClob(String columnLabel, Reader reader, long length) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader` object, which is the given number of characters long. When a very large UNICODE value is input to a `LONGVARCHAR` parameter, it may be more practical to send it via a `java.io.Reader` object. The JDBC driver does any necessary conversion from UNICODE to the database `char` format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

`length` – a `long` that specifies the number of characters in the parameter data

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNCharacterStream

```
public void updateNCharacterStream(int columnIndex, Reader x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value. The data is read from the stream as needed until end-of-stream is reached. The driver does the necessary conversion from Java character format to the national character set in the database. It is intended for use when updating `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Reader` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNCharacterStream

```
public void updateNCharacterStream(String columnLabel, Reader reader) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value. The data is read from the stream as needed until end-of-stream is reached. The driver does the necessary conversion from Java character format to the national character set in the database. It is intended for use when updating `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNCharacterStream

```
public void updateNCharacterStream(int columnIndex, Reader x, long length)
throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value, which is the specified number of bytes. The driver does the necessary conversion from Java character format to the national character set in the database. It is intended for use when updating `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `Reader` that specifies the new column value

`length` – a `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNCharacterStream

```
public void updateNCharacterStream(String columnLabel, Reader reader, long
length) throws AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a character stream value, which is the specified number of bytes. The driver does the necessary conversion from Java character format to the national character set in the database. It is intended for use when updating `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that specifies the new column value

`length` – a `long` that specifies the length of the stream

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNClob

```
public void updateNClob(int columnIndex, Reader reader) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader`. The data is read from the stream as needed until end-of-stream is reached. The JDBC driver does any necessary conversion from UNICODE to the database `char` format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`reader` – a `Reader` that contains the data to set the parameter value to

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNClob

```
public void updateNClob(int columnIndex, NClob nClob) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.NClob` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`nClob` – an `NClob` that specifies the value for the column to be updated

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNClob

```
public void updateNClob(String columnLabel, Reader reader) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader` object. The data is read from the stream as needed until end-of-stream is reached. The JDBC driver does any necessary conversion from UNICODE to the database char format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

reader – a Reader that contains the data to set the parameter value to

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented

updateNClob

```
public void updateNClob(String columnLabel, NClob nClob) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a java.sql.NClob value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the updateRow or insertRow methods are called to update the database.

Parameters

columnLabel – a String that indicates the label for the column specified with the SQL AS clause. If the SQL AS clause was not specified, then the label is the name of the column.

nClob – a NClob that specifies the value for the column to be updated

Exception

FlatFileSQLFeatureNotSupportedException this feature has not been implemented

updateNClob

```
public void updateNClob(int columnIndex, Reader reader, long length) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given Reader object, which is the given number of characters long. When a very large UNICODE value is input to a LONGVARCHAR parameter, it may be more practical to send it via a java.io.Reader object. The JDBC driver does any necessary conversion from UNICODE to the database char format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the updateRow or insertRow methods are called to update the database.

Parameters

columnIndex – an int that specifies the column. The first column is 1, the second 2, and so on.

reader – a Reader that contains the data to set the parameter value to

length – a long that specifies the number of characters in the parameter data

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNClob

```
public void updateNClob(String columnLabel, Reader reader, long length) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column using the given `Reader` object, which is the given number of characters long. When a very large UNICODE value is input to a `LONGVARCHAR` parameter, it may be more practical to send it via a `java.io.Reader` object. The JDBC driver does any necessary conversion from UNICODE to the database `char` format.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`reader` – a `Reader` that contains the data to set the parameter value to

`length` – a `long` that specifies the number of characters in the parameter data

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNString

```
public void updateNString(int columnIndex, String nString) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `String` value. It is intended for use when updating, `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`nString` – a `String` that specifies the value for the column to be updated

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateNString

```
public void updateNString(String columnLabel, String nString) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a String value. It is intended for use when updating `NCHAR`, `NVARCHAR` and `LONGNVARCHAR` columns.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`nString` – a `String` that specifies the value for the column to be updated

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateRowId

```
public void updateRowId(int columnIndex, RowId x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `RowId` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`x` – a `RowId` that specifies the column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateRowId

```
public void updateRowId(String columnLabel, RowId x) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `RowId` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`x` – a `RowId` that specifies the column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateSQLXML

```
public void updateSQLXML(int columnIndex, SQLXML xmlObject) throws
AbstractFlatFile.FlatFileSQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.SQLXML` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnIndex` – an `int` that specifies the column. The first column is 1, the second 2, and so on.

`xmlObject` – a `SQLXML` object that specifies the value for the column to be updated

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

updateSQLXML

```
public void updateSQLXML(String columnLabel, SQLXML xmlObject) throws
SQLFeatureNotSupportedException
```

Description

Updates the designated column with a `java.sql.SQLXML` value.

The updater methods are used to update column values in the current row or the insert row. The updater methods do not update the underlying database. Instead the `updateRow` or `insertRow` methods are called to update the database.

Parameters

`columnLabel` – a `String` that indicates the label for the column specified with the SQL `AS` clause. If the SQL `AS` clause was not specified, then the label is the name of the column.

`xmlObject` – a `SQLXML` object the specifies the column value

Exception

`FlatFileSQLFeatureNotSupportedException` this feature has not been implemented

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());

        iris.close();
        is.close();
```

```
    }
}
```

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 `getResourceAsStream` 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.*;
import java.util.StringTokenizer;

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);
        }

        reader.close();
        br.close();
        is.close();
    }
}
```

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
```

FlatFile.Parser interface

`public interface com.imsl.io.FlatFile.Parser`

Defines a method that parses a String into an Object.

Method

parse

`public Object parse(String input) throws SQLException`

Description

Parses a `String` into an `Object`.

Parameter

`input` – a `String` that specifies the input to be parsed

Returns

an `Object` that contains the value of the `String`

Exception

`SQLException` thrown if a database error occurs

Tokenizer class

`public class com.imsl.io.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.

Constructor

Tokenizer

```
public Tokenizer(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 String nextToken()
```

Description

Returns the next token.

Returns

the next token.

Exception

`NoSuchElementException` if there are no more tokens to be returned.

parse

```
public void parse(String line)
```

Description

Sets the line to be tokenized. Any tokens left from the previous line are discarded.

Parameter

`line` – is the line to be tokenized.

MPSReader class

```
public class com.imsl.io.MPSReader implements Serializable
```

Reads a linear programming problem from an MPS file.

An MPS file defines a linear or quadratic programming problem. Linear programming problems read using this class are assumed to be of the form:

$$\min_{x \in R^n} c^T x$$

subject to

$$b_l \leq Ax \leq b_u$$

$$x_l \leq x \leq x_u$$

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.

The following table helps map this notation into the use of `MPSReader`.

C	Objective
A	Constraint matrix
b_l	Lower Range
b_u	Upper Range
x_l	Lower Bound
x_u	Upper Bound

If the MPS file specifies an equality constraint or bound, the corresponding lower and upper values will be exactly equal.

The problem formulation assumes that the constraints and bounds are two-sided. If a particular constraint or bound has no lower limit, then the corresponding entry in the structure is set to negative machine infinity. If the upper limit is missing, then the corresponding entry in the structure is set to positive machine infinity.

MPS File Format

There is some variability in the MPS format. This section describes the MPS format accepted by this reader.

An MPS file consists of a number of sections. Each section begins with a name in column 1. With the exception of the NAME section, the rest of this line is ignored. Lines with a '*' or '$' in column 1 are considered comment lines and are ignored.

The body of each section consists of lines divided into fields, as follows:

Field Number	Columns	Content
1	2-3	Indicator
2	5-12	Name
3	15-22	Name
4	25-36	Value
5	40-47	Name
6	50-61	Value

The format limits MPS names to 8 characters and values to 12 characters. The names in fields 2, 3 and 5 are case sensitive. Leading and trailing blanks are ignored, but internal spaces are significant.

The sections in an MPS file are as follows:

NAME

ROWS

COLUMNS

RHS

RANGES (optional)

BOUNDS (optional)

QUADRATIC (optional)

ENDATA

Sections must occur in the above order.

MPS keywords, section names and indicator values, are case insensitive. Row, column and set names are case sensitive.

NAME Section

The NAME section contains the single line. A problem name can occur anywhere on the line after NAME and before columns 62. The problem name is truncated to 8 characters.

ROWS Section

The ROWS section defines the name and type for each row. Field 1 contains the row type and field 2 contains the row name. Row type values are not case sensitive. Row names are case sensitive. The following row types are allowed:

Row Type	Meaning
E	Equality constraint
L	Less than or equal constraint
G	Greater than or equal constraint
N	Objective of a free row

COLUMNS Section

The COLUMNS section defines the nonzero entries in the objective and the constraint matrix. The row names here must have been defined in the ROWS section.

Field	Contents
2	Column name
3	Row name
4	Value for the entry whose row and column are given by fields 2 and 3
5	Row name
6	Value for the entry whose row and column are given by fields 2 and 5

Note: Fields 5 and 6 are optional.

The COLUMNS section can also contain markers. These are indicated by the name 'MARKER' (with the quotes) in field 3 and the marker type in field 4 or 5.

Marker type 'INTORG' (with the quotes) begins an integer group. The marker type 'INTEND' (with the quotes) ends this group. The variables corresponding to the columns defined within this group are required to be integer.

RHS Section

The RHS section defines the right-hand side of the constraints. An MPS file can contain more than one RHS set, distinguished by the RHS set name. The row names here must be defined in the ROWS section.

Field	Contents
2	RHS name
3	Row name
4	Value for the entry whose row and column are given by fields 2 and 3
5	Row name
6	Value for the entry whose row and column are given by fields 2 and 5

Note: Fields 5 and 6 are optional.

RANGES Section

The optional RANGES section defines two-sided constraints. An MPS file can contain more than one range set, distinguished by the range set name. The row names here must have been defined in the ROWS section.

Field	Contents
2	Range set name
3	Row name
4	Value for the entry whose row and column are given by fields 2 and 3
5	Row name
6	Value for the entry whose row and column are given by fields 2 and 5

Note: Fields 5 and 6 are optional.

Ranges change one-sided constraints, defined in the RHS section, into two-sided constraints. The two-sided constraint for row i depends on the range value, r_i, defined in this section. The right-hand side value, b_i , is defined in the RHS section. The two-sided constraints for row i are given in the following table:

Row Type	Lower Constraint	Upper Constraint
G	b_i	$b_i + \lvert r_i \rvert$
L	$b_i - \lvert r_i \rvert$	b_i
E	$b_i + min(0, r_i)$	$b_i + max(0, r_i)$

BOUNDS Section

The optional BOUNDS section defines bounds on the variables. By default, the bounds are $0 \le x_i \le \infty$. The bounds can also be used to indicate that a variable must be an integer.

More than one bound can be set for a single variable. For example, to set $2 \le x_i \le 6$ use a LO bound with value 2 to set $2 \le x_i$ and an UP bound with value 6 to add the condition $x_i \le 6$.

An MPS file can contain more than one bounds set, distinguished by the bound set name.

Field	Contents
1	Bounds type
2	Bounds set name
3	Column name
4	Value for the entry whose set and column are given by fields 2 and 3
5	Column name
6	Value for the entry whose set and column are given by fields 2 and 5

Note: Fields 5 and 6 are optional.

The bound types are as follows. Here b_i are the bound values defined in this section, the x_i are the variables, and I is the set of integers.

Bound Type	Definintion	Formula
LO	Lower bound	$b_i \leq x_i$
UP	Upper bound	$x_i \leq b_i$
FX	Fixed Variable	$x_i = b_i$
FR	Free variable	$-\infty \leq x_i \leq \infty$
MI	Lower bound is minus infinity	$-\infty \leq x_i$
PL	Upper bound is positive infinity	$x_i \leq \infty$
BV	Binary variable (variable must be 0 or 1)	$x_i \in \{0,1\}$
UI	Upper bound and integer	$x_i \leq b_i$ and $x_i \in I$
LI	Lower bound and integer	$b_i \leq x_i$ and $x_i \in I$
SC	Semicontinuous	0 or $b_i \leq x_i$

The bound type names are not case sensitive.

If the bound type is UP or UI and $b_i \leq x_i$ then the lower bound is set to $-\infty$.

ENDATA Section

The ENDATA section ends the MPS file.

Fields

BINARY_VARIABLE

```
static final public int BINARY_VARIABLE
```

Variable must be either 0 or 1.

CONTINUOUS_VARIABLE

```
static final public int CONTINUOUS_VARIABLE
```

Variable is a real number.

INTEGER_VARIABLE

```
static final public int INTEGER_VARIABLE
```

Variable must be an integer.

Constructor

MPSReader

`public MPSReader()`

Methods

getLowerBound

`public double getLowerBound(int iVariable)`

Description

Returns the lower bound for a variable.

Parameter

`iVariable` – is the number of the variable.

getLowerRange

`public double getLowerRange(int iRow)`

Description

Returns the lower range value for a constraint equation.

Parameter

`iRow` – is the row number of the equation.

getName

`public String getName()`

Description

Returns the name of the MPS problem. This is the value of the NAME field.

getNameBounds

`public String getNameBounds()`

Description

Returns the name of the BOUNDS set. An MPS file can contain multiple sets of BOUNDS, but only one is retained by this reader. Returns null if there is no BOUNDS set.

getNameColumn

```
public String getNameColumn(int iColumn) throws
MPSReader.InvalidMPSFileException
```

Description

Returns the name of a constraint column. Constraint column names are also variable names.

Parameter

`iColumn` – is the number of the column.

getNameObjective

`public String getNameObjective()`

Description

Returns the name of the free row containing the objective.

getNameRHS

`public String getNameRHS()`

Description

Returns the name of the RHS section.

getNameRanges

`public String getNameRanges()`

Description

Returns the name of the RANGES set. An MPS file can contain multiple sets of RANGES, but only one is retained by this reader. Returns null if there is no RANGES set.

getNameRow

`public String getNameRow(int iRow)`

Description

Returns the name of a contraint row.

getNumberOfBinaryConstraints

`public int getNumberOfBinaryConstraints()`

Description

Returns the number of binary constraints. An binary constraint is the requirment that a variable be either 0 or 1. Binary constraints are also integer contraints.

getNumberOfColumns

`public int getNumberOfColumns()`

Description

Returns the number of columns in the constraint matrix.

getNumberOfIntegerConstraints

`public int getNumberOfIntegerConstraints()`

Description

Returns the number of integer constraints. An integer constraint is the requirment that a variable be an integer.

getNumberOfNonZeros

```
public int getNumberOfNonZeros()
```

Description

Returns the number of nonzeros in the constraint matrix.

getNumberOfRows

```
public int getNumberOfRows()
```

Description

Returns the number of rows in the constraint matrix.

getObjective

```
public MPSReader.Row getObjective()
```

Description

Returns the objective as a Row.

getObjectiveCoefficients

```
public double[] getObjectiveCoefficients()
```

Description

Returns the coefficents of the objective row.

getRow

```
public MPSReader.Row getRow(int iRow)
```

Description

Returns a row of the constraint matrix or a free row.

Parameter

`iRow` – is the number of the row.

getRowCoefficients

```
public double[] getRowCoefficients(int iRow)
```

Description

Returns the coefficents of a row.

Parameter

`iRow` – is the number of the row.

getTypeVariable

```
public int getTypeVariable(int iVariable)
```

Description

Returns the type of a variable. The variable types are CONTINUOUS_VARIABLE, BINARY_VARIABLE or INTEGER_VARIABLE.

Parameter

`iVariable` – is the number of the variable.

getUpperBound

```
public double getUpperBound(int iVariable)
```

Description

Returns the upper bound for a variable.

Parameter

`iVariable` – is the number of the variable.

getUpperRange

```
public double getUpperRange(int iRow)
```

Description

Returns the upper range value for a constraint equation.

Parameter

`iRow` – is the row number of the equation.

processCommand

```
protected String processCommand(String command, String line) throws
IOException, MPSReader.InvalidMPSFileException
```

Description

Process a section of the MPS file.

Returns

the next line to be processed. This line was read, but was not part of the section being processed.

read

```
public void read(Reader reader) throws IOException,
MPSReader.InvalidMPSFileException
```

Description

Reads and parses the MPS file.

setNameBounds

```
public void setNameBounds(String nameBounds)
```

Description

Sets the name of the BOUNDS set to be used. An MPS file can contain multiple sets of BOUNDS, but only one is retained by this reader. If not set name is set, then the first set in the file is used.

setNameObjective

```
public void setNameObjective(String nameObjective)
```

Description

Sets the name of the free row containing the objective. An MPS file can contain free rows, but only one is retained by this reader as the objective. If not set name is set, then the first free row in the file is used as the objective.

setNameRHS

```
public void setNameRHS(String nameRHS)
```

Description

Sets the name of the RHS set to be used. An MPS file can contain multiple sets of RHS values, but only one is retained by this reader. If not set name is set, then the first set in the file is used.

setNameRanges

```
public void setNameRanges(String nameRanges)
```

Description

Sets the name of the RANGES set to be used. An MPS file can contain multiple sets of RANGES, but only one is retained by this reader. If not set name is set, then the first set in the file is used.

Example: Reading an MPS file.

This example reads the data for a linear programming problem from an MPS file.

```
import com.imsl.io.MPSReader;
import java.io.*;
import java.util.Iterator;

public class MPSReaderEx1 {

    static public void main(String arg[])
            throws IOException, MPSReader.InvalidMPSFileException {
        InputStream stream
                = MPSReaderEx1.class.getResourceAsStream("testprob.mps");
        Reader reader = new InputStreamReader(stream);
        MPSReader mps = new MPSReader();
        mps.read(reader);

        System.out.println("Name   " + mps.getName());
        System.out.println("RHS    " + mps.getNameRHS());
        System.out.println("BOUNDS " + mps.getNameBounds());
        System.out.println("RANGES " + mps.getNameRanges());

        int nRows = mps.getNumberOfRows();
```

```
        System.out.println("NumberOfConstraints " + nRows);
        for (int i = 0; i < nRows; i++) {
            System.out.println("   "
                    + mps.getLowerRange(i)
                    + " <= row[" + i + "] = "
                    + mps.getNameRow(i)
                    + " <= " + mps.getUpperRange(i));
        }

        int nColumns = mps.getNumberOfColumns();
        System.out.println("NumberOfColumns " + nColumns);
        for (int i = 0; i < nColumns; i++) {
            System.out.println("   "
                    + mps.getLowerBound(i)
                    + " <= var[" + i + "] = "
                    + mps.getNameColumn(i)
                    + " <= " + mps.getUpperBound(i));
        }

        System.out.println("NumberOfNonZeros " + mps.getNumberOfNonZeros());
        for (int iRow = 0; iRow < nRows; iRow++) {
            System.out.println("       row " + mps.getNameRow(iRow));
            Iterator iter = mps.getRow(iRow).iterator();
            while (iter.hasNext()) {
                MPSReader.Element elem = (MPSReader.Element) iter.next();
                int iColumn = elem.getColumn();
                String nameColumn = mps.getNameColumn(iColumn);
                System.out.println("          " + nameColumn
                        + ": " + elem.getValue());
            }
        }

        reader.close();
        stream.close();
    }
}
```

Output

```
Name   TESTPROB
RHS    RHS1
BOUNDS BND1
RANGES null
NumberOfConstraints 3
   -Infinity <= row[0] = LIM1 <= 5.0
   10.0 <= row[1] = LIM2 <= Infinity
   7.0 <= row[2] = MYEQN <= 7.0
NumberOfColumns 3
   0.0 <= var[0] = XONE <= 4.0
   -1.0 <= var[1] = YTWO <= 1.0
   0.0 <= var[2] = ZTHREE <= Infinity
NumberOfNonZeros 6
       row LIM1
          XONE: 1.0
          YTWO: 1.0
```

```
row LIM2
   XONE: 1.0
   ZTHREE: 1.0
row MYEQN
   YTWO: -1.0
   ZTHREE: 1.0
```

MPSReader.InvalidMPSFileException class

`static public class com.imsl.io.MPSReader.InvalidMPSFileException extends com.imsl.IMSLException`

The MPS file is invalid.

Constructors

MPSReader.InvalidMPSFileException

`public MPSReader.InvalidMPSFileException(String message)`

Description

Constructs a `InvalidMPSFileException` object.

Parameter

`message` – a `String` containing the error message

MPSReader.InvalidMPSFileException

`public MPSReader.InvalidMPSFileException(String key, Object[] arguments)`

Description

Constructs a `InvalidMPSFileException` object.

Parameters

`key` – a `String` containing the error message

`arguments` – an `Object` array containing arguments used within the error message string

MPSReader.Row class

`public class com.imsl.io.MPSReader.Row implements Serializable`

A row either in the constraint matrix or a free row.

Methods

getCoefficients

```
public double[] getCoefficients()
```

Description

Returns the coeffients of this row as a dense array.

getName

```
public String getName()
```

Description

Returns the name of this row.

getNumberOfNonZeros

```
public int getNumberOfNonZeros()
```

Description

Returns the number of nonzero elements in this row.

iterator

```
public Iterator iterator()
```

Description

Returns an iterator over the elements in this row. This is used to retrieve the coeffients in a sparse form.

MPSReader.Element class

```
static public class com.imsl.io.MPSReader.Element implements Serializable
```

An element in the sparse contraint matrix.

Methods

getColumn

```
public int getColumn()
```

Description

Returns the column index.

getValue

```
public double getValue()
```

Description

Returns the value of the element.

Chapter 25: Finance

Types

interface BasisPart 1472
class Bond 1473
class DayCountBasis 1527
class Finance 1530

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.*

BasisPart interface

`public interface com.imsl.finance.BasisPart`

Component of `com.imsl.finance.DayCountBasis` (p. 1527) . The day count basis consists of a month basis and a yearly basis. Each of these components implements this interface.

Methods

daysBetween

`public int daysBetween(GregorianCalendar date1, 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

`public double daysInPeriod(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` – an `int` containing 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

```
public int getDaysInYear(GregorianCalendar date)
```

Description

Returns the number of days in the year.

Parameter

`date` – a `GregorianCalendar` date.

Returns

an `int` which specifies the number of days in the year

Bond class

```
public class com.imsl.finance.Bond
```

Collection of bond functions.

Definitions

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 acquires a bond.

maturity is the day a bond's principal is repaid.

Discount Bonds

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.

Treasury Bills

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)

Interest-Paying Bonds

Most bonds pay interest periodically. The interest-paying bond methods all have *settlement*, *maturity*, *basis*, and *frequency* as arguments. Again suppressing 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.

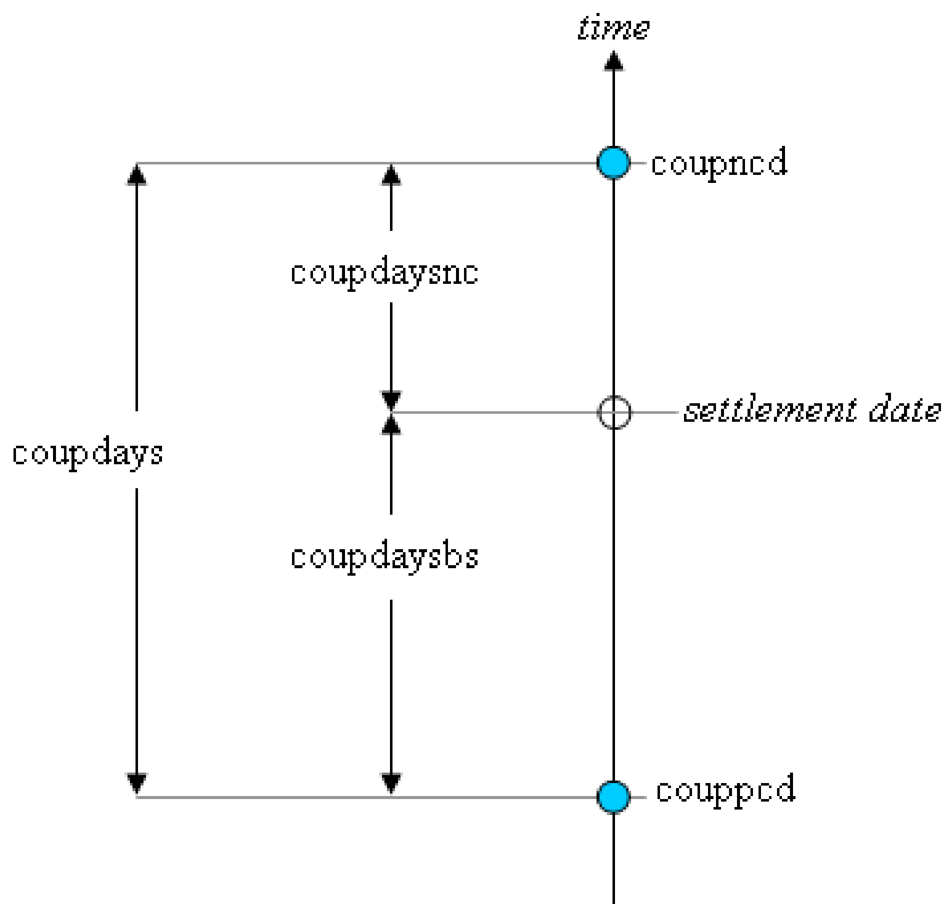
Interest-Paying Bonds with Odd Coupon Periods

An odd coupon period is one that is either shorter or longer than the regular coupon period. Odd coupon periods typically occur in the first or the last payment period. The yield and price functions accept two additional arguments in order to make the proper adjustments for odd coupon periods: `issueDate` and `firstCouponDate` or `lastCouponDate`.

- oddFirstPrice = price(settlement, maturity, issueDate, firstCouponDate, rate, yield, redemption, frequency, basis)
- oddLastPrice = price(settlement, maturity, lastCouponDate, rate, yield, redemption, frequency, basis)
- oddFirstYield = yield(settlement, maturity, issueDate, firstCouponDate, rate, price, redemption, frequency, basis)
- oddLastYield = yield(settlement, maturity, lastCouponDate, rate, price, redemption, frequency, basis)

Coupon days

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

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

Fields

ANNUAL

```
static final public int ANNUAL
```

Coupon payments are made annually.

BIMONTHLY

```
static final public int BIMONTHLY
```

Coupon payments are made bimonthly (6 times per year).

MONTHLY

```
static final public int MONTHLY
```

Coupon payments are made monthly.

QUARTERLY

```
static final public int QUARTERLY
```

Coupon payments are made quarterly.

SEMIANNUAL

```
static final public int SEMIANNUAL
```

Coupon payments are made semiannually (twice per year).

Methods

accrint

```
static public double accrint(GregorianCalendar issue, GregorianCalendar
firstCoupon, 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. NL_i represents the length of the normal *i*th 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

```
static public double accrintm(GregorianCalendar issue, 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

```
static public double amordegrc(double cost, GregorianCalendar issue,
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

```
static public double amorlinc(double cost, GregorianCalendar issue,
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

```
static public double convexity(GregorianCalendar settlement, 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 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 convexity for a security

coupdaybs

```
static public int coupdaybs(GregorianCalendar settlement, 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

```
static public double coupdays(GregorianCalendar settlement, 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

a `double` which specifies the number of days in the coupon period that contains the settlement date

coupdaysnc

```
static public int coupdaysnc(GregorianCalendar settlement, 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

```
static public GregorianCalendar coupncd(GregorianCalendar settlement,
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

a `GregorianCalendar` which specifies the next coupon date after the settlement date

coupnum

```
static public int coupnum(GregorianCalendar settlement, 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

```
static public GregorianCalendar couppcd(GregorianCalendar settlement,
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

a `GregorianCalendar` which specifies the previous coupon date before the settlement date

---

disc

```
static public double disc(GregorianCalendar settlement, 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

```
static public double duration(GregorianCalendar settlement, GregorianCalendar
maturity, double coupon, double yield, int frequency, DayCountBasis basis)
```

Description

Returns the Macaulay duration of a security where the security has periodic interest payments. The Macaulay 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:

$$\left(\frac{\frac{\left(N-1+\frac{DSC}{E}\right)\times 100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}}+\sum_{k=1}^{N}\left(\left(\frac{100\times coupon}{freq\times\left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right)\times\left(k-1+\frac{DSC}{E}\right)\right)}{\frac{100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}}+\sum_{k=1}^{N}\left(\frac{100\times coupon}{freq\times\left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right)}\right)\times\frac{1}{freq}$$

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

```
static public double intrate(GregorianCalendar settlement, 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

```
static public double mduration(GregorianCalendar settlement, GregorianCalendar
maturity, double coupon, double yield, int frequency, DayCountBasis basis)
```

Description

Returns the modified Macaulay 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 Macaulay duration for a security with an assumed par value of $100

price

```
static public double price(GregorianCalendar settlement, 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

price

```
static public double price(GregorianCalendar settlement, GregorianCalendar
maturity, GregorianCalendar lastCoupon, double rate, double yield, double
redemption, int frequency, DayCountBasis basis)
```

Description

Returns the price of an odd last period coupon bond, given its yield.

In the case of a short last period with one coupon period or less to redemption, the following formula is used:

$$P = \left[\frac{redemption + \left(\frac{100*rate}{frequency} * \frac{DLC}{E}\right)}{1 + \left(\frac{DSR}{E} * \frac{yield}{frequency}\right)}\right] - \left[\frac{A}{E} * \frac{100*rate}{frequency}\right]$$

where

Variable	Description
A	Number of days from last coupon date before redemption to settlement date.
DLC	Number of days from last coupon date before redemption to redemption.
DSR	Number of days from settlement date to redemption date.
E	Number of days in the quasi-coupon period.

In the case of a short last period with more than one coupon period to redemption, the following formula is used:

$$\begin{aligned} P \quad = \quad & \left[\frac{redemption}{\left(1 + \frac{yield}{frequency}\right)^{N-1+\frac{DSC}{E}+\frac{DLC}{NLL}}}\right] + \left[\sum_{K=1}^{N} \frac{100 * \frac{rate}{frequency}}{\left(1 + \frac{yield}{frequency}\right)^{K-1+\frac{DSC}{E}}}\right] \\ + \quad & \left[\frac{100 * \frac{rate}{frequency} * \frac{DLC}{NLL}}{\left(1 + \frac{yield}{frequency}\right)^{N-1+\frac{DSC}{E}+\frac{DLC}{NLL}}}\right] - \left[100 * \frac{rate}{frequency} * \frac{A}{E}\right] \end{aligned}$$

where

Variable	Description
A	Number of days from beginning of coupon period to settlement date.
DLC	Number of days from last coupon date before redemption to redemption.
DSC	Number of days from settlement date to next coupon date.
E	Number of days in coupon period in which the settlement date falls.
N	Number of coupons payable between settlement date and last coupon date before redemption.
NLL	Number of days in the full quasi-coupon period in which the odd last period falls.

In the case of a long last period with one coupon period or less to redemption, the following formula is used:

$$P = \left[\frac{redemption + \left(\left(\sum_{i=1}^{NCL} \frac{DLC_i}{NLL_i} \right) * \frac{100*rate}{frequency} \right)}{1 + \left(\sum_{i=1}^{NCL} \frac{DSC_i}{NLL_i} \right) * \frac{yield}{frequency}} \right] - \left[\left(\sum_{i=1}^{NCL} \frac{A_i}{NLL_i} \right) * \frac{100 * rate}{frequency} \right]$$

where

Variable	Description
A_i	Number of accrued days for the ith quasi-coupon period within the odd period counting forward from the last interest date before redemption.
DLC_i	Number of days in the ith quasi-coupon period as delimited by the length of the actual coupon period.
DSC_i	Number of days from settlement date (or beginning of quasi-coupon period) to next quasi-coupon within odd period (or to redemption date) for the ith quasi-coupon period.
NCL	Number of quasi-coupon periods that fit in odd period.
NLL_i	Normal length in days of the full i^{th} quasi-coupon period within odd last period.

In the case of a long first period with more than one coupon period to redemption, the following formula is used:

$$\begin{aligned} P \quad = & \quad \left[\frac{redemption}{\left(1 + \frac{yield}{frequency}\right)^{N-1+\frac{DSC}{E}+Nqf+\frac{DLQ}{LQL}}} \right] + \left[\sum_{K=1}^{N} \frac{100 * \frac{rate}{frequency}}{\left(1 + \frac{yield}{frequency}\right)^{K-1+\frac{DSC}{E}}} \right] \\ & + \quad \left[\frac{100 * \frac{rate}{frequency} * \left(\sum_{i=1}^{NCL} \frac{DLC_i}{NLL_i} \right)}{\left(1 + \frac{yield}{frequency}\right)^{N-1+\frac{DSC}{E}Nqf+\frac{DLQ}{LQL}}} \right] - \left[100 * \frac{rate}{frequency} * \frac{A}{E} \right] \end{aligned}$$

where

Variable	Description
A	Number of days from beginning of coupon period to settlement date.
DLC_i	Number of days from last coupon date before redemption to first quasi-coupon or number of days in quasi-coupon.
DLQ	Number of days counted in the last quasi-coupon period within the last odd period.
DSC	Number of days from settlement date to next coupon date.
E	Number of days in coupon period in which the settlement falls.
LQL	Normal length in days of the last quasi-coupon period within the odd last period.
N	Number of coupons payable between settlement date and last coupon date before redemption.
NCL	Number of quasi-coupon periods that fit in odd period.
NLL_i	Normal length in days of the full i^{th} quasi-coupon period within odd last period.
Nql	Number of whole quasi-coupon periods between last coupon date before redemption and the redemption date.

Parameters

`settlement` – a `GregorianCalendar` settlement date of the security

`maturity` – a `GregorianCalendar` maturity date of the security

`lastCoupon` – a `GregorianCalendar` last interest 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` that indicates the bond price.

price

```
static public double price(GregorianCalendar settlement, GregorianCalendar
maturity, GregorianCalendar issueDate, GregorianCalendar firstCoupon, double
rate, double yield, double redemption, int frequency, DayCountBasis basis)
```

Description

Returns the price of an odd first period, coupon bond, given its yield.

In the case of a short first period, the following formula is used:

$$P = \left[\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{N-1+\frac{DSC}{E}}}\right] + \left[\frac{100 * \frac{rate}{frequency} * \frac{DFC}{E}}{\left(1+\frac{yield}{frequency}\right)^{\frac{DSC}{E}}}\right] + \left[\sum_{K=2}^{N} \frac{100 * \frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{K-1+\frac{DSC}{E}}}\right] - \left[100 * \frac{rate}{frequency} * \frac{A}{E}\right]$$

where

Variable	Description
A	Number of days from the beginning of coupon period to the settlement date (accrued days).
DFC	Number of days from the beginning of odd first coupon period (`issueDate`) to the first coupon date.
DSC	Number of days from the settlement date to first coupon date.
E	Number of days in the quasi-coupon period in which the settlement date falls.
N	Number of coupons payable between settlement date and redemption date.

In the case of a long first period, the following formula is used:

$$P = \left[\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{N-Nqf+\frac{DSC}{E}}}\right] + \left[\frac{100 * \frac{rate}{frequency} * \left[\sum_{i=1}^{NCF} \frac{DFC_i}{NLF_i}\right]}{\left(1+\frac{yield}{frequency}\right)^{Nqf+\frac{DSC}{E}}}\right] + \left[\sum_{K=1}^{N} \frac{100 * \frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{K+Nqf+\frac{DSC}{E}}}\right] - \left[100 * \frac{rate}{frequency} * \left[\sum_{i=1}^{NCF} \frac{A_i}{NLF_i}\right]\right]$$

where

Variable	Description
A_i	Number of accrued days for the ith quasi-coupon period within the odd period.
DFC_i	Number of days from the `issueDate` to the first quasi-coupon or number of days in the the quasi-coupon.
DSC	Number of days from the settlement date to the next coupon or quasi-coupon date.
E	Number of days in the quasi-coupon or coupon period in which the settlement date falls.
N	Number of coupon periods between the first real coupon date and redemption date.
NCF	Number of quasi-coupon periods that fall in the odd period.
NLF_i	Number of days in the full i^{th} quasi-coupon period within the odd period.
Nqf	Number of whole quasi-coupon periods between `settlement` and `firstCoupon`.

Parameters

`settlement` – a `GregorianCalendar` settlement date of the security

`maturity` – a `GregorianCalendar` maturity date of the security

`issueDate` – a `GregorianCalendar` issue date of the security

`firstCoupon` – a `GregorianCalendar` first coupon 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` that indicates the bond price.

pricedisc

```
static public double pricedisc(GregorianCalendar settlement, 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

```
static public double pricemat(GregorianCalendar settlement, GregorianCalendar
maturity, 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

```
static public double priceyield(GregorianCalendar settlement, 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

```
static public double received(GregorianCalendar settlement, 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

```
static public double tbilleq(GregorianCalendar settlement, 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

```
static public double tbillprice(GregorianCalendar settlement, 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

`static public double tbillyield(GregorianCalendar settlement, 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

`static public double yearfrac(GregorianCalendar start, 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

```
static public double yield(GregorianCalendar settlement, 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

yield

```
static public double yield(GregorianCalendar settlement, GregorianCalendar
maturity, GregorianCalendar lastCoupon, double rate, double price, double
redemption, int frequency, DayCountBasis basis)
```

Description

Returns the yield of a security with an odd last coupon period that pays periodic interest.

The yield is determined by finding the zero of the function desired price - computed price, where computed price is the output of com.imsl.finance.Bond.price (p. 1485) .

Parameters

settlement – a GregorianCalendar settlement date of the security

maturity – a GregorianCalendar maturity date of the security

lastCoupon – a GregorianCalendar last coupon 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

yield

```
static public double yield(GregorianCalendar settlement, GregorianCalendar
maturity, GregorianCalendar issueDate, GregorianCalendar firstCoupon, double
rate, double price, double redemption, int frequency, DayCountBasis basis)
```

Description

Returns the yield of a security with an odd first coupon period that pays periodic interest.

The yield is determined by finding the zero of the function desired price - computed price, where computed price is the output of com.imsl.finance.Bond.price (p. 1488) .

Parameters

`settlement` – a `GregorianCalendar` settlement date of the security

`maturity` – a `GregorianCalendar` maturity date of the security

`issueDate` – a `GregorianCalendar` issueDate date of the security

`firstCoupon` – a `GregorianCalendar` first coupon 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

```
static public double yielddisc(GregorianCalendar settlement, 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

```
static public double yieldmat(GregorianCalendar settlement, GregorianCalendar
maturity, 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` which specifies 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.

Output

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
```

Price of a Discounted Security

The price of a discounted 1 year bond is returned in this example.

priceyieldEx1

```
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);
    }
}
```

Output

```
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 0.9166666666666667
```

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 0.06999999682843033
```

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
```

Example: Bond Price - Odd Short First Coupon

This example calculates the price of an odd short first coupon.

Settlement	11/11/1992
Maturity	03/01/2005
Issue date	10/15/1992
First Coupon	03/01/1993
Rate	0.0785
Yield	0.0625
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.BasisActualActual

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;
```

```
public class OddShortFirstCoupPriceEx {

    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("11/11/1992");
        GregorianCalendar maturity = parse("03/01/2005");
        GregorianCalendar issueDate = parse("10/15/1992");
        GregorianCalendar firstCoupon = parse("03/01/1993");
        double price = Bond.price(settlement, maturity, issueDate,
                firstCoupon, 0.0785, .0625, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.BasisActualActual);
        System.out.println("The price is " + price);
    }
}
```

Output

```
The price is 113.59771747407882
```

Example: Bond Price - Odd Long First Coupon

This example calculates the price of an odd long first coupon.

Settlement	11/11/1992
Maturity	03/01/2005
Issue date	06/15/1992
First Coupon	03/01/1993
Rate	0.0935
Yield	0.0775
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.BasisActualActual

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddLongFirstCoupPriceEx {

    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("11/11/1992");
        GregorianCalendar maturity = parse("03/01/2005");
        GregorianCalendar issue = parse("06/15/1992");
        GregorianCalendar firstCoupon = parse("03/01/1993");
        double price = Bond.price(settlement, maturity, issue, firstCoupon,
                0.0935, .0775, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.BasisActualActual);
        System.out.println("The price is " + price);
    }
}
```

Output

```
The price is 112.47810623329781
```

Example: Bond Price - Odd Short Last Coupon

This example calculates the price of an odd short last coupon with one or less coupon period to redemption.

Settlement	02/07/1993
Maturity	08/01/1993
Last Coupon	02/04/1993
Rate	0.065
Yield	0.0535
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddShortLastOneCoupPriceEx {

    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("02/07/1993");
        GregorianCalendar maturity = parse("08/01/1993");
        GregorianCalendar lastCoupon = parse("02/04/1993");
        double price = Bond.price(settlement, maturity, lastCoupon, 0.065,
                0.0535, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The price is " + price);
    }
}
```

Output

```
The price is 100.54045734737038
```

Example: Bond Price - Odd Short Last Coupon

This example calculates the price of an odd short last coupon with multiple coupon period to redemption.

Settlement	02/25/1993
Maturity	06/01/2005
Last Coupon	12/15/2004
Rate	0.06875
Yield	0.0725
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddShortLastMultiCoupPriceEx {

    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("02/25/1993");
        GregorianCalendar maturity = parse("06/01/2005");
        GregorianCalendar lastCoupon = parse("12/15/2004");
        double price = Bond.price(settlement, maturity, lastCoupon, 0.06875,
```

```
                0.0725, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The price is " + price);
    }
}
```

Output

```
The price is 96.97412041120907
```

Example: Bond Price - Odd Long Last Coupon

This example calculates the price of an odd long last coupon with one coupon period to redemption.

Settlement	02/07/1993
Maturity	06/15/1993
Last Coupon	10/15/1992
Rate	0.0375
Yield	0.0405
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddLongLastOneCoupPriceEx {

    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("02/07/1993");
        GregorianCalendar maturity = parse("06/15/1993");
        GregorianCalendar lastCoupon = parse("10/15/1992");
        double price = Bond.price(settlement, maturity, lastCoupon, 0.0375,
                0.0405, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The price is " + price);
    }
}
```

Output

```
The price is 99.87828601472134
```

Example: Bond Price - Odd Long Last Coupon

This example calculates the price of an odd long last coupon with multiple coupon periods to redemption.

Settlement	02/25/1993
Maturity	09/15/2004
Last Coupon	12/15/2003
Rate	0.06875
Yield	0.0893
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddLongLastMultiCoupPriceEx {

    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("02/25/1993");
        GregorianCalendar maturity = parse("09/15/2004");
        GregorianCalendar lastCoupon = parse("12/15/2003");
        double price = Bond.price(settlement, maturity, lastCoupon, 0.06875,
                0.0893, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The price is " + price);
    }
}
```

Output

```
The price is 85.33445170277066
```

Example: Bond Yield - Odd Short First Coupon

This example calculates the yield of an odd short first coupon.

Settlement	11/11/1992
Maturity	03/01/2005
Issue date	10/15/1992
First Coupon	03/01/1993
Rate	0.0785
Price	113.597717
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.BasisActualActual

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddShortFirstCoupYieldEx {

    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("11/11/1992");
        GregorianCalendar maturity = parse("03/01/2005");
        GregorianCalendar issueDate = parse("10/15/1992");
        GregorianCalendar firstCoupon = parse("03/01/1993");
        double yield = Bond.yield(settlement, maturity, issueDate,
                firstCoupon, 0.0785, 113.597717, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.BasisActualActual);
        System.out.println("The yield is " + yield);
    }
}
```

Output

```
The yield is 0.06250000051314245
```

Example: Bond Yield - Odd Long First Coupon

This example calculates the yield of an odd long first coupon.

Settlement	11/11/1992
Maturity	03/01/2005
Issue date	06/15/1992
First Coupon	03/01/1993
Rate	0.0935
Price	112.478106
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.BasisActualActual

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddLongFirstCoupYieldEx {

    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("11/11/1992");
        GregorianCalendar maturity = parse("03/01/2005");
        GregorianCalendar issue = parse("06/15/1992");
        GregorianCalendar firstCoupon = parse("03/01/1993");
        double yield = Bond.yield(settlement, maturity, issue, firstCoupon,
                0.0935, 112.478106, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.BasisActualActual);
        System.out.println("The yield is " + yield);
    }
}
```

Output

```
The yield is 0.07750000027421643
```

Example: Bond Yield - Odd Short Last Coupon

This example calculates the yield of an odd short last coupon with one or less coupon period to redemption.

Settlement	02/07/1993
Maturity	08/01/1993
Last Coupon	02/04/1993
Rate	0.065
Price	100.540457
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddShortLastOneCoupYieldEx {

    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("02/07/1993");
        GregorianCalendar maturity = parse("08/01/1993");
        GregorianCalendar lastCoupon = parse("02/04/1993");
        double yield = Bond.yield(settlement, maturity, lastCoupon, 0.065,
                100.540457, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The yield is " + yield);
    }
}
```

Output

```
The yield is 0.05350000732923516
```

Example: Bond Yield - Odd Short Last Coupon

This example calculates the yield of an odd short last coupon with multiple coupon period to redemption.

Settlement	02/25/1993
Maturity	06/01/2005
Last Coupon	12/15/2004
Rate	0.06875
Price	96.974120
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddShortLastMultiCoupYieldEx {

    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("02/25/1993");
        GregorianCalendar maturity = parse("06/01/2005");
        GregorianCalendar lastCoupon = parse("12/15/2004");
        double yield = Bond.yield(settlement, maturity, lastCoupon, 0.06875,
                96.974120, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The yield is " + yield);
    }
}
```

Output

```
The yield is 0.07250000052203028
```

Example: Bond Yield - Odd Long Last Coupon

This example calculates the yield of an odd long last coupon with one coupon period to redemption.

Settlement	02/07/1993
Maturity	06/15/1993
Last Coupon	10/15/1992
Rate	0.0375
Price	99.878286
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddLongLastOneCoupYieldEx {

    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("02/07/1993");
        GregorianCalendar maturity = parse("06/15/1993");
        GregorianCalendar lastCoupon = parse("10/15/1992");
        double yield = Bond.yield(settlement, maturity, lastCoupon, 0.0375,
                99.878286, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The yield is " + yield);
    }
}
```

Output

```
The yield is 0.04050000041565726
```

Example: Bond Yield - Odd Long Last Coupon

This example calculates the yield of an odd long last coupon with multiple coupon periods to redemption.

Settlement	02/25/1993
Maturity	09/15/2004
Last Coupon	12/15/2003
Rate	0.06875
Price	85.334452
Redemption Value	100.0
Payment Frequency	Bond.SEMIANNUAL
Day Count Basis	DayCountBasis.Basis30e360

```
import com.imsl.finance.*;
import java.text.*;
import java.util.*;

public class OddLongLastMultiCoupYieldEx {

    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("02/25/1993");
        GregorianCalendar maturity = parse("09/15/2004");
        GregorianCalendar lastCoupon = parse("12/15/2003");
        double yield = Bond.yield(settlement, maturity, lastCoupon, 0.06875,
                85.334452, 100.0, Bond.SEMIANNUAL,
                DayCountBasis.Basis30e360);
        System.out.println("The yield is " + yield);
    }
}
```

Output

```
The yield is 0.0892999995362467
```

DayCountBasis class

```
public class com.imsl.finance.DayCountBasis
```

The Day Count Basis. Rules for computing the number of 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.

Fields

Basis30e360

```
static final public DayCountBasis Basis30e360
```

Computations are based on the assumption of 30 days per month and 360 days per year.

BasisActual360

```
static final public 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.

BasisActual365

```
static final public 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.

BasisActualActual

```
static final public DayCountBasis BasisActualActual
```

Computations are based on the actual calendar.

BasisNASD

```
static final public DayCountBasis BasisNASD
```

Computations are based on the assumption of 30 days per month and 360 days per year.

BasisPart30E360

```
static final public BasisPart BasisPart30E360
```

Computations are 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.

BasisPart365

```
static final public BasisPart BasisPart365
```

Computations are based on the assumption of 365 days per year.

BasisPartActual

```
static final public BasisPart BasisPartActual
```

Computations are are based on the actual calendar.

BasisPartNASD

```
static final public BasisPart BasisPartNASD
```

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` – a `BasisPart` that contains the month basis

`yearBasis` – a `BasisPart` that contains 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`

setEOM

```
public void setEOM(boolean EOM)
```

Description

Specifies whether to use the End-Of-Month rule.

If a security follows the End-Of-Month rule:

1. - If Date2 (day) is the last day of February
   - and Date1 (day) is the last day of February
   - change Date2 (day) to 30
2. - If Date1 (day) is the last day of February
   - change Date1 (day) to 30

The EOM rule is only used in day count calculations of the basis type, 30/360 (NASD Superclass).

Parameter

`EOM` – a `boolean` which indicates if the End-Of-Month rule is to be used.
Default: EOM = `true`.

Finance class

`public class com.imsl.finance.Finance`

Collection of finance functions.

Fields

AT_BEGINNING_OF_PERIOD

`static final public int AT_BEGINNING_OF_PERIOD`

Flag used to indicate that payment is made at the beginning of each period.

AT_END_OF_PERIOD

`static final public int AT_END_OF_PERIOD`

Flag used to indicate that payment is made at the end of each period.

Methods

cumipmt

`static public 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 ith 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 `com.imsl.finance.Finance.AT_END_OF_PERIOD` (p. 1530) or `com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD` (p. 1530)

Returns

a `double`, the cumulative interest paid between the first period and the last period

cumprinc

```
static public 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 ith 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 `com.imsl.finance.Finance.AT_END_OF_PERIOD` (p. 1530) or `com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD` (p. 1530) .

Returns

a `double`, the cumulative principal paid between the first period and the last period

db

```
static public 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,

$$(\text{cost} - \text{total depreciation from prior periods}) \times rate$$

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

```
static public 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

```
static public 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

```
static public 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` – an `int`, the denominator

Returns

a `double`, a dollar price expressed as a fraction

effect

```
static public 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

```
static public 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 $rate \neq 0$,

$$pv(1+rate)^{nper} + pmt\,[1+rate\,(when)]\,\frac{(1+rate)^{nper}-1}{rate} + fv = 0$$

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 `com.imsl.finance.Finance.AT_END_OF_PERIOD` (p. 1530) or `com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD` (p. 1530)

Returns

a `double`, the future value of an investment

fvschedule

```
static public 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:

$$\sum_{i=1}^{count} (principal \times schedule_i)$$

where $schedule_i$ = interest rate at the ith 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

```
static public 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 `com.imsl.finance.Finance.AT_END_OF_PERIOD` (p. 1530) or `com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD` (p. 1530)

Returns

a `double`, the interest payment for a given period for an investment

irr

```
static public 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`.

Parameter

`pmt` – a `double` array which contains cash flow values which occur at regular intervals

Returns

a `double`, the internal rate of return

irr

```
static public 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

```
static public 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

```
static public 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:

$$\left[(1+effectiveRate)^{\frac{1}{nper}}-1\right]\times nper$$

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

```
static public 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 $rate \neq 0$,

$$pv(1+rate)^{nper} + pmt\,[1+rate\,(when)]\,\frac{(1+rate)^{nper}-1}{rate} + fv = 0$$

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 `com.imsl.finance.Finance.AT_END_OF_PERIOD` (p. 1530) or `com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD` (p. 1530)

Returns

an `int`, the number of periods for an investment

npv

```
static public 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 ith cash flow, and count is `value.length`.

Parameters

`rate` – a `double`, the interest rate per period. It must not be -1.

`value` – a `double` array of equally-spaced cash flows

Returns

a `double`, the net present value of the investment

pmt

```
static public 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 com.imsl.finance.Finance.AT_END_OF_PERIOD (p. 1530) or com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD (p. 1530)

Returns

a double, the interest payment for a given period for an investment

ppmt

```
static public 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 ith period, $interest_i$ is calculated from ipmt for the ith 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 com.imsl.finance.Finance.AT_END_OF_PERIOD (p. 1530) or com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD (p. 1530)

Returns

a `double`, the payment on the principal for a given period

pv

```
static public 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 `com.imsl.finance.Finance.AT_END_OF_PERIOD` (p. 1530) or `com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD` (p. 1530)

Returns

a `double`, the present value of the investment

rate

```
static public 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 com.imsl.finance.Finance.AT_END_OF_PERIOD (p. 1530) or com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD (p. 1530)

Returns

a double, the interest rate per period of an annuity

rate

```
static public 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 $rate \neq 0$,

$$pv(1+rate)^{nper} + pmt\left[1+rate\,(when)\right]\frac{(1+rate)^{nper}-1}{rate} + fv = 0$$

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 com.imsl.finance.Finance.AT_END_OF_PERIOD (p. 1530) or com.imsl.finance.Finance.AT_BEGINNING_OF_PERIOD (p. 1530)

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

```
static public 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

```
static public 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

```
static public 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 ith 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

```
static public double xirr(double[] pmt, 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 ith payment date. d_1 represents the 1st payment date. *value* represents 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 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

```
static public double xirr(double[] pmt, 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 ith payment date. d_1 represents the 1st payment date. *value* represents the ith 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

```
static public double xnpv(double rate, double[] value, 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 ith payment date, d_1 represents the first payment date, $value_i$ represents the ith 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.*;

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.*;
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 26: Chart 2D

Types

class Chart 1562
class AbstractChartNode 1567
class ChartNode 1586
class Background 1608
class ChartTitle 1609
class Legend 1609
class Annotation 1610
class Grid 1612
class Axis 1613
class AxisXY 1615
class Axis1D 1617
class AxisLabel 1622
class AxisLine 1623
class AxisTitle 1623
class AxisUnit 1624
class MajorTick 1625
class MinorTick 1625
interface Transform 1626
class TransformDate 1627
class AxisR 1628
class AxisRLabel 1630
class AxisRLine 1631
class AxisRMajorTick 1631
class AxisTheta 1632
class GridPolar 1633
class Data 1634
interface ChartFunction 1645
class ChartSpline 1645
class Text 1646
class ToolTip 1648
class FillPaint 1650

class Draw 1653
class JFrameChart 1664
class JPanelChart 1665
class DrawPick 1667
class PickEvent 1673
interface PickListener 1674
class JspBean 1675
class ChartServlet 1678
class DrawMap 1679
class BoxPlot 1685
class Contour 1695
class ErrorBar 1703
class HighLowClose 1708
class Candlestick 1714
class CandlestickItem 1716
class SplineData 1717
class Bar 1720
class BarItem 1726
class BarSet 1727
class Pie 1728
class PieSlice 1732
class Dendrogram 1733
class Polar 1741
class Heatmap 1745
class Treemap 1755
interface Colormap 1764
class ChartXML 1766

Chart class

`public class com.imsl.chart.Chart extends com.imsl.chart.ChartNode implements Cloneable, java.awt.print.Printable`

The root node of the chart tree.

This chart node creates the following child nodes: `com.imsl.chart.Background` (p. 1608) , `com.imsl.chart.ChartTitle` (p. 1609) and `com.imsl.chart.Legend` (p. 1609) .

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(Component component)
```

Description

This is the root of our tree, it has no parent. This creates the Chart with the named component

Parameter

`component` – the `Component` that contains the chart.

Chart

```
public Chart(Image image)
```

Description

This is the root of our tree, it has no parent. This creates the Chart drawn into the image.

Parameter

`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`. This method is intended to be called from within the `paint` method of classes which explicitly paint their own child chart nodes. The child chart nodes to be included in the legend must be added to the legend during each call to `paint` of the parent chart node.

Typical users of the chart library do not need to call this routine. This method is for use by those writing new charting classes.

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(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(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 Object clone()
```

Description

Returns a clone of the graphics tree.

Returns

an `Object` which is a clone of this graphics tree

clone

```
protected Object clone(Map hashClonedNode)
```

Description

Returns a clone of this node.

Parameter

`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()
```

paint

```
public void paint(Draw draw)
```

Description

Paints this node and all of its children.

Parameter

`draw` – a `Draw` object to be painted

paint

```
public void paint(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.

Parameter

`g` – `Graphics` object to be painted

paintChart

```
public void paintChart(Graphics graphics)
```

Description

Draw the chart using the given Graphics object.

Parameter

`graphics` – is the object for which the chart is to be drawn.

paintImage

```
public 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(MouseEvent event)
```

Description

Fire the PickListeners for the nodes hit by the event.

Parameter

`event` – `MouseEvent` whose position determines which nodes have been selected

print

```
public int print(Graphics graphics, PageFormat pageFormat, int param) throws
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(Component component)
```

Description

Sets the Component for this chart. Also registers MouseListeners or MouseMotionListeners that could not be added previously.

update

```
public void update(Graphics g)
```

writePNG

```
public void writePNG(OutputStream os, int width, int height) throws IOException
```

Description

Writes the chart as an PNG file. PNG (Portable Network Graphics) is a lossless bitmap format.

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.

Exceptions

`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(Writer writer, boolean useCSS) throws IOException
```

Description

Writes the chart as an SVG file. This method requires the Apache Batik library.

Parameters

`writer` – is the output character stream

`useCSS` – is true if the CSS style attribute is to be used

Exceptions

`java.io.IOException` if there is a problem writing the file.

`java.lang.NoClassDefFoundError` if the Batik library cannot be found.

AbstractChartNode class

`abstract public class com.imsl.chart.AbstractChartNode implements Serializable, Cloneable`

The base class of all of the nodes in both the 2D and 3D chart trees.

Fields

AUTOSCALE_DATA

`static final public int AUTOSCALE_DATA`

Flag used to indicate that autoscaling is to be done by scanning the data nodes.

AUTOSCALE_DENSITY

`static final public int AUTOSCALE_DENSITY`

Flag used to indicate that autoscaling is to adjust the "Density" attribute. This applies only to time axes.

AUTOSCALE_NUMBER

`static final public int AUTOSCALE_NUMBER`

Flag used to indicate that autoscaling is to adjust the "Number" attribute.

AUTOSCALE_OFF

`static final public int AUTOSCALE_OFF`

Flag used to indicate that autoscaling is turned off.

AUTOSCALE_WINDOW

`static final public int AUTOSCALE_WINDOW`

Flag used to indicate that autoscaling is to be done by using the "Window" attribute.

AXIS_X

`static final public int AXIS_X`

Flag to indicate x-axis.

AXIS_Y

`static final public int AXIS_Y`

Flag to indicate y-axis.

AXIS_Z
```
static final public int AXIS_Z
```

Flag to indicate z-axis.

LABEL_TYPE_NONE
```
static final public int LABEL_TYPE_NONE
```

Flag used to indicate the an element is not to be labeled.

LABEL_TYPE_TITLE
```
static final public int LABEL_TYPE_TITLE
```

Flag used to indicate that an element is to be labeled with the value of its title attribute.

LABEL_TYPE_X
```
static final public int LABEL_TYPE_X
```

Flag used to indicate that an element is to be labeled with the value of its x-coordinate.

LABEL_TYPE_Y
```
static final public int LABEL_TYPE_Y
```

Flag used to indicate that an element is to be labeled with the value of its y-coordinate.

LABEL_TYPE_Z
```
static final public int LABEL_TYPE_Z
```

Flag used to indicate that an element is to be labeled with the value of its y-coordinate.

TRANSFORM_CUSTOM
```
static final public int TRANSFORM_CUSTOM
```

Flag used to indicate that the axis using a custom transformation.

TRANSFORM_LINEAR
```
static final public int TRANSFORM_LINEAR
```

Flag used to indicate that the axis uses linear scaling.

TRANSFORM_LOG
```
static final public int TRANSFORM_LOG
```

Flag used to indicate that the axis uses logarithmic scaling.

Constructor

AbstractChartNode
```
public AbstractChartNode(AbstractChartNode parent)
```

Methods

clone

`protected Object clone(Map 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;
}
```

Parameter

`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 Object clone(Object value, Map 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

`final protected List clone(List in, Map hashClonedNode)`

Description

Returns a deep copy of a vector of ChartNode's.

clone

`final protected Map clone(Map hashIn, Map 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.)

getAbstractParent

`public AbstractChartNode getAbstractParent()`

Description

Returns the parent of this node. Note that this is *not* an attribute setting. Note that there is no setParent function.

Returns

A `AbstractChartNode` 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.

getAttribute

`public Object getAttribute(String name)`

Description

Gets an attribute.

Parameter

`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.

getBooleanAttribute

`public boolean getBooleanAttribute(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.

getChildList

```
final protected List getChildList()
```

Description

Returns the children of this node.

Returns

a `List` array which contains the children of this node. It may be null.

getColorAttribute

```
public Color getColorAttribute(String name)
```

Description

Convenience routine to get a Color-valued attribute.

Parameter

`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.

getCustomTransform

```
public Transform getCustomTransform()
```

Description

Returns the value of the “CustomTransform” attribute.

Returns

an `Transform` which contains the value of the “Transform” attribute

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(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.

getFillColor

```
public 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.

getFont

```
public 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 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.

getImage

```
public Image getImage()
```

Description

Returns the value of the “Image” attribute.

Returns

the `Image` value of the “Image” attribute

getIntegerAttribute

```
public int getIntegerAttribute(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 `com.imsl.chart.AbstractChartNode.LABEL_TYPE_NONE` (p. 1568) is returned.

Returns

The `int` value of the “LabelType” attribute.

getLightColor

```
public Color getLightColor()
```

Description

Returns the value of the “LightColor” attribute.

Returns

The `Color` value of the “LightColor” attribute, if defined. Otherwise, a default color value is returned.

getLineColor

`public 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.

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 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 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

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.

getName

```
public 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.

getStringAttribute

```
public String getStringAttribute(String name)
```

Description

Convenience routine to get a String-valued attribute.

Parameter

`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.

getTextColor

```
public 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 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.

getTransform

```
public int getTransform()
```

Description

Returns the value of the "Transform" attribute.

Returns

an `int` which contains the value of the "Transform" attribute

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(AbstractChartNode node)`

Description

Returns true if this node is an ancestor of the argument node.

Parameter

`node` – a `AbstractChartNode` object

Returns

a `boolean`, true if this node is an ancestor of the argument, node

isAttributeSet

`public boolean isAttributeSet(String name)`

Description

Determines if an attribute is defined (may have been inherited).

Parameter

`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(String name)`

Description

Determines if an attribute is defined in this node (not inherited).

Parameter

`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

`static public 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

parseColor

`static public Color parseColor(String nameColor)`

Description

Returns a color specified by name or a red-green-blue triple.

Parameter

`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`.

Exception

`IllegalArgumentException` is thrown if the color name is not known.

remove

`public void remove()`

Description

Removes the node from its parents list of children.

setAttribute

`public void setAttribute(String name, 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.

Parameter

`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.

Parameter

`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.

Parameter

`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).

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.

Parameter

`value` – an object implementing the Transform interface.

---

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.

Parameter

`value` – `int` "Density" value which specifies the number of minor tick marks per major tick mark.

---

setFillColor

```
public void setFillColor(Color color)
```

Description

Sets the value of the "FillColor" attribute.

Parameter

`color` – `Color` "FillColor" value

---

setFillColor

```
public void setFillColor(String color)
```

Description

Sets the "FillColor" attribute to a color specified by name.

Parameter

`color` – `String` name of a color.

---

setFont

```
public void setFont(Font font)
```

Description

Sets the value of the font attributes. This function sets the "FontName", "FontStyle" and "FontSize" attributes. There is *no* "Font" attribute.

Parameter

`font` – `Font` object whose components are used to set three different attributes.

---

setFontName

```
public void setFontName(String value)
```

Description

Sets the value of the “FontName” attribute. This is used in the constructor for java.awt.Font.

Parameter

`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.

Parameter

`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.

Parameter

`value` – an `int` “FontStyle” value.

setImage

```
public void setImage(ImageIcon value)
```

Description

Sets the value of the “Image” attribute.

Parameter

`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.

Parameter

`type` – the `int` “LabelType” value

setLightColor

```
public void setLightColor(Color color)
```

Description

Sets the value of the “LightColor” attribute.

Parameter

`color` – a `Color` which contains the “LightColor” value

setLightColor

`public void setLightColor(String color)`

Description

Sets the value of the “LightColor” attribute to a color specified by name.

Parameter

`color` – `String` name of a color.

setLineColor

`public void setLineColor(Color color)`

Description

Sets the value of the “LineColor” attribute.

Parameter

`color` – the `LineColor` value

setLineColor

`public void setLineColor(String color)`

Description

Sets the value of the “LineColor” attribute.

Parameter

`color` – the `LineColor` value

setLineWidth

`public void setLineWidth(double value)`

Description

Sets the value of the “LineWidth” attribute.

Parameter

`value` – the `double` “LineWidth” value

setLocale

`public void setLocale(Locale value)`

Description

Sets the value of the “Locale” attribute. This attribute controls how formatting is done.

Parameter

`value` – the `Locale` value

setMarkerColor

`public void setMarkerColor(Color color)`

Description

Sets the value of the “MarkerColor” attribute.

Parameter

`color` – a `Color` which contains the “MarkerColor” value

setMarkerColor

```
public void setMarkerColor(String color)
```

Description

Sets the value of the “MarkerColor” attribute to a color specified by name.

Parameter

`color` – `String` name of a color.

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.

Parameter

`size` – a `double` which specifies the “MarkerSize” value

setName

```
public void setName(String value)
```

Description

Sets the value of the “Name” attribute. This the user-friendly name of the node.

Parameter

`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.

Parameter

`value` – the `int` “Number” value

setPaint

```
public void setPaint(boolean value)
```

Description

Sets the value of the “Paint” attribute.

Parameter

`value` – the `boolean` "Paint" value. If false, this node and its children are not drawn.

setTextColor

```
public void setTextColor(Color color)
```

Description

Sets the value of the "TextColor" attribute.

Parameter

`color` – a `Color` which contains the "TextColor" value

setTextColor

```
public void setTextColor(String color)
```

Description

Sets the value of the "TextColor" attribute to a color specified by name.

Parameter

`color` – `String` name of a color.

setTextFormat

```
public void setTextFormat(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.

Parameter

`value` – a `String` which contains the "TextFormat" value

setTextFormat

```
public void setTextFormat(Format value)
```

Description

Sets the value of the “TextFormat” attribute.

Parameter

`value` – a `Format` which contains the “TextFormat” value

setTickLength

`public void setTickLength(double tickLength)`

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.

Parameter

`tickLength` – a double which contains the “TickLength” 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.

Parameter

`value` – The “Transform” value. Legal values are TRANSFORM_LINEAR (the default), TRANSFORM_LOG and TRANSFORM_CUSTOM.

setX

`public void setX(Object value)`

Description

Sets the value of the “X” attribute.

Parameter

`value` – an `Object` which contains the “X” value

setY

`public void setY(Object value)`

Description

Sets the value of the “Y” attribute.

Parameter

`value` – the `Object` which contains the “Y” value

toString

`public String toString()`

Description

Returns the name of this `ChartNode`

Returns

a `String`, the name of this `ChartNode`

ChartNode class

`abstract public class com.imsl.chart.ChartNode extends com.imsl.chart.AbstractChartNode`

The base class of all of the nodes in the 2D chart tree.

It may be desirable to enable anti-aliasing on a particular ChartNode. This can be accomplished in the following way:

```
AxisXY axis = new AxisXY(chart);

RenderingHints hints = new RenderingHints(RenderingHints.KEY_ANTIALIASING,
     RenderingHints.VALUE_ANTIALIAS_ON);
axis.setAttribute("RenderingHints", hints);
```

Fields

AXIS_X_TOP

`static final public int AXIS_X_TOP`

Flag to indicate x-axis placed on top of the chart.

AXIS_Y_RIGHT

`static final public int AXIS_Y_RIGHT`

Flag to indicate y-axis placed to the right of the chart.

BAR_TYPE_HORIZONTAL

`static final public int BAR_TYPE_HORIZONTAL`

Flag to indicate a horizontal bar chart.

BAR_TYPE_VERTICAL

`static final public int BAR_TYPE_VERTICAL`

Flag to indicate a vertical bar chart.

DASH_PATTERN_DASH

```
static final public double[] DASH_PATTERN_DASH
```

Flag to draw a dashed line.

DASH_PATTERN_DASH_DOT

```
static final public double[] DASH_PATTERN_DASH_DOT
```

Flag to draw a dash-dot pattern line.

DASH_PATTERN_DOT

```
static final public double[] DASH_PATTERN_DOT
```

Flag to draw a dotted line.

DASH_PATTERN_SOLID

```
static final public double[] DASH_PATTERN_SOLID
```

Flag to draw solid line.

DATA_TYPE_FILL

```
static final public 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.

DATA_TYPE_LINE

```
static final public int DATA_TYPE_LINE
```

Value for attribute "DataType" indicating that the data points should be connected with line segments. This is the default setting.

DATA_TYPE_MARKER

```
static final public int DATA_TYPE_MARKER
```

Value for attribute "DataType" indicating that a marker should be drawn at each data point.

DATA_TYPE_PICTURE

```
static final public 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.

DATA_TYPE_TUBE

```
static final public int DATA_TYPE_TUBE
```

Value for attribute "DataType" indicating that an a tube connecting the data points should be drawn. Tubes are similar to lines, but tubes are shaded. The diameter of the tube is controlled by the attribute "LineWidth". Tube color is controlled by the attribute "LineColor".

DENDROGRAM_TYPE_HORIZONTAL

```
static final public int DENDROGRAM_TYPE_HORIZONTAL
```

Flag to indicate a horizontal dendrogram.

DENDROGRAM_TYPE_VERTICAL

```
static final public int DENDROGRAM_TYPE_VERTICAL
```

Flag to indicate a vertical dendrogram.

FILL_TYPE_GRADIENT

```
static final public 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.

FILL_TYPE_NONE

```
static final public int FILL_TYPE_NONE
```

Value for attribute "FillType" and "FillOutlineType" indicating that the region is not to be drawn.

FILL_TYPE_PAINT

```
static final public int FILL_TYPE_PAINT
```

Value for attribute "FillType" indicating that the region is to be drawn using the texture specified by the attribute FillPaint.

FILL_TYPE_SOLID

```
static final public 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.

LABEL_TYPE_PERCENT

```
static final public 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.

MARKER_TYPE_ASTERISK

```
static final public int MARKER_TYPE_ASTERISK
```

Flag for a asterisk data marker.

MARKER_TYPE_CIRCLE_CIRCLE

```
static final public int MARKER_TYPE_CIRCLE_CIRCLE
```

Flag for a circle in a circle data marker.

MARKER_TYPE_CIRCLE_PLUS

```
static final public int MARKER_TYPE_CIRCLE_PLUS
```

Flag for a plus in a circle data marker.

MARKER_TYPE_CIRCLE_X

```
static final public int MARKER_TYPE_CIRCLE_X
```

Flag for an x in a circle data marker.

MARKER_TYPE_DIAMOND_PLUS

```
static final public int MARKER_TYPE_DIAMOND_PLUS
```

Flag for a plus in a diamond data marker.

MARKER_TYPE_FILLED_CIRCLE

```
static final public int MARKER_TYPE_FILLED_CIRCLE
```

Flag for a filled circle data marker.

MARKER_TYPE_FILLED_DIAMOND

```
static final public int MARKER_TYPE_FILLED_DIAMOND
```

Flag for a filled diamond data marker.

MARKER_TYPE_FILLED_SQUARE

```
static final public int MARKER_TYPE_FILLED_SQUARE
```

Flag for a filled square data marker.

MARKER_TYPE_FILLED_TRIANGLE

```
static final public int MARKER_TYPE_FILLED_TRIANGLE
```

Flag for a filled triangle data marker.

MARKER_TYPE_HOLLOW_CIRCLE

```
static final public int MARKER_TYPE_HOLLOW_CIRCLE
```

Flag for a hollow circle data marker.

MARKER_TYPE_HOLLOW_DIAMOND

```
static final public int MARKER_TYPE_HOLLOW_DIAMOND
```

Flag for a hollow diamond data marker.

MARKER_TYPE_HOLLOW_SQUARE

```
static final public int MARKER_TYPE_HOLLOW_SQUARE
```

Flag for a hollow square data marker.

MARKER_TYPE_HOLLOW_TRIANGLE

```
static final public int MARKER_TYPE_HOLLOW_TRIANGLE
```

Flag for hollow triangle data marker.

MARKER_TYPE_OCTAGON_PLUS

```
static final public int MARKER_TYPE_OCTAGON_PLUS
```

Flag for a plus in an octagon data marker.

MARKER_TYPE_OCTAGON_X

```
static final public int MARKER_TYPE_OCTAGON_X
```

Flag for a x in an octagon data marker.

MARKER_TYPE_PLUS

```
static final public int MARKER_TYPE_PLUS
```

Flag for a plus-shaped data marker.

MARKER_TYPE_SQUARE_PLUS

```
static final public int MARKER_TYPE_SQUARE_PLUS
```

Flag for a plus in a square data marker.

MARKER_TYPE_SQUARE_X

```
static final public int MARKER_TYPE_SQUARE_X
```

Flag for an x in a square data marker.

MARKER_TYPE_X

```
static final public int MARKER_TYPE_X
```

Flag for a x-shaped data marker.

TEXT_X_CENTER

```
static final public int TEXT_X_CENTER
```

Value for attribute “TextAlignment” indicating that the text should be centered.

TEXT_X_LEFT

```
static final public int TEXT_X_LEFT
```

Value for attribute “TextAlignment” indicating that the text should be left adjusted. This is the default setting.

TEXT_X_RIGHT

```
static final public int TEXT_X_RIGHT
```

Value for attribute “TextAlignment” indicating that the text should be right adjusted.

TEXT_Y_BOTTOM

```
static final public int TEXT_Y_BOTTOM
```

Value for attribute “TextAlignment” indicating that the text should be drawn on the baseline. This is the default setting.

TEXT_Y_CENTER

```
static final public int TEXT_Y_CENTER
```

Value for attribute “TextAlignment” indicating that the text should be vertically centered.

TEXT_Y_TOP

```
static final public 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.

Constructor

ChartNode

```
public ChartNode(ChartNode parent)
```

Description

Construct a `ChartNode` object.

Parameter

`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.

Parameter

`pickListener` – the `PickListener` to be added to this node

firePickListeners

```
public void firePickListeners(MouseEvent event)
```

Description

Fires the pick listeners defined at this node and at all of its ancestors, if the event “hits” the node.

Parameter

`event` – `MouseEvent` which determines which nodes have been selected

getALT

`public String getALT()`

Description

Returns the value of the “ALT” attribute.

Returns

The value of the “ALT” 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.

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

`final public 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.

getComponent

`public 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

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.

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.)

getFillOutlineColor

```
public 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 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.

getGradient

```
public 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 String getHREF()
```

Description

Returns the value of the "HREF" attribute.

Returns

The value of the “HREF” attribute.

getLegend

`public Legend getLegend()`

Description

Returns the value of the “Legend” attribute.

Returns

the `Legend` value of the “Legend” attribute

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.

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.

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.

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 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 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..

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 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.

getTitle

```
public Text getTitle()
```

Description

Returns the value of the “Title” attribute.

Returns

the `Text` value of the "Title" attribute

getToolTip

```
public String getToolTip()
```

Description

Returns the value of the "ToolTip" attribute.

Returns

the `String` value of the "ToolTip" attribute

getViewport

```
public double[] getViewport()
```

Description

Returns the value of the "Viewport" attribute.

Returns

a `double`[4] array containing xmin, xmax, ymin, ymax

isBitSet

```
static public 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

```
abstract public void paint(Draw draw)
```

Description

Paints this node and all of its children.

Parameter

`draw` – the `Draw` object to be painted

prePaint

```
public void prePaint()
```

Description

The prePaint method is called in all nodes in a chart just before the chart is painted. The default implementation does nothing. Override this method to do computations just before painting in a chart node.

removePickListener

`public void removePickListener(PickListener pickListener)`

Description

Removes a `PickListener` from this node.

Parameter

`pickListener` – the `PickListener` to be removed from this node

setALT

`public void setALT(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. *

Parameter

`value` – "ALT" value.

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.

Parameter

`value` – the `double` "BarGap" value

setBarType

`public void setBarType(int value)`

Description

Sets the value of the "BarType" attribute.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`value` – an object implementing the Transform interface.

setDataType

```
public void setDataType(int value)
```

Description

Sets the value of the “DataType” attribute.

Parameter

`value` – “DataType” value. This should be some xor-ed combination of DATA_TYPE_LINE, DATA_TYPE_MARKER.

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.

Parameter

`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.

Parameter

`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.

setFillOutlineColor

```
public void setFillOutlineColor(Color color)
```

Description

Sets the value of the “FillOutlineColor” attribute.

Parameter

`color` – a `Color` “FillOutlineColor” value.

setFillOutlineColor

```
public void setFillOutlineColor(String color)
```

Description

Sets the value of the “FillOutlineColor” attribute to a color specified by name.

Parameter

`color` – `String` name of a color.

setFillOutlineType

```
public void setFillOutlineType(int value)
```

Description

Sets the value of the “FillOutlineType” attribute.

Parameter

`value` – “FillOutlineType” value. This value should be FILL_TYPE_NONE or FILL_TYPE_SOLID.

setFillPaint

```
public void setFillPaint(Paint value)
```

Description

Sets the value of the “FillPaint” attribute.

Parameter

`value` – “FillPaint” value.

setFillPaint

```
public void setFillPaint(URL urlImage)
```

Description

Sets the value of the “FillPaint” attribute.

Parameter

`urlImage` – is the URL of an image used to set the FillPaint attribute.

setFillPaint

```
public void setFillPaint(ImageIcon imageIcon)
```

Description

Sets the value of the “FillPaint” attribute.

Parameter

`imageIcon` – is used to create a Paint object that is used as the value of the “FillPaint” attribute.

setFillType

```
public void setFillType(int value)
```

Description

Sets the value of the “FillType” attribute.

Parameter

`value` – “FillType” value. This value should be FILL_TYPE_NONE, FILL_TYPE_SOLID, FILL_TYPE_GRADIENT or FILL_TYPE_PAINT.

setGradient

```
public void setGradient(Color[] colorGradient)
```

Description

Sets the value of the “Gradient” attribute.

Parameter

`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 `com.imsl.chart.ChartNode.setGradient` (p. 1603) for details on the interpretation of these colors.

setGradient

```
public void setGradient(Color colorLL, Color colorLR, Color colorUR, 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(String colorLL, String colorLR, String colorUR, 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(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.

Parameter

`value` – "HREF" value.

setImage

```
public void setImage(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.

Parameter

`value` – `Image` value.

setLineDashPattern

```
public void setLineDashPattern(double[] value)
```

Description

Sets the value of the "LineDashPattern" attribute.

Parameter

`value` – `double` "LineDashPattern" value.

setMarkerDashPattern

```
public void setMarkerDashPattern(double[] value)
```

Description

Sets the value of the "MarkerDashPattern" attribute.

Parameter

`value` – `double` array which contains the "MarkerDashPattern" 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.

Parameter

`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.

Parameter

`type` – the `int` “MarkerType” value.

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.

Parameter

`value` – the `double` “Reference” value

setScreenSize

```
public void setScreenSize(Dimension value)
```

Description

Sets the value of the “ScreenSize” attribute.

Parameter

`value` – the `Dimension` “ScreenSize” value.

setSize

```
public void setSize(Dimension value)
```

Description

Sets the value of the “Size” attribute.

Parameter

`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.

Parameter

`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.

Parameter

`value` – an `int` “TextAngle” value

---

setTextColor

```
public void setTextColor(Color color)
```

Description

Sets the value of the “TextColor” attribute.

Parameter

`color` – a `Color` which contains the “TextColor” value

---

setTextColor

```
public void setTextColor(String color)
```

Description

Sets the value of the “TextColor” attribute to a color specified by name.

Parameter

`color` – `String` name of a color.

---

setTitle

```
public void setTitle(Text value)
```

Description

Sets the value of the “Title” attribute.

Parameter

`value` – a `Text` which contains the “Title” value

---

setTitle

```
public void setTitle(String value)
```

Description

Sets the value of the “Title” attribute.

Parameter

`value` – a `String` which contains the “Title” value

---

setToolTip

```
public void setToolTip(String value)
```

Description

Sets the value of the “ToolTip” attribute.

Parameter

`value` – a `String` which contains the "ToolTip" value

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 upper left corner. This attribute affects only Axis nodes, since they contain the mappings to device space.

Parameter

`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

Background class

```
public class com.imsl.chart.Background extends com.imsl.chart.AxisXY
```

The background of a chart.

Grid is created by `com.imsl.chart.Chart` (p. 1562) as its child. It can be retrieved using the method `com.imsl.chart.ChartNode.getBackground` (p. 1592).

Fill attributes in this node control the drawing of the background.

Method

paint

```
public void paint(Draw draw)
```

Description

Paint this node. This is not normally called by a user program.

Parameter

`draw` – the `Draw` object to be painted

ChartTitle class

```
public class com.imsl.chart.ChartTitle extends com.imsl.chart.AxisXY
```

The main title of a chart.

ChartTitle is created by `com.imsl.chart.Chart` (p. 1562) as its child. It can be retrieved using the method `com.imsl.chart.ChartNode.getChartTitle` (p. 1593) .

The chart title is the value of the "Title" attribute at this node. Text attributes in this node control the drawing of the title.

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.

Parameter

`draw` – the `Draw` object to be painted

Legend class

```
public class com.imsl.chart.Legend extends com.imsl.chart.AxisXY
```

The chart legend.

Legend is created by `com.imsl.chart.Chart` (p. 1562) as its child. It can be retrieved using the method `com.imsl.chart.ChartNode.getLegend` (p. 1596) .

By default the legend is not drawn. To have it drawn, set its "Paint" attribute to true.

`com.imsl.chart.Data` (p. 1634) objects that have their "Title" attribute defined are automatically entered into the legend. The "Viewport" attribute for this node is set to [0.83,0.98] by [0.1,0.6].

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.

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.

Parameter

`draw` – the `Draw` object to be painted

Annotation class

`public class com.imsl.chart.Annotation extends com.imsl.chart.Data`

Draws an annotation. Locations are defined in chart (user) coordinates. To convert between device and user coordinates see `com.imsl.chart.Axis.mapDeviceToUser` (p. 1614) and `com.imsl.chart.Axis.mapUserToDevice` (p. 1614) .

Constructors

Annotation

`public Annotation(ChartNode parent, Text text, double x, double y)`

Description

Creates a `Text` object at the specific x,y location in chart coordinates.

Parameters

`parent` – the `ChartNode` parent of this data node, usually an Axis object

`text` – the `Text` object to draw

`x` – the x location in user coordinates

`y` – the y location in user coordinates

Annotation

`public Annotation(ChartNode parent, Image img, double x, double y)`

Description

Renders an `Image` object centered at an x,y location in chart coordinates.

Parameters

`parent` – the `ChartNode` parent of this data node, usually an Axis object

`img` – the `Image` object to draw

`x` – the x location in user coordinates

`y` – the y location in user coordinates

Annotation

`public Annotation(ChartNode parent, String string, double x, double y)`

Description

Draws a `String` at the specific x,y location in chart coordinates.

Parameters

`parent` – the `ChartNode` parent of this data node, usually an Axis object

`string` – the `String` to draw

`x` – the x location in user coordinates

`y` – the y location in user coordinates

Methods

getText

`public Text getText()`

Description

Gets the `Text` for this `Annotation` object.

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.

Parameter

`draw` – the `Draw` object to be painted

setLocation

```
public void setLocation(double x, double y)
```

Description

Update the location of this `Annotation` instance.

Parameters

`x` – a `double` specifying the new x location in chart coordinates.

`y` – a `double` specifying the new y location in chart coordinates.

setString

```
public void setString(String string)
```

Description

Sets the `String` for the `Text` object to render.

Parameter

`string` – the new `String` value

setText

```
public void setText(Text text)
```

Description

Sets the `Text` object to render.

Parameter

`text` – the new `Text` object

Grid class

```
public class com.imsl.chart.Grid extends com.imsl.chart.ChartNode
```

Draws the grid lines perpendicular to an axis.

Grid is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getGrid` (p. 1619) .

Line attributes in this node control the drawing of the grid lines.

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.

Parameter

`draw` – the `Draw` object to be painted

Axis class

`abstract public class com.imsl.chart.Axis extends com.imsl.chart.ChartNode`

The Axis node provides the mapping for all of its children from the user coordinate space to the device (screen) space.

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.

Parameter

`chart` – a `Chart` object, the parent of this node

Methods

mapDeviceToUser

```
abstract 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 `double`[2] array on input, on output, the user coordinates

mapUserToDevice

```
abstract 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 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.

Parameter

`draw` – a `Draw` object which specifies the chart tree to be rendered on the screen

setupMapping

```
abstract 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.

AxisXY class

`public class com.imsl.chart.AxisXY extends com.imsl.chart.Axis`

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)

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" attribute for this node is set to [0.2,0.8] by [0.2,0.8].

Parameter

`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.

Parameter

`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.

Parameter

`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

setWindow

```
public void setWindow(double[] value)
```

Description

Sets the window in user coordinates along an axis.

Parameter

`value` – a `double` array which contains the minimum and maximum of the window along an axis

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.

Axis1D class

```
public class com.imsl.chart.Axis1D extends com.imsl.chart.ChartNode
```

An x-axis or a y-axis.

Axis1D is created by `com.imsl.chart.AxisXY` (p. 1615) as its child. It can be retrieved using the method `com.imsl.chart.AxisXY.getAxisX` (p. 1615) or `com.imsl.chart.AxisXY.getAxisY` (p. 1615) .

It in turn creates the following child nodes: `com.imsl.chart.AxisLine` (p. 1623) , `com.imsl.chart.AxisLabel` (p. 1622) , `com.imsl.chart.AxisTitle` (p. 1623) , `com.imsl.chart.AxisUnit` (p. 1624) , `com.imsl.chart.MajorTick` (p. 1625) , `com.imsl.chart.MinorTick` (p. 1625) and `com.imsl.chart.Grid` (p. 1612) .

The number of tick marks (“Number” attribute) is set to 5, but autoscaling can change this value.

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.

Parameter

`draw` – the `Draw` object to be painted

setFirstTick

`public void setFirstTick(double firstTick)`

Description

Convenience routine to set the "FirstTick" attribute.

Parameter

`firstTick` – a `double`, the location of the first tick

setTickInterval

`public void setTickInterval(double tickInterval)`

Description

Sets the tick interval.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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

AxisLabel class

`public class com.imsl.chart.AxisLabel extends com.imsl.chart.ChartNode`

The labels on an axis.

`AxisLabel` is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getAxisLabel` (p. 1618) .

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.

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.

Parameter

`draw` – the `Draw` object to be painted

setLabels

`public void setLabels(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`.

Parameter

`value` – a `String` array containing the labels for the major tick marks

AxisLine class

`public class com.imsl.chart.AxisLine extends com.imsl.chart.ChartNode`

The axis line.

`AxisLine` is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getAxisLine` (p. 1618) .

Line attributes in this node control the drawing of the axis line.

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.

Parameter

`draw` – the `Draw` object to be painted

AxisTitle class

`public class com.imsl.chart.AxisTitle extends com.imsl.chart.ChartNode`

The title on an axis.

`AxisTitle` is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getAxisTitle` (p. 1618) .

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.

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.

Parameter

`draw` – the `Draw` object to be painted

AxisUnit class

`public class com.imsl.chart.AxisUnit extends com.imsl.chart.ChartNode`

The unit title on an axis.

`AxisUnit` is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getAxisUnit` (p. 1618).

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.

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.

Parameter

`draw` – the `Draw` object to be painted

MajorTick class

`public class com.imsl.chart.MajorTick extends com.imsl.chart.ChartNode`

The major tick marks.

MajorTick is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getMajorTick` (p. 1619) .

Line attributes in this node control the drawing of the major tick marks.

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.

Parameter

`draw` – the `Draw` object to be painted

MinorTick class

`public class com.imsl.chart.MinorTick extends com.imsl.chart.ChartNode`

The minor tick marks.

MinorTick is created by `com.imsl.chart.Axis1D` (p. 1617) as its child. It can be retrieved using the method `com.imsl.chart.Axis1D.getMinorTick` (p. 1619) .

Line attributes in this node control the drawing of the minor tick marks.

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.

Parameter

`draw` – the `Draw` object to be painted

Transform interface

```
public interface com.imsl.chart.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.

Methods

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.

TransformDate class

`public class com.imsl.chart.TransformDate implements com.imsl.chart.Transform`

Defines a transformation along an axis that skips weekend dates.

Constructor

TransformDate

`public TransformDate()`

Methods

isWeekday

`public boolean isWeekday(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.

AxisR class

`public class com.imsl.chart.AxisR extends com.imsl.chart.ChartNode`

The R-axis in a polar plot.

`AxisR` is created by `com.imsl.chart.Polar` (p. 1741) as its child. It can be retrieved using the method `com.imsl.chart.Polar.getAxisR` (p. 1742) .

It in turn creates the following child nodes: `com.imsl.chart.AxisRLine` (p. 1631) , `com.imsl.chart.AxisRLabel` (p. 1630) and `com.imsl.chart.AxisRMajorTick` (p. 1631) .

The number of tick marks (“Number” attribute) is set to 4, but autoscaling can change this value.

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.

Parameter

`draw` – the `Draw` object to be painted

setTickInterval

```
public void setTickInterval(double tickInterval)
```

Description

Sets the tick interval.

Parameter

`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.

Parameter

`rmax` – a `double` specifying the radius at which the `AxisTheta` is drawn.

AxisRLabel class

`public class com.imsl.chart.AxisRLabel extends com.imsl.chart.ChartNode`

The labels on an axis.

`AxisRLabel` is created by `com.imsl.chart.AxisR` (p. 1628) as its child. It can be retrieved using the method `com.imsl.chart.AxisR.getAxisRLabel` (p. 1628).

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.

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.

Parameter

`draw` – the `Draw` object to be painted

setLabels

`public void setLabels(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`.

Parameter

`value` – a `String` array containing the labels to be used to label the major tick marks

AxisRLine class

`public class com.imsl.chart.AxisRLine extends com.imsl.chart.ChartNode`

The radius axis line in a polar plot.

`AxisRLine` is created by `com.imsl.chart.AxisR` (p. 1628) as its child. It can be retrieved using the method `com.imsl.chart.AxisR.getAxisRLine` (p. 1628) .

Line attributes in this node control the drawing of the axis line.

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.

Parameter

`draw` – the `Draw` object to be painted

AxisRMajorTick class

`public class com.imsl.chart.AxisRMajorTick extends com.imsl.chart.ChartNode`

The major tick marks for the radius axis in a polar plot.

`AxisRMajorTick` is created by `com.imsl.chart.AxisR` (p. 1628) as its child. It can be retrieved using the method `com.imsl.chart.AxisR.getAxisRMajorTick` (p. 1628) .

Line attributes in this node control the drawing of the major tick marks.

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.

Parameter

`draw` – the `Draw` object to be painted

AxisTheta class

`public class com.imsl.chart.AxisTheta extends com.imsl.chart.ChartNode`

The angular axis in a polar plot.

`AxisTheta` is created by `com.imsl.chart.Polar` (p. 1741) as its child. It can be retrieved using the method `com.imsl.chart.Polar.getAxisTheta` (p. 1742).

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/π. So, for labels like `1.5\\u03c0`, where `\\u03c0` is the Unicode character for π, 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.

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.

Parameter

`draw` – the `Draw` object to be painted

setWindow

```
public void setWindow(double[] window)
```

Description

Sets the window for an `AxisTheta`.

Parameter

`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.

GridPolar class

```
public class com.imsl.chart.GridPolar extends com.imsl.chart.ChartNode
```

Draws the grid lines for a polar plot.

PolarGrid is created by `com.imsl.chart.Polar` (p. 1741) as its child. It can be retrieved using the method `com.imsl.chart.Polar.getGridPolar` (p. 1742) .

Line attributes in this node control the drawing of the grid lines.

Method

paint

`public void paint(Draw draw)`

Description

Paints this node and all of its children.

Parameter

`draw` – the `Draw` object to be painted

Data class

`public class com.imsl.chart.Data extends com.imsl.chart.ChartNode`

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 `com.imsl.chart.ChartNode.DATA_TYPE_LINE` (p. 1587) bit is set, the line attributes are active. If the `com.imsl.chart.ChartNode.DATA_TYPE_MARKER` (p. 1587) bit is set, the marker attributes are active. If the `com.imsl.chart.ChartNode.DATA_TYPE_FILL` (p. 1587) 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.

Constructors

Data

`public Data(ChartNode parent)`

Description

Creates a data node.

Parameter

`parent` – the `ChartNode` parent of this data node

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

---

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

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. The argument `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.

Parameter

`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.

Parameter

`draw` – the `Draw` object to be painted

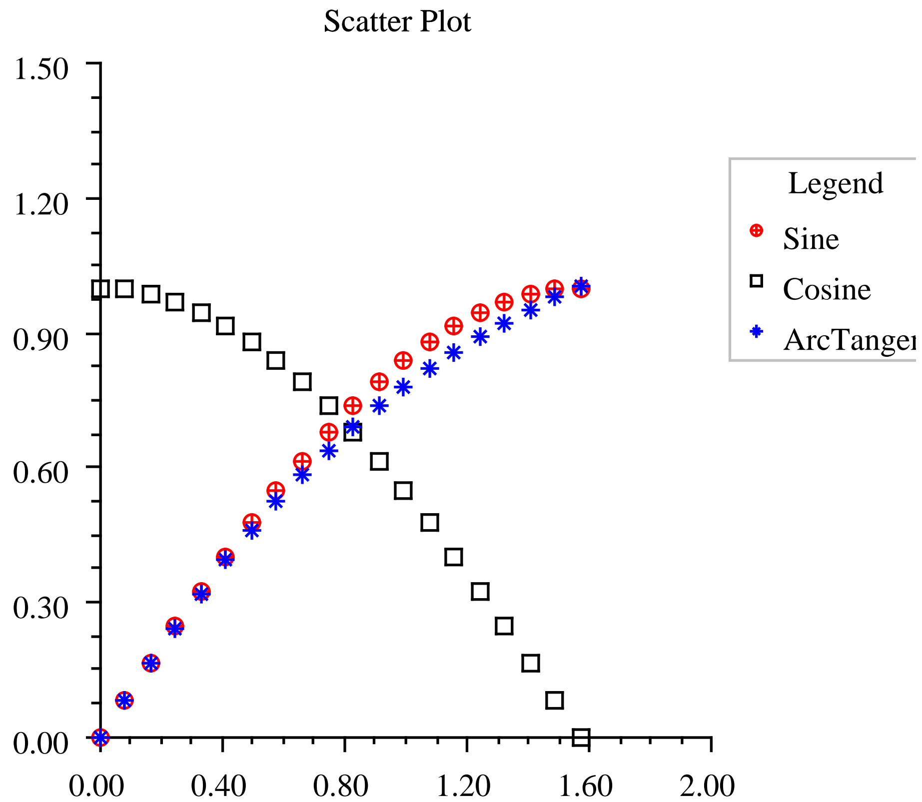
Example: Scatter Chart

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(Data.DATA_TYPE_MARKER);
        d2.setDataType(Data.DATA_TYPE_MARKER);
        d3.setDataType(Data.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"));
        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.setVisible(true);
    }
}
```

Output



Example: Line Chart

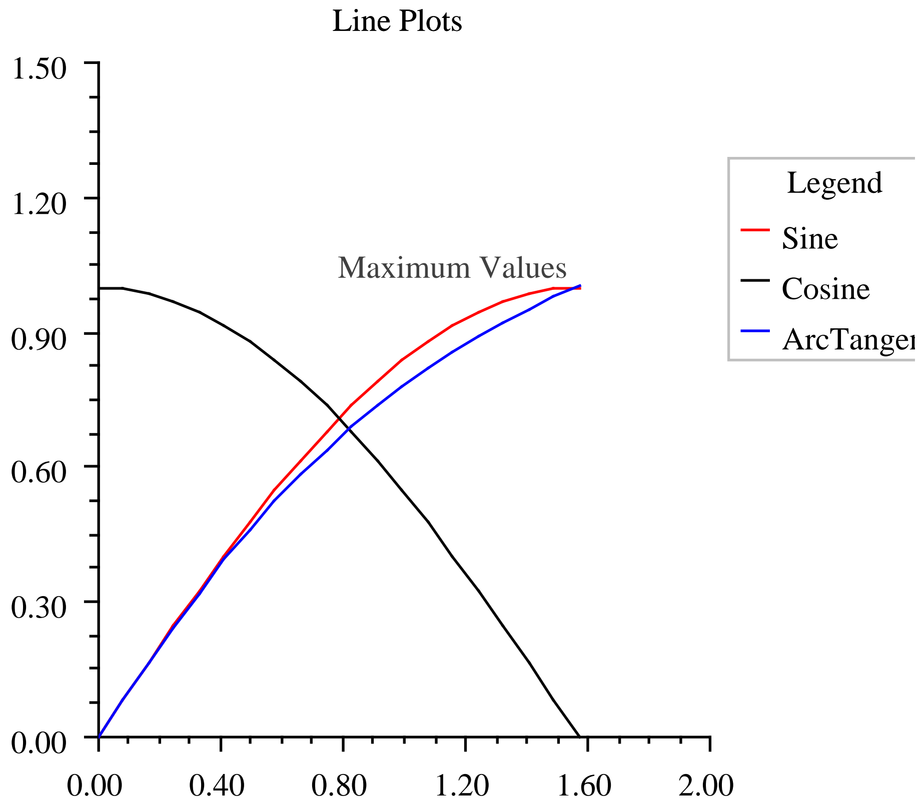
A simple line chart is constructed in this example. Three data sets are used and a legend is added to the chart. An annotation is included indicating a feature on the chart. This class can be used either as an applet or as an application.

```
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"));
    legend.setPaint(true);

    //  Add an Annotation
    Text label = new Text("Maximum Values");
    label.setAlignment(Axis.TEXT_X_RIGHT);
    Annotation ann = new Annotation(axis, label, 1.6, 1.02);
    ann.setTextColor(Color.darkGray);

    //  Set the Chart Title
    chart.getChartTitle().setTitle("Line Plots");
}
```

```
    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        LineEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



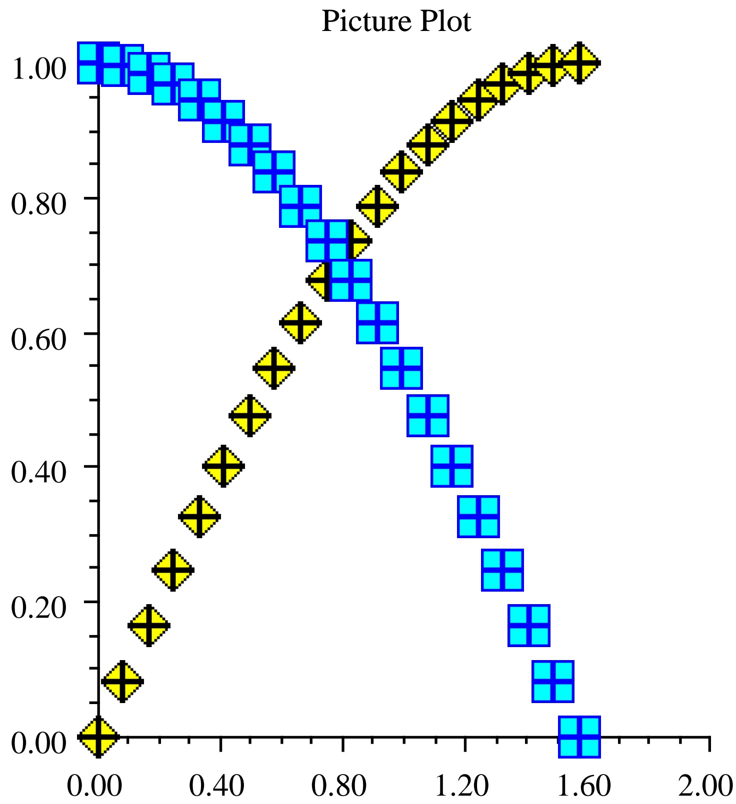
Example: Picture Chart

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 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("marker.gif"));
        d2.setDataType(Data.DATA_TYPE_PICTURE);
        d2.setImage(loadImage("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.setVisible(true);
```

```
    }
}
```

Output



Example: Area Chart

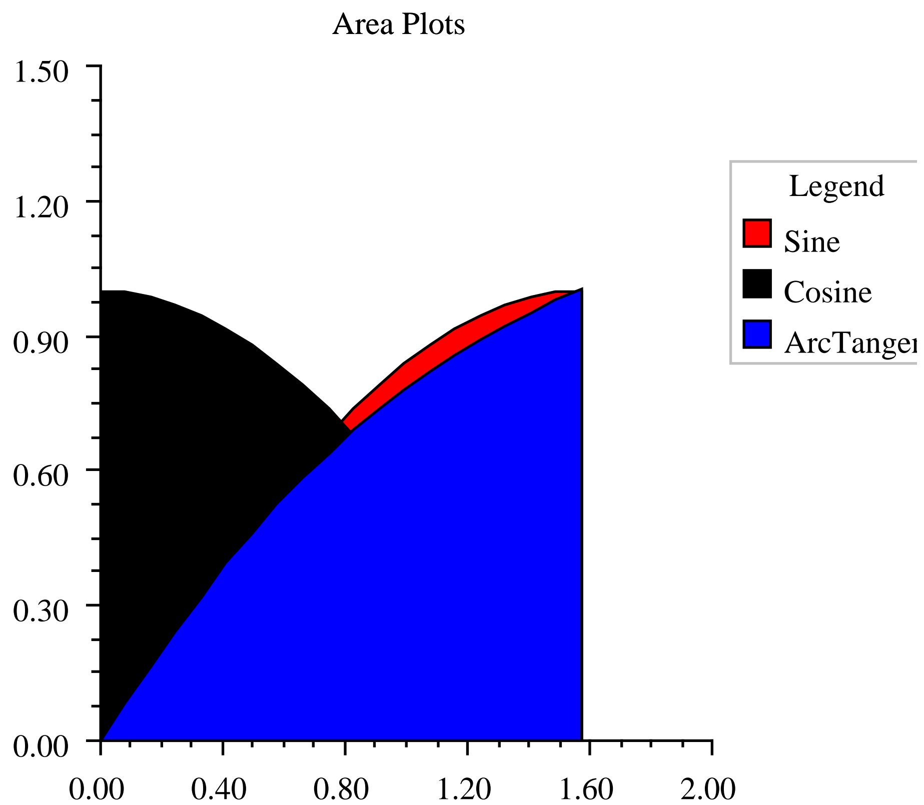
An area chart is constructed in this example. Three data sets are used and a legend is added to the chart. This class can be used either as an applet or as an application.

```
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.setVisible(true);
    }
}
```

Output



ChartFunction interface

`public interface com.imsl.chart.ChartFunction`

An interface that allows a function to be plotted.

Method

f

`public double f(double x)`

Description

Function to be charted.

ChartSpline class

`public class com.imsl.chart.ChartSpline implements com.imsl.chart.ChartFunction`

Wrap a spline into a ChartFunction to be plotted.

Constructors

ChartSpline

`public ChartSpline(Spline spline)`

Description

Creates a ChartSpline from a Spline.

Parameter

`spline` – The Spline to be plotted.

ChartSpline

`public ChartSpline(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.

Text class

`public class com.imsl.chart.Text implements Serializable`

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.

Constructors

Text

`public Text(String string)`

Description

Construct a `Text` object.

Parameter

`string` – a `String`

Text

`public Text(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.

Text

`public 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.

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 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.

Parameter

`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).

Parameter

`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(String string)`

Description

Sets the string for this `Text` object.

Parameter

`string` – the `String`

ToolTip class

`public class com.imsl.chart.ToolTip extends com.imsl.chart.ChartNode implements com.imsl.chart.PickListener, java.awt.event.MouseMotionListener`

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.

Constructor

ToolTip

`public ToolTip(ChartNode parent)`

Description

Creates a ToolTip node that enables ToolTips on charts.

Parameter

`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(MouseEvent e)`

Description

Part of the MouseMotionListener interface.

mouseMoved

`public void mouseMoved(MouseEvent event)`

Description

Part of the MouseMotionListener interface.

paint

`public void paint(Draw draw)`

Description

Paints this node and all of its children.

Parameter

`draw` – the `Draw` object to be painted

pickPerformed

`public void pickPerformed(PickEvent event)`

Description

Part of the PickListener interface.

FillPaint class

`public class com.imsl.chart.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.

Methods

checkerboard

```
static public Paint checkerboard(int n, Color colorA, 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

```
static public Paint crosshatch(int n, int p, Color colorBackground, 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

```
static public Paint diagonal(int n, Color colorA, 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

```
static public Paint diamond(int n, int p, Color colorBackground, 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

```
static public Paint diamondHatch(int n, int p, Color colorBackground, 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

```
static public Paint dot(int n, int r, Color colorBackground, 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

```
static public Paint horizontalStripe(int n, int p, Color colorBackground, 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

```
static public Paint image(ImageIcon imageIcon)
```

Description

Returns a tiling of an image.

Parameter

`imageIcon` – is the image to be tiled.

Returns

a Paint containing the tiling of the image.

verticalStripe

`static public Paint verticalStripe(int n, int p, Color colorBackground, 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.

Draw class

`public class com.imsl.chart.Draw`

Chart tree renderer.

Renders the chart tree to the screen.

Fields

ERROR_BAR

`static final protected int ERROR_BAR`

FILL

`static final protected int FILL`

IMAGE

`static final protected int IMAGE`

LAST

```
static final protected int LAST
```

Flag for the last data marker.

LINE

```
static final protected int LINE
```

MARKER

```
static final protected int MARKER
```

MARKER_SCALE

```
static final protected float MARKER_SCALE
```

Normal marker size in pixels is screen width times MARKER_SCALE.

NONE

```
static final protected int NONE
```

RADIAN

```
static final protected double RADIAN
```

TEXT

```
static final protected int TEXT
```

currentType

```
protected int currentType
```

fillColor

```
protected Color fillColor
```

fillOutlineColor

```
protected Color fillOutlineColor
```

fillOutlineType

```
protected int fillOutlineType
```

fillPaint

```
protected Paint fillPaint
```

fillType

```
protected int fillType
```

graphics

```
protected Graphics2D graphics
```

haveErrorBarProperties

`protected boolean haveErrorBarProperties`

haveFillProperties

`protected boolean haveFillProperties`

haveImageProperties

`protected boolean haveImageProperties`

haveLineProperties

`protected boolean haveLineProperties`

haveMarkerProperties

`protected boolean haveMarkerProperties`

haveTextProperties

`protected boolean haveTextProperties`

imageObserver

`protected Component imageObserver`

lineColor

`protected Color lineColor`

lineDashPattern

`protected float[] lineDashPattern`

lineWidth

`protected float lineWidth`

markerColor

`protected Color markerColor`

markerDashPattern

`protected float[] markerDashPattern`

markerSize

`protected float markerSize`

markerThickness

`protected float markerThickness`

markerType

`protected int markerType`

node

```
protected ChartNode node
```

outline

```
static final protected 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).

path

```
protected GeneralPath path
```

scaleFont

```
protected float scaleFont
```

textAngle

```
protected int textAngle
```

textColor

```
protected Color textColor
```

textFont

```
protected Font textFont
```

Constructor

Draw

```
public Draw(Graphics graphics, 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.

drawClippedImage

```
public void drawClippedImage(Image image, int x, int y)
```

Description

Draws an image such that any portion of the image beyond the axis range is clipped.

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

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(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(Graphics g, Text text)
```

Description

Draws the text.

---

drawText

```
public Dimension drawText(Text text, int x, int y)
```

Description

Draws a text object.

Parameters

`text` – the `Text object 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 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(Polygon polygon)
```

Description

Fill a polygon defined by a `Polygon` object.

Parameter

`polygon` – a `Polygon` object which specifies the polygon to be filled

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

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 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 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(Rectangle clip)
```

Description

Set the clipping rectangle.

Parameter

`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.

Parameter

`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

JFrameChart class

`public class com.imsl.chart.JFrameChart extends javax.swing.JFrame`

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 “File”, “Edit”, and “Help” menu items.

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.

Parameter

`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.

Parameter

`chart` – is the new chart

JPanelChart class

`public class com.imsl.chart.JPanelChart extends javax.swing.JPanel`

A Swing JPanel that contains a chart. This class causes the contained chart to be redrawn as necessary.

Field

chart

`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.

Parameter

`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(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.

Parameter

`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.

Parameter

`chart` – is the Chart to be displayed by this container

DrawPick class

`public class com.imsl.chart.DrawPick extends com.imsl.chart.Draw`

The DrawPick class.

Constructor

DrawPick

`public DrawPick(MouseEvent event, Graphics graphics, 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, int flag)
```

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

`flag` – an `int` that indicates which caps to draw (0=none, 1=bottom, 2=top, 3=both).

---

drawImage

```
public void drawImage(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 Dimension drawText(Text text, int x, int y)
```

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(Polygon polygon)
```

Description

Fill a polygon defined by a `Polygon` object.

Parameter

`polygon` – a `Polygon` object which specifies the polygon to be filled

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

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.

Parameter

`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.

Parameter

`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

PickEvent class

`public class com.imsl.chart.PickEvent extends java.awt.event.MouseEvent`

An event that indicates that a chart element has been selected.

Constructors

PickEvent

`public PickEvent(MouseEvent event)`

Description

Construct a `PickEvent` object.

Parameter

`event` – a `MouseEvent`

PickEvent

`public PickEvent(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

Methods

getNode

`public ChartNode getNode()`

Description

Gets this `ChartNode`.

pointToLine

```
static public 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`.

Parameter

`node` – the `ChartNode` to be set

PickListener interface

```
public interface com.imsl.chart.PickListener implements java.util.EventListener
```

The listener interface for receiving pick events.

Method

pickPerformed

```
public void pickPerformed(PickEvent event)
```

Description

Public interface for `PickListener`.

Parameter

`event` – a `PickEvent`

JspBean class

`public class com.imsl.chart.JspBean implements Serializable`

JspBean is used to refer to charts in a Java Server Page that are later rendered using the ChartServlet.

Constructor

JspBean

`public JspBean()`

Description

Creates a JspBean object.

Methods

getChartServletName

`public 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 String getId()`

Description

Returns the identifier number for the chart. It is assigned a unique random value by the constructor.

getImageMap

`public 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 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 String getMapName()
```

Description

Returns the name of the client-size imagemap. This is null if CreateImageMap is false.

getSize

```
public Dimension getSize()
```

Description

Returns the size of the generated image.

registerChart

```
public void registerChart(Chart chart, 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` (p. 1678) .

`request` – from the Java Server Page.

saveChart

```
protected void saveChart(Chart chart, 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(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.

Parameter

`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.

Parameter

`createImageMap` – is true if a client-side image map is to be generated.

setSize

`public void setSize(Dimension size)`

Description

Sets the size of the generated image. Its initial value is `new Dimension(300,300)`.

Parameter

`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.

ChartServlet class

```
public class com.imsl.chart.ChartServlet extends javax.servlet.http.HttpServlet
```

The base class for chart servlets.

This class requires a `javax.servlet` Images are rendered using `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)`).

Constructor

ChartServlet

```
public ChartServlet()
```

Methods

doGet

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, 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(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.

Parameter

`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 ServletException
```

DrawMap class

```
public class com.imsl.chart.DrawMap extends com.imsl.chart.Draw
```

Creates an HTML client-side imagemap from a chart tree. Entries in the imagemap correspond to nodes that define the HREF attribute.

Constructor

DrawMap

```
public DrawMap(Graphics graphics, 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(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 Dimension drawText(Text text, int x, int y, boolean dimensionOnly)
```

---

endErrorBar

```
public void endErrorBar()
```

---

endFill

```
public void endFill()
```

---

endImage

```
public void endImage()
```

---

endLine

```
public void endLine()
```

endMarker

```
public void endMarker()
```

endText

```
public void endText()
```

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(Polygon polygon)
```

Description

Fill a polygon defined by a `Polygon` object.

Parameter

`polygon` – a `Polygon` object which specifies the polygon to be filled

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

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 String getALT()
```

Description

Returns the current ALT string.

getHREF

```
protected String getHREF()
```

Description

Returns the current HREF string.

getMap

```
public 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.

Parameter

`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.

Parameter

`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()
```

startFill

```
public void startFill()
```

startImage

```
public void startImage()
```

startLine

```
public void startLine()
```

Description

Start drawing lines.

startMarker

```
public void startMarker()
```

Description

Start drawing markers.

startText

```
public void startText()
```

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

BoxPlot class

```
public class com.imsl.chart.BoxPlot extends com.imsl.chart.Data
```

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.

Fields

BOXPLOT_TYPE_HORIZONTAL

```
static final public int BOXPLOT_TYPE_HORIZONTAL
```

Value for attribute “BoxPlotType” indicating that this is a horizontal box plot. Used in connection with `BoxPlot` nodes.

BOXPLOT_TYPE_VERTICAL

```
static final public 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

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

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.

Parameter

`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.

Parameter

`draw` – the `Draw` object to be painted

setBoxPlotType

`public void setBoxPlotType(int value)`

Description

Sets the “BoxPlotType” attribute value.

Parameter

`value` – an `int` which specifies the “BoxPlotType” attribute. Legal values are `BOXPLOT_TYPE_VERTICAL` or `BOXPLOT_TYPE_HORIZONTAL`.

setLabels

`public void setLabels(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.

Parameter

`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(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”.

Parameter

`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”.

Parameter

`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`.

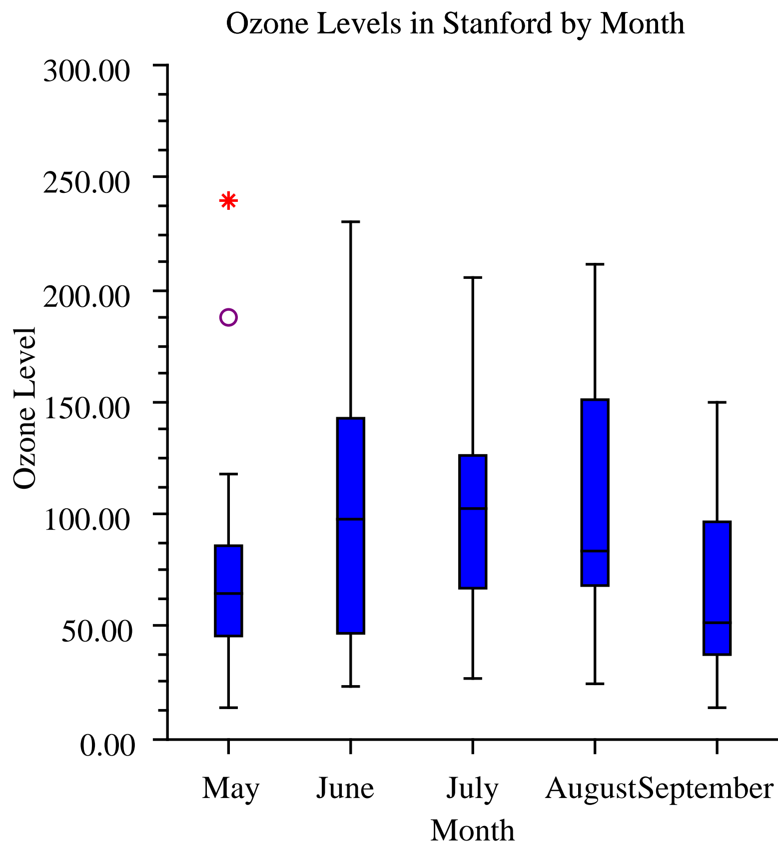
Example: Box Plot Chart

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}
        };
        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(java.awt.Color.blue);
        boxPlot.getOutsideMarkers().
                setMarkerType(boxPlot.MARKER_TYPE_HOLLOW_CIRCLE);
        boxPlot.getOutsideMarkers().setMarkerColor("purple");
        boxPlot.getFarMarkers().setMarkerType(boxPlot.MARKER_TYPE_ASTERISK);
        boxPlot.getFarMarkers().setMarkerColor(java.awt.Color.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.setVisible(true);
    }
}
```

Output



BoxPlot.Statistics class

`static public class com.imsl.chart.BoxPlot.Statistics implements Serializable`

Computes the statistics for one set of observations in a `Boxplot`.

Constructor

BoxPlot.Statistics

`public BoxPlot.Statistics(double[] obs)`

Description

Creates a new instance of `BoxPlot.Statistics`.

Parameter

`obs` – a `double` array containing the set of observations. There must be at least 4 observations to compute the statistics.

Exception

`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)

Contour class

```
public class com.imsl.chart.Contour extends com.imsl.chart.Data
```

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 $c_{k-1} < z \leq c_k$.

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.

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. The argument `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.

Parameter

`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)
```

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

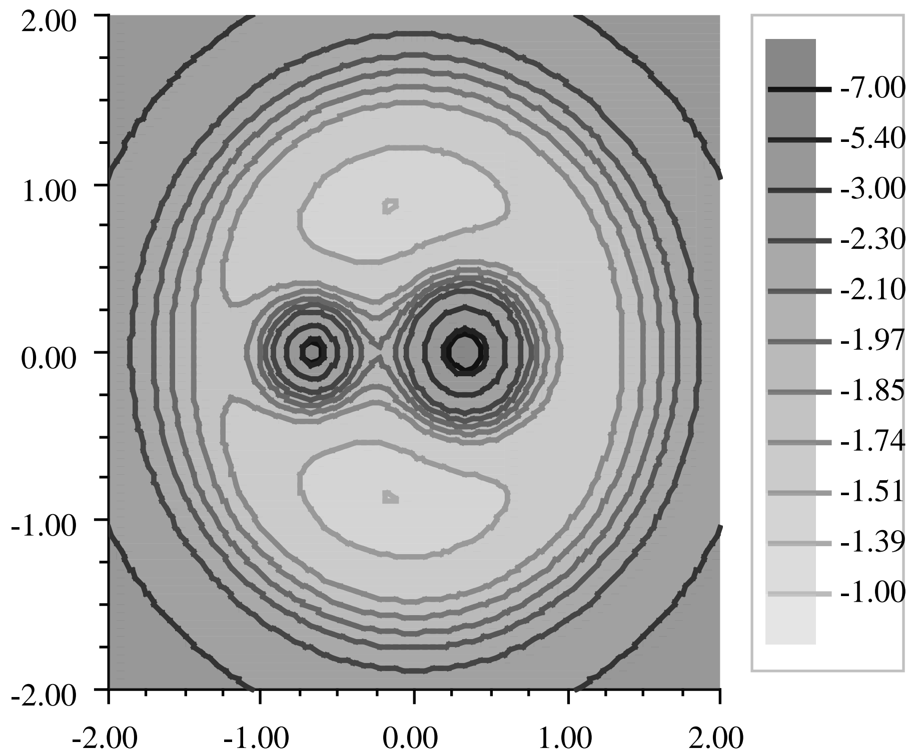
$$V = \frac{m(M_1 + M_2)G}{a}\left[\frac{x_2}{\sqrt{(x-x_1)^2+y^2}} - \frac{x_1}{\sqrt{(x-x_2)^2+y^2}} - \frac{1}{2}\left(x^2+y^2\right)\right]$$

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.setVisible(true);
    }
}
```

Output



Example: Contour Chart from Scattered Data

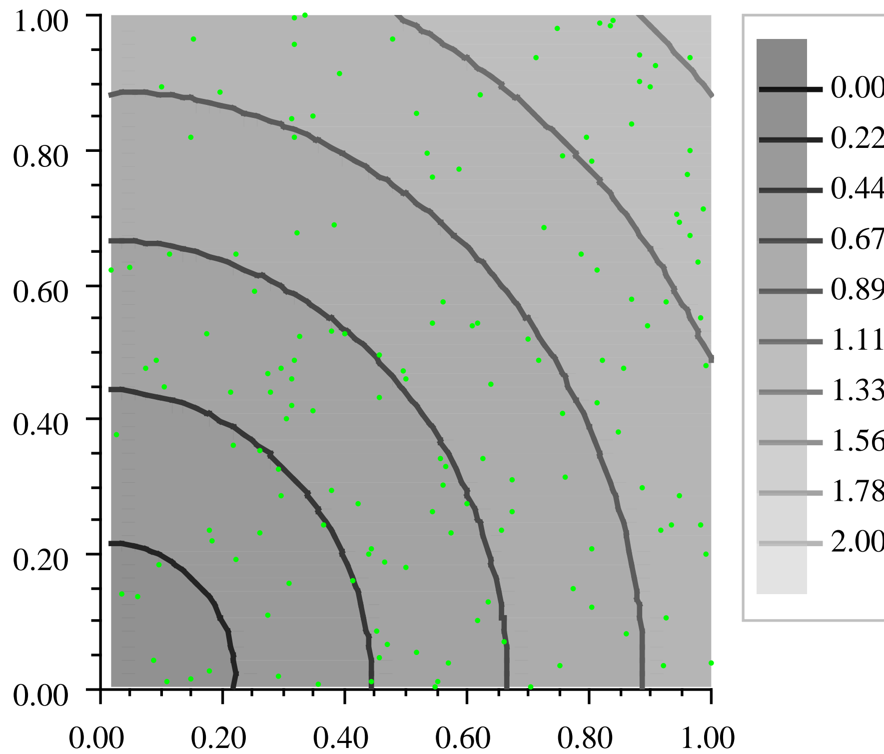
In this example, a contour chart is created from 150, randomly choosen, scattered data points. The function is $\sqrt{x^2+y^2}$, so the level curve should be circles.

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) {
        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(java.awt.Color.green);
        dataPoints.setMarkerSize(0.5);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        ContourEx2.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Contour.Legend class

`public class com.imsl.chart.Contour.Legend extends com.imsl.chart.AxisXY`

A legend for a contour chart.

This legend should be used for contour charts, instead of usual chart legend. The “Viewport” attribute

for this node is set to [0.83,0.98] by [0.1,0.6].

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.

Parameter

`draw` – the `Draw` object to be painted

ErrorBar class

`public class com.imsl.chart.ErrorBar extends com.imsl.chart.Data`

Data points with error bars.

Fields

DATA_TYPE_ERROR_X

`static final public int DATA_TYPE_ERROR_X`

Value for attribute "DataType" indicating that this is a horizontal error bar. Used in connection with ErrorBar nodes.

DATA_TYPE_ERROR_Y

`static final public 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

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

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.

Parameter

`draw` – the `Draw` object to be painted

setHigh

`public void setHigh(double[] value)`

Description

Convenience routine to set the "High" attribute.

Parameter

`value` – an `double` array which contains the "High" values.

setLow

`public void setLow(double[] value)`

Description

Convenience routine to set the "Low" attribute.

Parameter

`value` – an `double` array which contains the "Low" values.

Example: ErrorBar Chart

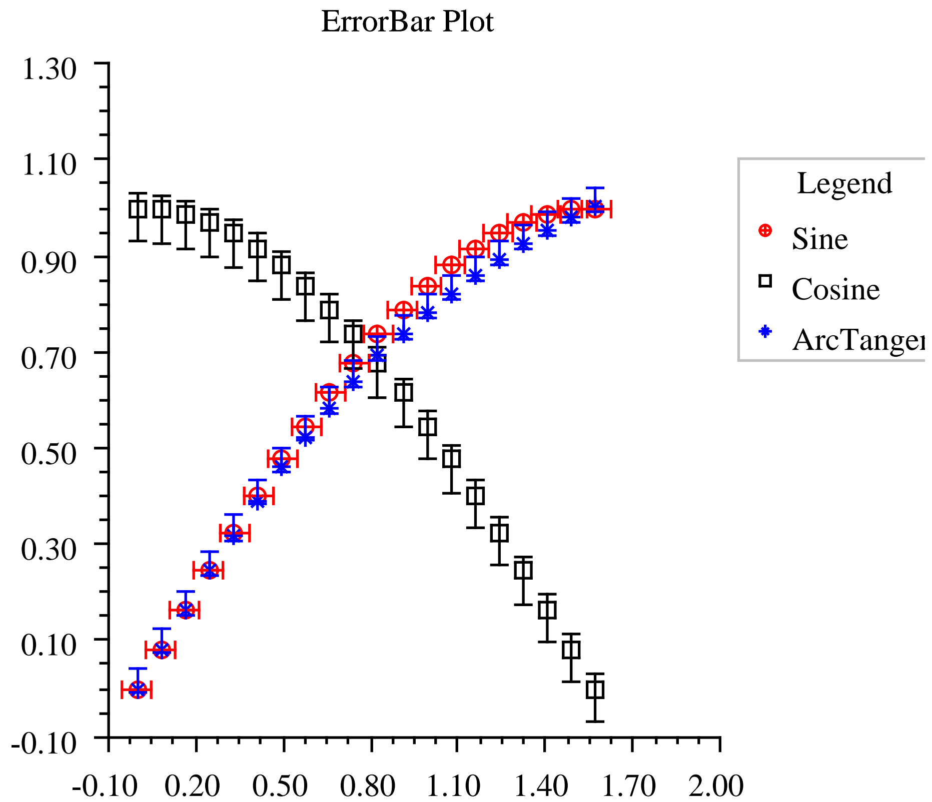
An ErrorBar chart is constructed in this example. Three data sets are used and a legend is added to the chart. This class can be used either as an applet or as an application.

```
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(Data.DATA_TYPE_MARKER);
        d2.setDataType(Data.DATA_TYPE_MARKER);
        d3.setDataType(Data.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(ErrorBar.DATA_TYPE_ERROR_X);
        //  Set Data Type to Error_Y
        ebar2.setDataType(ErrorBar.DATA_TYPE_ERROR_Y);
        ebar3.setDataType(ErrorBar.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"));
        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.setVisible(true);
    }
}
```

Output



HighLowClose class

```
public class com.imsl.chart.HighLowClose extends com.imsl.chart.Data
```

High-low-close plot of stock data.

Field

DAY

`static final public long DAY`

Milliseconds per day

Constructors

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, 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, 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.

HighLowClose

```
public HighLowClose(AxisXY axis, 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.

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

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.

Parameter

`draw` – the `Draw` object to be painted

setClose

```
public void setClose(double[] value)
```

Description

Sets the attribute "Close".

Parameter

`value` – a `double` array of closing stock prices.

setDateAxis

```
public void setDateAxis(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.

Parameter

`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.

Parameter

`value` – an `double` array of high stock prices.

setLow

```
public void setLow(double[] value)
```

Description

Convenience routine to set the “Low” attribute.

Parameter

`value` – an `double` array of low stock prices.

setOpen

```
public void setOpen(double[] value)
```

Description

Sets the attribute “Open”.

Parameter

`value` – a `double` array of opening stock prices.

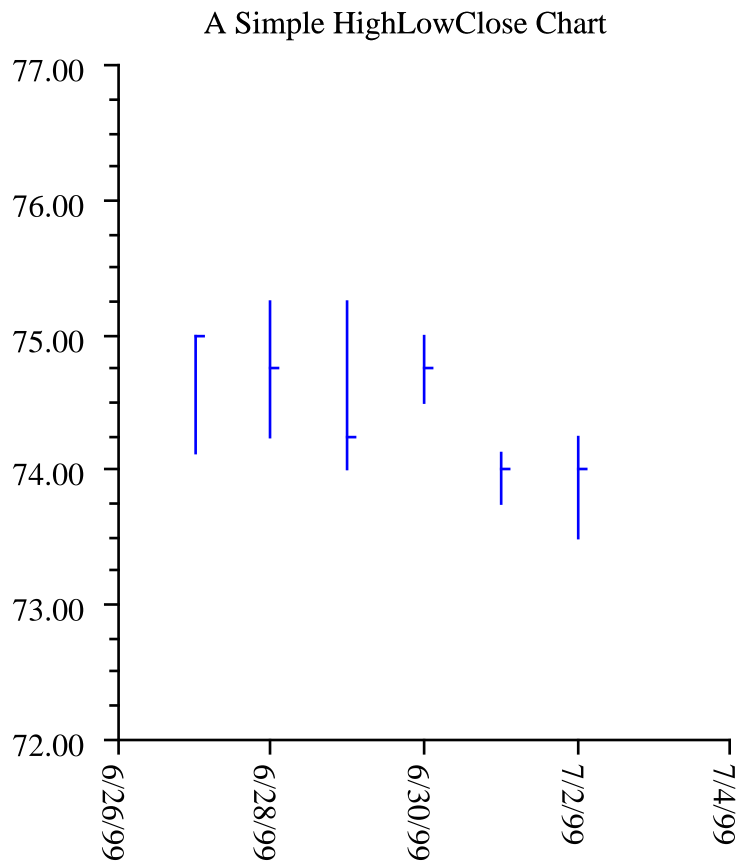
Example: High-Low-Close Chart

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.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(java.awt.Color.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.setVisible(true);
    }
}
```

Output



Candlestick class

`public class com.imsl.chart.Candlestick extends com.imsl.chart.HighLowClose`

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.

Constructors

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.

Candlestick

```
public Candlestick(AxisXY axis, 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.

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.

Parameter

`draw` – the `Draw` object to be painted

CandlestickItem class

`public class com.imsl.chart.CandlestickItem extends com.imsl.chart.Data`

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.

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.

Parameter

`draw` – the `Draw` object to be painted

SplineData class

`public class com.imsl.chart.SplineData extends com.imsl.chart.Data`

A data set created from a Spline.

Constructor

SplineData

`public SplineData(ChartNode parent, 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

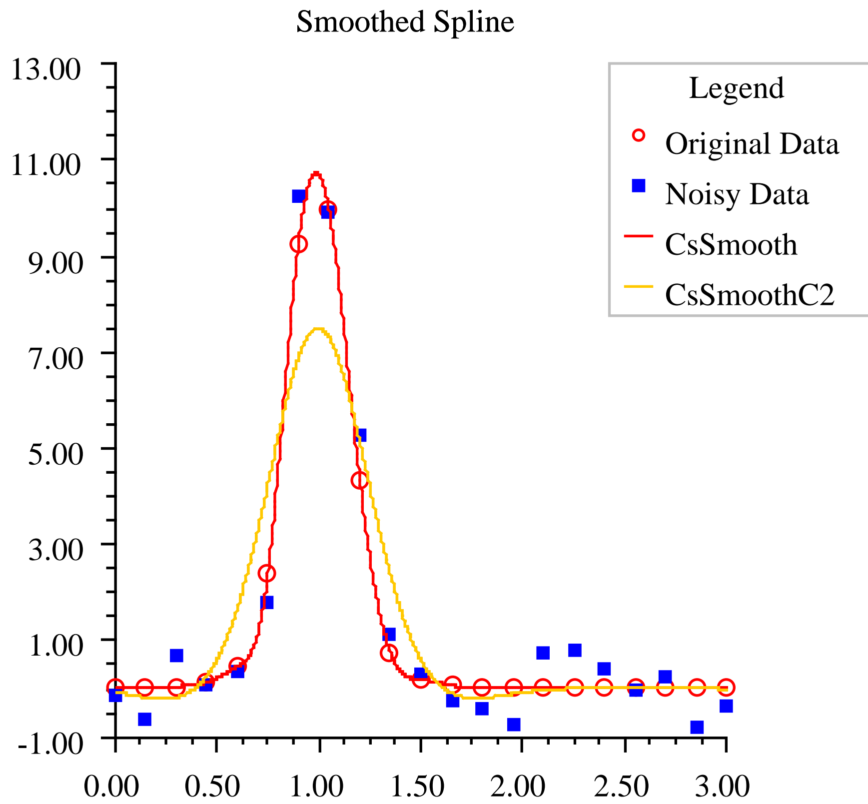
This example makes use of the SplineData class as well as the two spline smoothing classes in the package com.imsl.math. This class can be used either as an applet or as an application.

```
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;

    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.setVisible(true);
    }
}
```

Output



Bar class

`public class com.imsl.chart.Bar extends com.imsl.chart.Data`

A bar chart.

The class `Bar` has children of class `com.imsl.chart.BarItem` (p. 1726) . The attribute “BarItem” in class `Bar` is set to the `BarItem` array of children.

Constructors

Bar

`public Bar(AxisXY axis)`

Description

Constructs a bar chart.

Parameter

`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

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

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.

Parameter

`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.

Parameter

`draw` – the `Draw` object to be painted

setBarData

```
public void setBarData(double[][][] value)
```

Description

Convenience routine to set the "BarData" attribute.

Parameter

`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(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.

Parameter

`labels` – a `String` array with which to label the axis. The number of labels must equal the number of items.

setLabels

```
public void setLabels(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.

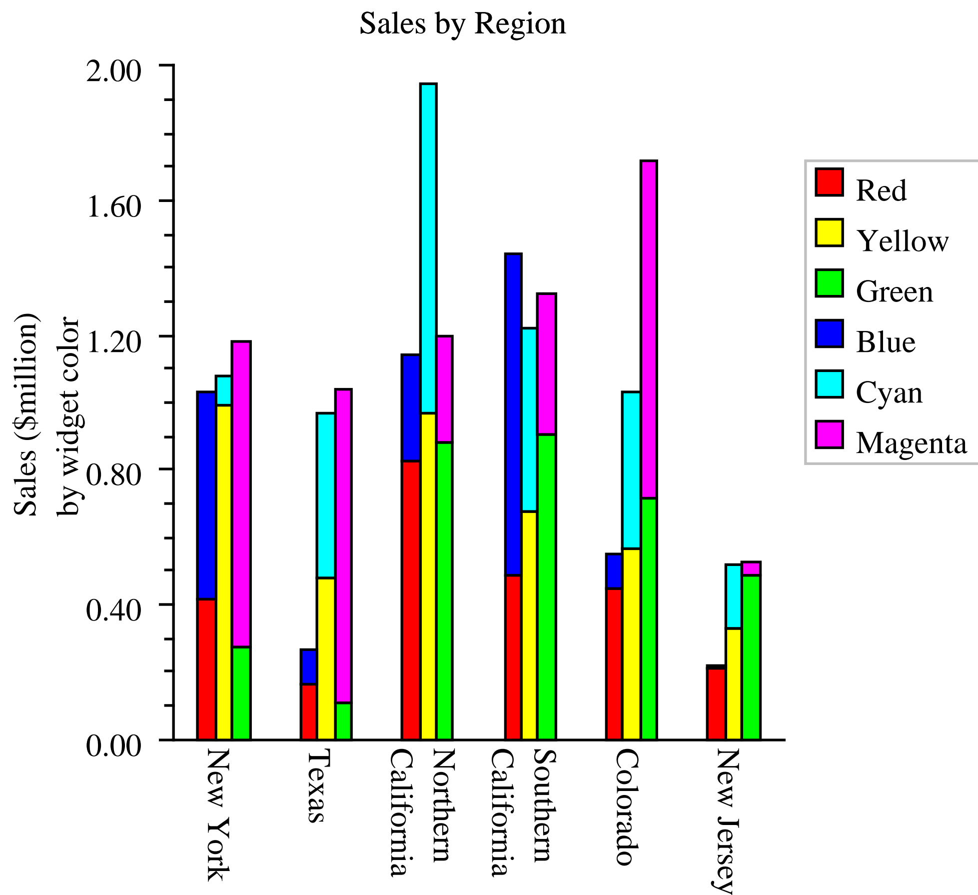
Example: Stacked Bar Chart

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 y[][][] = new double[nStacks][nGroups][nItems];

        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.setVisible(true);
    }
}
```

Output



BarItem class

`public class com.imsl.chart.BarItem extends com.imsl.chart.Data`

A single bar in a bar chart.

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. The argument `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.

Parameter
`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

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.

Parameter
`draw` – the `Draw` object to be painted

BarSet class

`public class com.imsl.chart.BarSet extends com.imsl.chart.ChartNode`

A set of bars in a bar chart.

A `BarSet is 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.`

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

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.

Parameter

`index` – an `int` which specifies the index

Returns

a `BarItem` associated with the specified index

paint

```
public void paint(Draw draw)
```

Pie class

```
public class com.imsl.chart.Pie extends com.imsl.chart.Axis
```

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.

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].

Parameter

`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.

Parameter

`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.

Parameter

`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

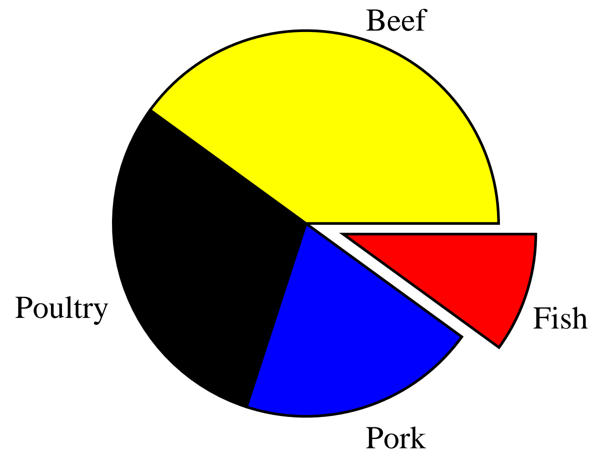
Initializes the mappings between user and coordinate space. This must be called whenever the screen size, the window or the viewport may have changed. Generally, it is safest to call this each time the chart is repainted.

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;

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.setVisible(true);
    }
}
```

Output



PieSlice class

`public class com.imsl.chart.PieSlice extends com.imsl.chart.Data`

One wedge of a pie chart.

`com.imsl.chart.Pie` (p. 1728) creates PieSlice objects as its children, one per pie wedge. A specific

slice can be retrieved using the method `com.imsl.chart.Pie.getPieSlice` (p. 1729) . All of the slices can be retrieved using the method `com.imsl.chart.Pie.getPieSlice` (p. 1729) .

The drawing of the slice is controlled by the fill attributes in this node.

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

Dendrogram class

`public class com.imsl.chart.Dendrogram extends com.imsl.chart.Data`

A Dendrogram chart for cluster analysis.

Constructors

Dendrogram

`public Dendrogram(AxisXY axis, ClusterHierarchical clusterHierarchical)`

Description

Constructs a vertical dendrogram chart using supplied ClusterHierarchical object.

Parameters

`axis` – the `AxisXY` parent of this node

`clusterHierarchical` – a `ClusterHierarchical` object

Dendrogram

```
public Dendrogram(AxisXY axis, ClusterHierarchical clusterHierarchical, int
type)
```

Description

Constructs a dendrogram chart using supplied ClusterHierarchical object.

Parameters

`axis` – the `AxisXY` parent of this node

`clusterHierarchical` – a `ClusterHierarchical` object

`type` – an `int` which specifies the DendrogramType. Legal values are `DENDROGRAM_TYPE_VERTICAL` or `DENDROGRAM_TYPE_HORIZONTAL`.

Dendrogram

```
public Dendrogram(AxisXY axis, double[] clusterLevel, int[] leftSons, int[]
rightSons)
```

Description

Constructs a vertical dendrogram chart using supplied data.

Parameters

`axis` – the `AxisXY` parent of this node

`clusterLevel` – a `double` array which contains the levels at which the clusters are joined

`leftSons` – an `int` array which contains the left sons of each merged cluster

`rightSons` – an `int` array which contains the right sons of each merged cluster

Dendrogram

```
public Dendrogram(AxisXY axis, double[] clusterLevel, int[] leftSons, int[]
rightSons, int type)
```

Description

Constructs a dendrogram chart using supplied data.

Parameters

`axis` – the `AxisXY` parent of this node

`clusterLevel` – a `double` array which contains the levels at which the clusters are joined

`leftSons` – an `int` array which contains the left sons of each merged cluster

`rightSons` – an `int` array which contains the right sons of each merged cluster

`type` – an `int` which specifies the DendrogramType. Legal values are `DENDROGRAM_TYPE_VERTICAL` or `DENDROGRAM_TYPE_HORIZONTAL`.

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

getCoordinates

```
public double[][] getCoordinates()
```

Description

Convenience routine to get the “Coordinates” attribute.

Returns

the `double`[][] array of coordinates.

getLeftSons

```
public int[] getLeftSons()
```

Description

Convenience routine to get the “LeftSons” attribute.

Returns

the `int` array of left sons.

getLevels

```
public double[] getLevels()
```

Description

Convenience routine to get the “Levels” attribute.

Returns

the `double` array of cluster levels.

getOrder

```
public int[] getOrder()
```

Description

Convenience routine to get the “Order” attribute.

Returns

an `int` array of the order of clusters as they appear in the dendrogram.

getRightSons

```
public int[] getRightSons()
```

Description

Convenience routine to get the “RightSons” attribute.

Returns

an `int` array of right sons.

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.

Parameter

`draw` – the `Draw` object to be painted

setCoordinates

```
public void setCoordinates(double[][] value)
```

Description

Convenience routine to set the “Coordinates” attribute.

Parameter

`value` – a `double`[][] array of coordinates.

setLabels

```
public void setLabels(String[] labels)
```

Description

Sets up the axis labels for dendrogram plot. This turns off autoscaling on the axis and sets the Window attribute depending on the number of points being plotted.

Note that user-defined labels will be re-ordered to match the order of the clusters displayed in the plot.

Parameter

`labels` – a `String` array with which to label the axis. The number of labels must equal the number of items.

setLeftSons

```
public void setLeftSons(int[] value)
```

Description

Convenience routine to set the “LeftSons” attribute.

Parameter

`value` – an `int` array of left sons.

---

setLevels

```
public void setLevels(double[] value)
```

Description

Convenience routine to set the “Levels” attribute.

Parameter

`value` – a `double` array of cluster levels.

---

setLineColor

```
public void setLineColor(Color[] colors)
```

Description

Define colors for individual clusters. The color of the topmost level should be set using `ChartNode.setLineColor(java.awt.Color color)`. This method will color N clusters, where N is the number of elements in the colors[] array.

Parameter

`colors` – a `Color` array which contains each color to use for the subclusters.

---

setLineColor

```
public void setLineColor(String[] colors)
```

Description

Define colors for individual clusters. The color of the topmost level should be set using `ChartNode.setLineColor(String color)`. This method will color N clusters, where N is the number of elements in the colors[] array.

Parameter

`colors` – a `String` array which contains each color to use for the subclusters.

---

setOrder

```
public void setOrder(int[] value)
```

Description

Convenience routine to set the “Order” attribute.

Parameter

`value` – an `int` array of the order of clusters as they appear in the dendrogram.

---

setRightSons

```
public void setRightSons(int[] value)
```

Description

Convenience routine to set the “RightSons” attribute.

Parameter

`value` – an `int` array of right sons.

Example: Dendrogram

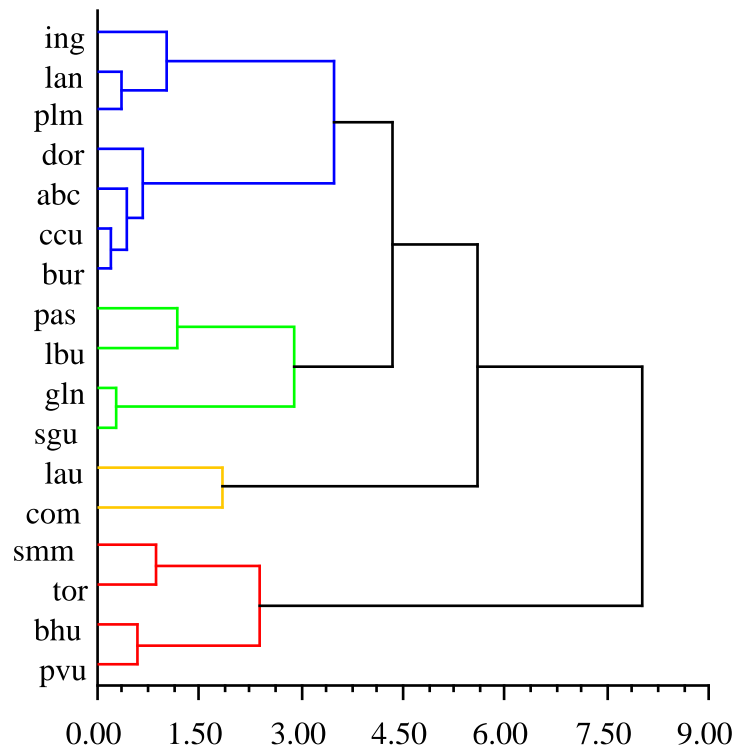
A Dendrogram.

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

public class DendrogramEx1 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) {

        /*
         1998 test data from 17 school districts in Los Angeles County.

         The variables were:
         lep - Proportion of LEP students to total tested
         read - The Reading Scaled Score for 5th Grade
         math - The Math Scaled Score for 5th Grade
         lang - The Language Scaled Score for 5th Grade

         The districts were:
         lau - Los Angeles
         ccu - Culver City
         bhu - Beverly Hills
         ing - Inglewood
         com - Compton
         smm - Santa Monica Malibu
         bur - Burbank
         gln - Glendale
         pvu - Palos Verdes
         sgu - San Gabriel
         abc - Artesia, Bloomfield, and Carmenita
         pas - Pasadena
         lan - Lancaster
         plm - Palmdale
         tor - Torrance
         dow - Downey
         lbu - Long Beach

         input lep read math lang str3 district
         .38 626.5 601.3 605.3 lau
         .18 654.0 647.1 641.8 ccu
```

```
 .07 677.2 676.5 670.5 bhu
 .09 639.9 640.3 636.0 ing
 .19 614.7 617.3 606.2 com
 .12 670.2 666.0 659.3 smm
 .20 651.1 645.2 643.4 bur
 .41 645.4 645.8 644.8 gln
 .07 683.5 682.9 674.3 pvu
 .39 648.6 647.8 643.1 sgu
 .21 650.4 650.8 643.9 abc
 .24 637.0 636.9 626.5 pas
 .09 641.1 628.8 629.4 lan
 .12 638.0 627.7 628.6 plm
 .11 661.4 659.0 651.8 tor
 .22 646.4 646.2 647.0 dow
 .33 634.1 632.0 627.8 lbu
 */
double[][] data = {
    {.38, 626.5, 601.3, 605.3},
    {.18, 654.0, 647.1, 641.8},
    {.07, 677.2, 676.5, 670.5},
    {.09, 639.9, 640.3, 636.0},
    {.19, 614.7, 617.3, 606.2},
    {.12, 670.2, 666.0, 659.3},
    {.20, 651.1, 645.2, 643.4},
    {.41, 645.4, 645.8, 644.8},
    {.07, 683.5, 682.9, 674.3},
    {.39, 648.6, 647.8, 643.1},
    {.21, 650.4, 650.8, 643.9},
    {.24, 637.0, 636.9, 626.5},
    {.09, 641.1, 628.8, 629.4},
    {.12, 638.0, 627.7, 628.6},
    {.11, 661.4, 659.0, 651.8},
    {.22, 646.4, 646.2, 647.0},
    {.33, 634.1, 632.0, 627.8}};

String[] lab = {
    "lau", "ccu", "bhu", "ing", "com", "smm",
    "bur", "gln", "pvu", "sgu", "abc", "pas",
    "lan", "plm", "tor", "dor", "lbu"
};

// 3rd arg in Dissimilarities gives different results for 0,1,2
try {
    Dissimilarities dist = new Dissimilarities(data);
    dist.setScalingOption(Dissimilarities.STD_DEV);
    dist.compute();

    ClusterHierarchical clink
            = new ClusterHierarchical(dist.getDistanceMatrix());
    clink.setMethod(ClusterHierarchical.LINKAGE_WARDS);
    clink.compute();

    AxisXY axis = new AxisXY(chart);

    Dendrogram dc = new Dendrogram(axis, clink,
            Data.DENDROGRAM_TYPE_HORIZONTAL);
```

```
            dc.setLabels(lab);
            dc.setLineColor(new java.awt.Color[]{java.awt.Color.blue,
                java.awt.Color.green, java.awt.Color.red,
                java.awt.Color.orange});
        } catch (com.imsl.IMSLException e) {
            System.out.println(e.getStackTrace());
        }
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        DendrogramEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Polar class

```
public class com.imsl.chart.Polar extends com.imsl.chart.Axis
```

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. The “Viewport” attribute for this node is set to [0.1,0.9] by [0.1,0.9].

Constructor

Polar

`public Polar(Chart chart)`

Description

Create an AxisPolar.

Parameter

`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.

Parameter

`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.

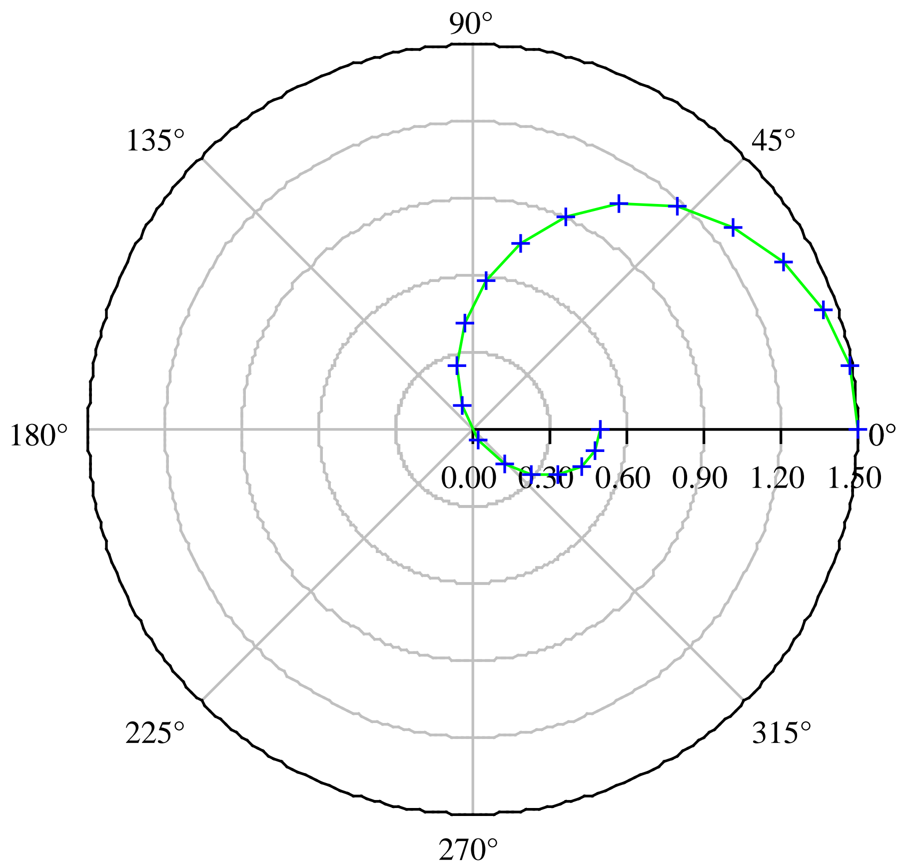
Example: Polar Chart

A simple Polar chart is constructed in this example. The function r = 0.5 + cos(theta), for 0 = theta = p is plotted. This class extends JFrameChart, which manages the window.

```
import com.imsl.chart.*;
import java.awt.Color;

public class PolarEx1 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) {
        Polar axis = new Polar(chart);

        double r[] = new double[20];
        double theta[] = new double[r.length];
        for (int k = 0; k < r.length; k++) {
            theta[k] = Math.PI * k / (r.length - 1);
            r[k] = 0.5 + Math.cos(theta[k]);
        }
        Data data = new Data(axis, r, theta);
        data.setDataType(Data.DATA_TYPE_MARKER | Data.DATA_TYPE_LINE);
        data.setLineColor(Color.green);
        data.setMarkerColor(Color.blue);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        PolarEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Heatmap class

`public class com.imsl.chart.Heatmap extends com.imsl.chart.Data`

`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.

Constructors

Heatmap

```
public Heatmap(AxisXY axis, double xmin, double xmax, double ymin, double ymax,
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

$$s = \frac{t - \text{zmin}}{\text{zmax} - \text{zmin}}$$

.

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. The argument `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.

Parameter

`range` – a `double` array which contains 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 `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.

Parameter

`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`.

Parameter

`colorMap` – The `Colormap` object's "ColorMap" value.

setHeatmapLabels

```
public void setHeatmapLabels(Text[][] labels)
```

Description

Sets the value of the "HeatmapLabels" attribute.

Parameter

`labels` – A two-dimensional array of `com.imsl.chart.Text` (p. 1646) objects that are used to set the "HeatmapLabels" attribute.

setHeatmapLabels

```
public void setHeatmapLabels(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.

Parameter

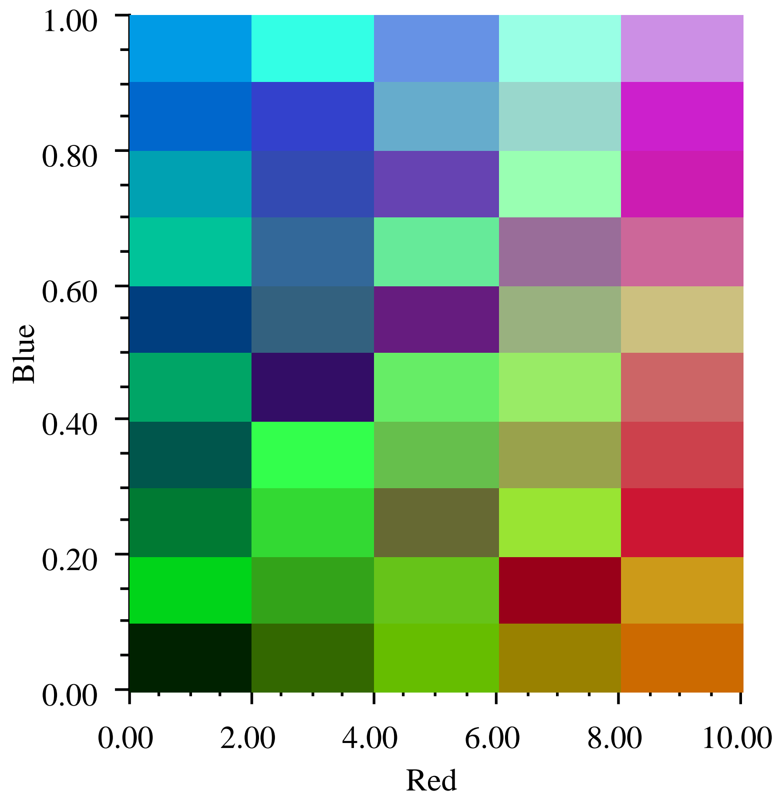
`labels` – A two-dimensional array of `String` objects used to create the two dimensional array of `Text` objects that is the value of the attribute. The array of labels and the array of `Text` objects have the same shape.

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) {
        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[]) {
        JFrameChart frame = new JFrameChart();
        HeatmapEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Example: Heatmap from Color array

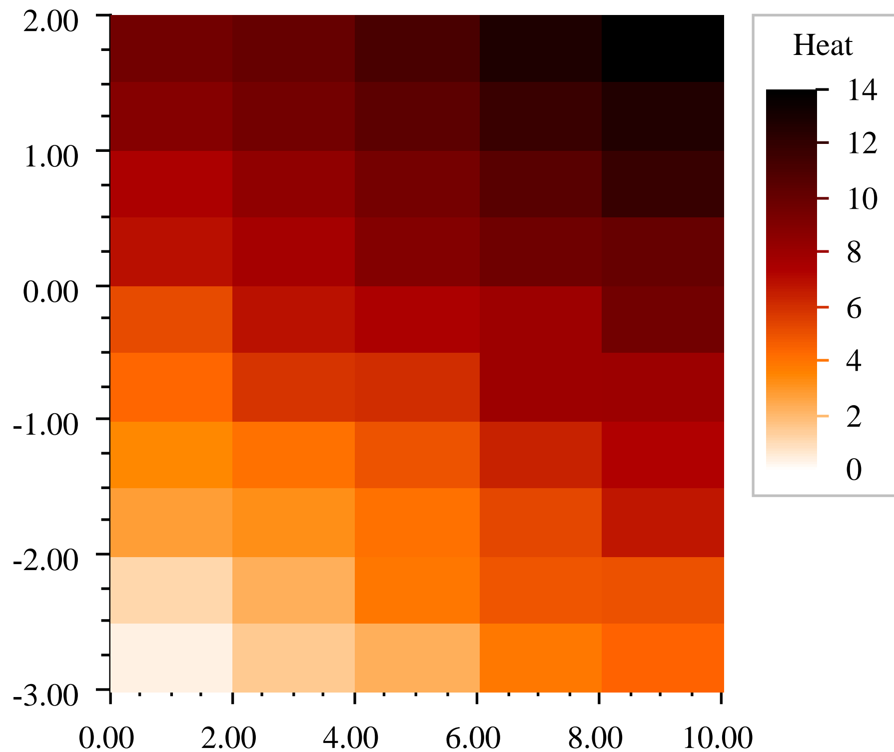
A 5 by 10 data array is created by linearly interpolating from the lower left corner to the upper right corner and adding in a uniform random variable. A red temperature color map is used. This maps the minimum data value to light green and the maximum data value to dark green.

The legend is enabled by setting its paint attribute to true.

```
import com.imsl.chart.*;
import java.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) {
        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,
                fmin, fmax, data, Colormap.RED_TEMPERATURE);
        heatmap.getHeatmapLegend().setPaint(true);
        heatmap.getHeatmapLegend().setTitle("Heat");
        heatmap.getHeatmapLegend().setTextFormat("0");
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        HeatmapEx2.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Example: Heatmap with Labels

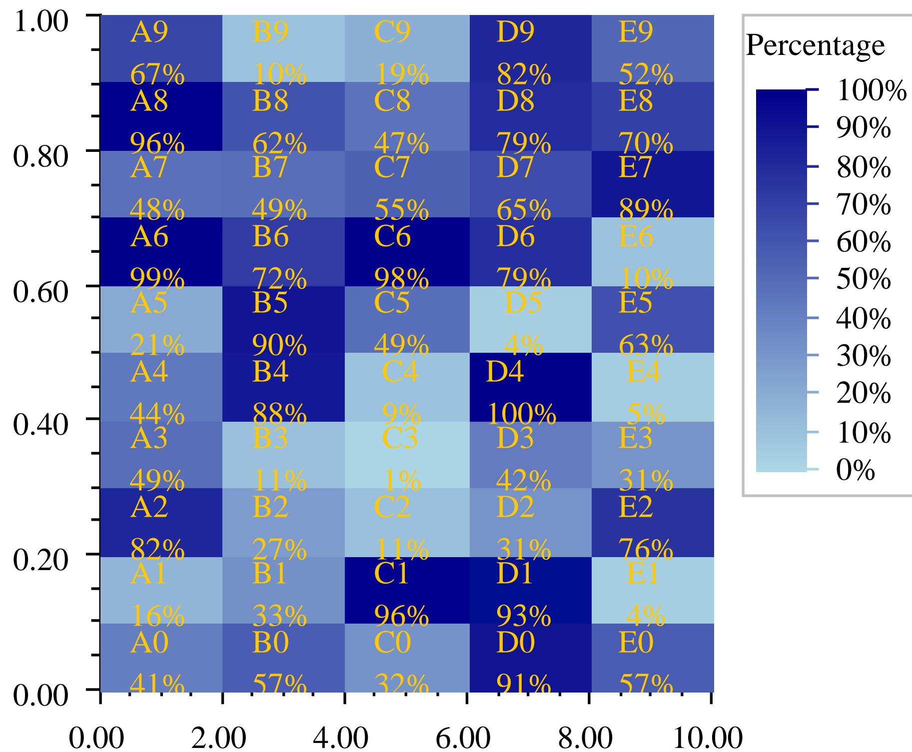
A 5 by 10 array of random data is created and a similarly sized array of strings is also created. These labels contain spreadsheet-like indices and the random data value expressed as a percentage.

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.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) {
        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(java.awt.Color.orange);
        heatmap.getHeatmapLegend().setPaint(true);
        heatmap.getHeatmapLegend().setTextFormat(format);
        heatmap.getHeatmapLegend().setTitle("Percentage");
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        HeatmapEx3.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Heatmap.Legend class

`public class com.imsl.chart.Heatmap.Legend extends com.imsl.chart.AxisXY`

A legend for use with a heatmap.

This `Legend` should be used with `heatmaps`, rather than the usual chart legend.The “Viewport” attribute

for this node is set to [0.83,0.98] by [0.1,0.6].

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.

Parameter

`draw` – The `Draw` object to be painted.

Treemap class

`public class com.imsl.chart.Treemap extends com.imsl.chart.Data`

`Treemap` creates a chart from two arrays of double precision values or one data array and one array of `java.awt.Color` values. The size of each element is scaled using the first input array. The second array of values or colors is used to shade the corresponding area.

The algorithm is adapted from Bruls, Mark and Huizing, Kees and Wijk, Jarke J. van (2000) *Squarified Treemaps*. In Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization.

Fields

AUTOMATIC

`static final public int AUTOMATIC`

Flag to set the treemap orientation automatically.

COLUMNFIRST

`static final public int COLUMNFIRST`

Flag to set the treemap orientation drawing columns first.

ROWFIRST

`static final public int ROWFIRST`

Flag to set the treemap orientation drawing rows first.

Constructors

Treemap

`public Treemap(AxisXY axis, double[] data, Color[] colors)`

Description

Constructs a treemap using supplied data and colors array.

Parameters

`axis` – the `AxisXY` parent of this node

`data` – the area values for the treemap in decreasing order

`colors` – an array of colors for each area

Treemap

`public Treemap(AxisXY axis, double[] data, double[] shades, Colormap colormap)`

Description

Constructs a treemap using supplied data and a colormap.

Parameters

`axis` – the `AxisXY` parent of this node

`data` – the area values for the treemap in decreasing order

`shades` – an array of values to use for shading each area

`colormap` – a `Colormap` to use for the shading

Methods

dataRange

`public void dataRange(double[] range)`

Description

Updates the data range, `range = {xmin,xmax,ymin,ymax}` or `range = {xmin,xmax,ymin,ymax,zmin,zmax}`. The entries in the range are updated to reflect the extent of the data in this node. The argument `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. The third dimension is not required.

Default values are `{0,100,0,100}` for the data extents and the `z` range is comupted from the color data.

Parameter

`range` – a `double` array which contains the updated range, `{xmin,xmax,ymin,ymax}` or `{xmin, xmax,ymin,ymax,zmin,zmax}`

getColormap

`public Colormap getColormap()`

Description

Returns the value of the "Colormap" attribute. This is the `Colormap` associated with this `Treemap`.

Returns

The `Colormap` value of the "Colormap" attribute, if defined. Otherwise, `null` is returned.

getLabels

```
public Text[] getLabels()
```

Description

Returns the value of the "TreemapLabels" attribute.

Returns

An array of `Text` objects that are the value of the "TreemapLabels" attribute, if defined. Otherwise, `null` is returned.

getOrientation

```
public int getOrientation()
```

Description

Gets the value of the "Orientation" attribute.

Returns

One of `AUTOMATIC`, `ROWFIRST` or `COLUMNFIRST`. The default behavior is `AUTOMATIC` and filling the graph is based on the aspect ratio of the parent Axis object such that if the height is less than the width, columns are drawn first; otherwise rows are drawn first.

getTreemapLegend

```
public Treemap.Legend getTreemapLegend()
```

Description

Returns the treemap 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.getTreemapLegend().setPaint(true);`

Returns

The `Legend` object associated with the `Treemap`.

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.

Parameter

`draw` – the `Draw` object to be painted

setAxisRange

```
public void setAxisRange(double[] range)
```

Description

Set the axis range, `range = {xmin,xmax,ymin,ymax}`. The default value is `{0, 100, 0, 100}`. In some cases the aspect ratio of the graph should be changed so as to result in an improved layout. Adjust this array to modify the layout in that manner. The actual values are not important, but the values should match the aspect ratio of the final chart for best results.

Parameter

`range` – a `double` array which contains the data range, `{xmin,xmax,ymin,ymax}`

---

setColormap

`public void setColormap(Colormap colorMap)`

Description

Sets the value of the "Colormap" attribute. This is the `Colormap` associated with this `Treemap`.

Parameter

`colorMap` – The `Colormap` object's "ColorMap" value.

---

setLabels

`public void setLabels(Text[] labels)`

Description

Sets the value of the "TreemapLabels" attribute.

Parameter

`labels` – an array of `com.imsl.chart.Text` (p. 1646) objects that are used to set the "TreemapLabels" attribute.

---

setLabels

`public void setLabels(String[] labels)`

Description

Sets the value of the "TreemapLabels" attribute. The value of the "TreemapLabels" 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.

Parameter

`labels` – An array of `String` objects used to create the array of `Text` objects that is the value of the attribute.

---

setOrientation

`public void setOrientation(int value)`

Description

Sets the value of the "Orientation" attribute.

Parameter

`value` – One of `AUTOMATIC`, `ROWFIRST`, or `COLUMNFIRST`. The default behavior is `AUTOMATIC` and filling the graph is based on the aspect ratio of the parent Axis object such that if the height is less than the width, columns are drawn first; otherwise rows are drawn first.

setZRange

`public void setZRange(double[] range)`

Description

Set the Z data (shading) range, `range = {zmin,zmax}`. The default value is computed from the input `shades` array or else is {0,1}.

Parameter

`range` – a `double` array which contains the data range, {`zmin,zmax`}

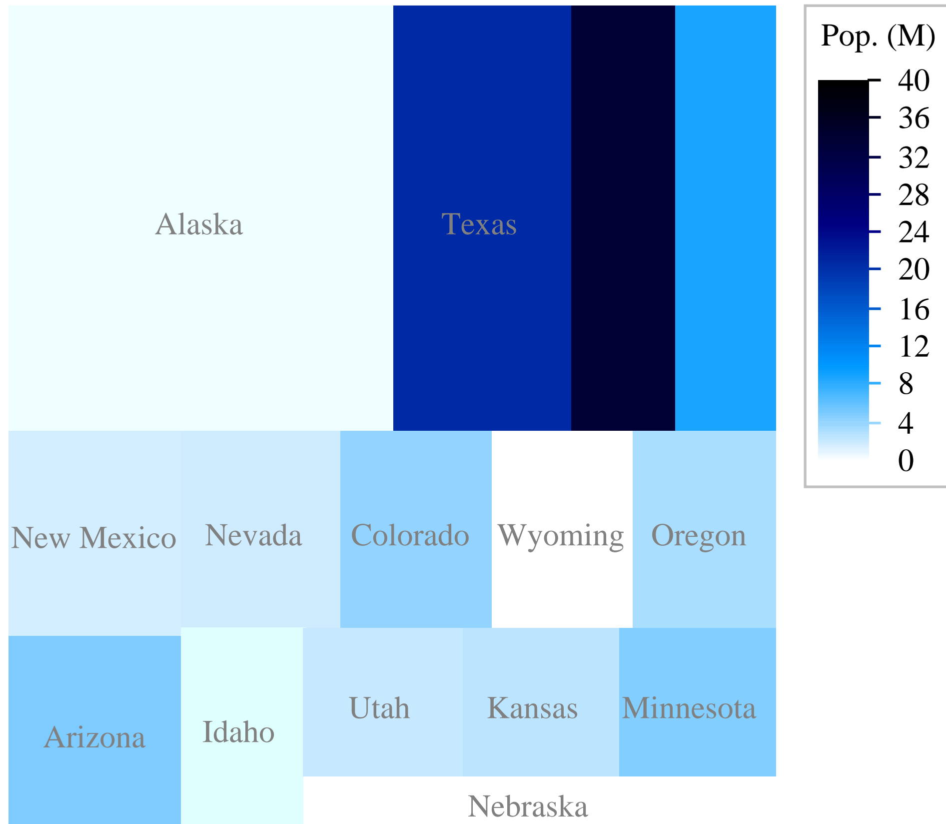
Example 1: Treemap

A treemap is constructed from area and population data of the 15 largest US states. Each rectangle is proportional to the state's area (in square miles) and is shaded by its population (in millions).

```
import com.imsl.chart.*;

public class TreemapEx1 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) {
        AxisXY axis = new AxisXY(chart);

        double[] areas = {
            570374, 261914, 155973, 145556, 121364,
            113642, 109806, 103729, 97105, 96002,
            82751, 82168, 81823, 79617, 76878
        };

        double[] population = {
            0.626932, 20.851820, 33.871648, 9.02195, 1.819046,
            5.130632, 1.998257, 4.301261, 0.493782, 3.421399,
            1.293953, 2.233169, 2.688418, 4.919479, 0.1711263
        };

        String[] names = {
            "Alaska", "Texas", "California", "Montana", "New Mexico",
            "Arizona", "Nevada", "Colorado", "Wyoming", "Oregon",
            "Idaho", "Utah", "Kansas", "Minnesota", "Nebraska"
        };
```

```
        Treemap treemap = new Treemap(axis, areas, population,
                Colormap.BLUE_WHITE);
        treemap.setZRange(new double[]{0, 40});
        treemap.setLabels(names);
        treemap.setTextColor(java.awt.Color.gray);
        treemap.getTreemapLegend().setPaint(true);
        treemap.getTreemapLegend().setTitle("Pop. (M)");
        treemap.getTreemapLegend().setTextFormat("0");
        axis.setViewport(0.05, 0.8, 0.1, 0.95);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        TreemapEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



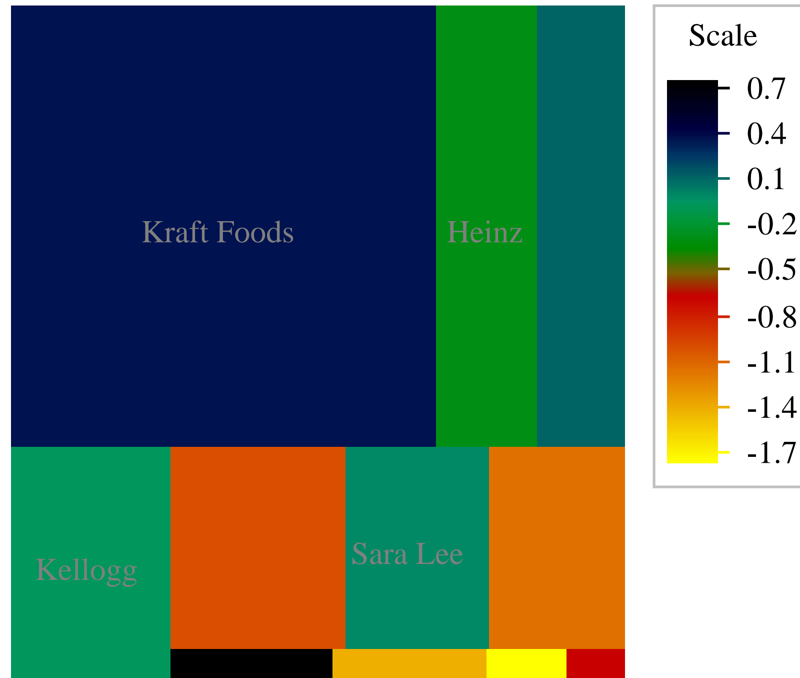
Example 2: Treemap with unsorted data

A treemap is constructed from business analytics data. The area is proportional to the company's sales volume and the color scale maps to a performance indicator. The raw data are unsorted and must be sorted by the area values before creating the chart.

```
import com.imsl.chart.*;

public class TreemapEx2 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) {
        AxisXY axis = new AxisXY(chart);

        double[] areas = {
            834, 11359, 1621, 12282, 14646, 8686,
            12114, 61402, 582, 9448, 1678
        };
        double[] perf = {
            -1.75, -.99, -1.4, 0.12, -0.28, -1.14,
            -0.06, 0.37, -0.66, 0, 0.75};
        String[] labels = {
            "Amer. It. Pasta", "Campbell Soup", "Dean Foods",
            "General Mills", "Heinz", "Hershey Foods", "Kellogg",
            "Kraft Foods", "Ralston Purina", "Sara Lee", "Suiza Foods"
        };

        // sort data
        double[] s_perf = new double[perf.length];
        String[] s_lab = new String[perf.length];
        int[] idx = new int[perf.length];
        com.imsl.stat.Sort.descending(areas, idx);
        for (int i = 0; i < perf.length; i++) {
            s_perf[i] = perf[idx[i]];
            s_lab[i] = labels[idx[i]];
        }

        Treemap treemap = new Treemap(axis, areas, s_perf,
                Colormap.BLUE_GREEN_RED_YELLOW);
        treemap.setLabels(s_lab);
        treemap.setTextColor(java.awt.Color.gray);
        treemap.getTreemapLegend().setPaint(true);
        treemap.getTreemapLegend().setTitle("Scale");
        treemap.getTreemapLegend().setTextFormat("0.0");
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        TreemapEx2.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Treemap.Legend class

`public class com.imsl.chart.Treemap.Legend extends com.imsl.chart.AxisXY`

A legend for use with a treemap.

This `Legend` should be used with `treemaps`, rather than the usual chart legend.The “Viewport” attribute

for this node is set to [0.83,0.98] by [0.1,0.6].

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.

Parameter

`draw` – The `Draw` object to be painted.

Colormap interface

`public interface com.imsl.chart.Colormap`

`Colormaps` are mappings from the unit interval to `Colors`. They are a one-dimensional parameterized path through the color cube.

Fields

BLUE

`static final public Colormap BLUE`

Linear blue colormap.

BLUE_GREEN_RED_YELLOW

`static final public Colormap BLUE_GREEN_RED_YELLOW`

Blue/green/red/yellow colormap.

BLUE_RED

`static final public Colormap BLUE_RED`

Blue/red colormap.

BLUE_WHITE

`static final public Colormap BLUE_WHITE`

Blue/white colormap.

BW_LINEAR

```
static final public Colormap BW_LINEAR
```

Black and white (grayscale) colormap.

GREEN

```
static final public Colormap GREEN
```

Linear green colormap.

GREEN_PINK

```
static final public Colormap GREEN_PINK
```

Green/pink colormap.

GREEN_RED_BLUE_WHITE

```
static final public Colormap GREEN_RED_BLUE_WHITE
```

Green/red/blue/white colormap.

GREEN_WHITE_EXPONENTIAL

```
static final public Colormap GREEN_WHITE_EXPONENTIAL
```

Exponential green/white colormap.

GREEN_WHITE_LINEAR

```
static final public Colormap GREEN_WHITE_LINEAR
```

Linear green/white colormap.

PRISM

```
static final public Colormap PRISM
```

Prism colormap.

RED

```
static final public Colormap RED
```

Linear red colormap.

RED_PURPLE

```
static final public Colormap RED_PURPLE
```

Red/purple colormap.

RED_TEMPERATURE

```
static final public Colormap RED_TEMPERATURE
```

Red temperature colormap.

SPECTRAL

`static final public Colormap SPECTRAL`

Spectral colormap.

STANDARD_GAMMA

`static final public Colormap STANDARD_GAMMA`

Standard gamma colormap.

WB_LINEAR

`static final public Colormap WB_LINEAR`

Black and white (grayscale) colormap, the reverse of `BW_LINEAR`.

Method

color

`public Color color(double t)`

Description

Maps the parameterization interval [0,1] into `Colors`.

Parameter

`t` – A parameter value in the interval [0,1].

Returns

A `Color` value corresponding to `t`.

Exception

`IllegalArgumentException` is thrown if `t` is outside of the range [0,1]

ChartXML class

`public class com.imsl.chart.xml.ChartXML implements org.xml.sax.ErrorHandler`

Create a Chart from an XML file.

Constructors

ChartXML

`public ChartXML(String filename) throws SAXException, IOException, ParserConfigurationException`

Description

Creates a ChartXML from an XML file. A validating XML parser is used.

Parameter

`filename` – is the name of a file containing an XML description of a chart.

Exceptions

`IOException` if there is a problem reading the file.

`SAXException` if there is a problem parsing the file.

`ParserConfigurationException` if there is a problem configuring the XML parser.

ChartXML

`public ChartXML(Document document) throws SAXException`

Description

Creates a ChartXML from a DOM tree.

Parameter

`document` – is a description of a chart.

ChartXML

`public ChartXML(String filename, boolean validating) throws SAXException, IOException, ParserConfigurationException`

Description

Creates a ChartXML from an XML file.

Parameters

`filename` – is the name of a file containing an XML description of a chart.

`validating` – is true if a validating parser is to be used.

Exceptions

`IOException` if there is a problem reading the file.

`SAXException` if there is a problem parsing the file.

`ParserConfigurationException` if there is a problem configuring the XML parser.

ChartXML

`public ChartXML(InputSource source, boolean validating) throws SAXException, IOException, ParserConfigurationException`

Description

Creates a ChartXML from an XML source.

Parameters

`source` – is an InputSource containing an XML description of a chart.

`validating` – is true if a validating parser is to be used.

Exceptions

`IOException` if there is a problem reading the source.

`SAXException` if there is a problem parsing the source.

`ParserConfigurationException` if there is a problem configuring the XML parser.

Methods

error

```
public void error(SAXParseException exception)
```

Description

Receive notification of a recoverable error.

error

```
protected void error(String key, Object[] args)
```

Description

Handles error messages.

Parameters

`key` – is the key to the error message string in the ErrorMessages bundle in this package.

`args` – are the arguments to be filled into the error message string.

fatalError

```
public void fatalError(SAXParseException exception)
```

Description

Receive notification of a non-recoverable error.

get

```
public Object get(String id)
```

Description

Returns a generated object given the id attribute in the XML tag that created the object. This allows charts generated from XML files to be programmatically manipulated.

getChart

```
public Chart getChart()
```

Description

Returns the root node of the chart tree.

getEnumValue

```
static protected Integer getEnumValue(String value)
```

Description

Returns the int corresponding to an enumeration. The resource bundle Enum in this package is searched for the value.

keySet

```
public Set keySet()
```

Description

Returns the Set view of all id's defined in the XML file.

main

```
static public void main(String[] argv) throws Exception
```

Description

Displays a chart created from an XML file. Usage: java com.imsl.chart.xml.ChartXML filename

stringToObject

```
protected Object stringToObject(Class theClass, String value)
```

Description

Converts a String into an Object of the given class.

warning

```
public void warning(SAXParseException exception)
```

Description

Receive notification of a warning.

Chapter 27: Quality Control and Improvement Charts

Types

class ShewhartControlChart 1771
class ControlLimit 1777
class XbarR 1779
class RChart 1785
class XbarS 1789
class SChart 1796
class XmR 1799
class NpChart 1802
class PChart 1805
class CChart 1809
class UChart 1812
class EWMA 1815
class CuSum 1819
class CuSumStatus 1822
class ParetoChart 1830

ShewhartControlChart class

`public class com.imsl.chart.qc.ShewhartControlChart extends com.imsl.chart.Data`

`ShewhartControlChart` is the base class for the Shewhart control charts.

The control limits are generally calculated as

$$\text{center} + k\sigma$$

where the meaning of *center* and σ depend on the specific control chart being drawn. The variable k is the value of the attribute "ControlLimit" associated with the line being drawn. Typically there are three lines with $k = -3, 0, 3$.

If all of the samples sizes are equal, the lines are horizontal. If the sample sizes are unequal, the lines have a "stairstep" pattern. Horizontal lines have a title drawn just above the line and right-adjusted on the chart. The contents of the title is determined by the line's "Title" attribute. If the title contains the placeholder "$\{0\}$", it is replaced by the lines value. This replacement is done using `java.text.MessageFormat` .

Fields

d2

```
static final public double[] d2
```

This field contains $d_{2,n}$ the mean of the ranges of n samples from a Gaussian distribution. The entries for n= 0 and 1 are `NaN` because these values are not defined. Valid entries in the table are for n=2 through 50.

d3

```
static final public double[] d3
```

This field contains $d_{3,n}$ the standard deviation of the ranges of n samples from a Gaussian distribution. The entries for n= 0 and 1 are `NaN` because these values are not defined. Valid entries in the table are for n=2 through 50.

Constructor

ShewhartControlChart

```
public ShewhartControlChart(AxisXY axis)
```

Description

Constructs a Shewhart control chart.

Parameter

`axis` – the `AxisXY` parent of this node

Methods

addCenterLine

```
public ControlLimit addCenterLine()
```

Description

Adds the center line to the control chart and returns the newly added line.

Returns

the `ControlLimit` which draws the center line.

addControlLimit

```
public ControlLimit addControlLimit()
```

Description

Adds a control limit to the chart. These can be used as tolerance limit lines or warning control limit lines.

Returns

a `ControlLimit` object.

addLowerControlLimit

```
public ControlLimit addLowerControlLimit()
```

Description

Creates lower `ControlLimit`, adds it to the control chart, and returns the newly created object.

Returns

the `ControlLimit` which draws the lower control limit line.

addUpperControlLimit

```
public ControlLimit addUpperControlLimit()
```

Description

Creates upper `ControlLimit`, adds it to the control chart, and returns the newly created object.

Returns

the `ControlLimit` which draws the upper control limit line.

addWecoLimits

```
public ControlLimit[] addWecoLimits()
```

Description

Adds lines for the Western Electric Company Rules. These additional lines are at

$$\text{center} + k\sigma \quad \text{for } k = -2, -1, 1, 2$$

The meaning of *center* and σ depend on the chart to which these lines are added. The "LineColor" attribute for these lines is set to yellow.

Returns

an array containing the four added lines. They are in order with the `ControlLimit` corresponding to k = -2 first and the `ControlLimit` corresponding to k = +2 last.

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

getCenter

```
public double getCenter()
```

Description

Returns the value of the attribute “Center”. This is used to position the center line.

Returns

the value of the attribute “Center”. Its default value is 0.

getCenterLine

```
public ControlLimit getCenterLine()
```

Description

Returns the center line.

Returns

the `ControlLimit` which draws the center line.

getControlData

```
public Data getControlData()
```

Description

Returns the `Data` object for the control data.

Returns

a `Data` object for charting the control data.

getLowerControlLimit

```
public ControlLimit getLowerControlLimit()
```

Description

Returns the lower control limit.

Returns

the `ControlLimit` which draws the lower control limit line.

getMeanSampleSize

```
public double getMeanSampleSize()
```

Description

Returns the value of the attribute “MeanSampleSize”. This is set by the method setSampleSize. Its default value is zero.

Returns

the average sample size

getSampleSize

`public int[] getSampleSize()`

Description

Returns the value of the attribute “SampleSize”.

Returns

sampleSize is the value of the attribute “SampleSize”. Its default value is an array of length one containing a one, (`new int[]{1}`).

getUpperControlLimit

`public ControlLimit getUpperControlLimit()`

Description

Returns the upper control limit.

Returns

the `ControlLimit` which draws the upper control limit line.

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.

Parameter

`draw` – a `Draw` object which specifies the chart tree to be rendered on the screen

removeControlLimit

`public void removeControlLimit(ControlLimit line)`

Description

Removes a control limit from the chart.

Parameter

`line` – is the `ControlLimit` object to be removed from this chart.

setCenter

`public void setCenter(double center)`

Description

Sets the value of the attribute “Center”. This is used to position the center line. Its default value is 0.

Parameter

`center` – is the value of the attribute “Center”.

setData

```
public Data setData(double[] y)
```

Description

Sets the data in the control chart. By default, the data points are drawn as filled circle markers connected by a line.

Parameter

`y` – is an array containing the data values.

Returns

a `Data` object containing the data points.

setData

```
public Data setData(int[] y)
```

Description

Sets the integer data in the control chart. By default, the data points are drawn as filled circle markers connected by a line.

Parameter

`y` – is an array containing the data values.

Returns

a `Data` object containing the data points.

setSampleSize

```
public void setSampleSize(int sampleSize)
```

Description

Sets the value of the attribute “SampleSize”. Its default value is an array of length one containing a one, (`new int[]{1}`).

Parameter

`sampleSize` – is the value of the attribute “SampleSize”.

setSampleSize

```
public void setSampleSize(int[] sampleSize)
```

Description

Sets the value of the attribute “SampleSize”.

Parameter

`sampleSize` – is the value of the attribute “SampleSize”.

setX

```
public void setX(double[] x)
```

Description

Sets the x-coordinates of the data. The “X” chart attribute of each `ControlLimit` added to this chart is also set. If not set the values 0, 1, 2, ... are used.

Parameter

`x` – is an array containing the x-coordinates. The length of `x` must equal the length of the data array used to construct this object.

ControlLimit class

```
public class com.imsl.chart.qc.ControlLimit extends com.imsl.chart.Data
```

`ControlLimit` is a control limit line on a process control chart.

This class draw either a horizontal line or a stair step line depending on the value of the attribute “Value”. Its value is an array. If the has a just one entry then a horizontal line is drawn at *y* equal to this value. This line extends across the limit given by the x-axis window attribute. If the array has more than one entry then a stair step line is drawn using the array values as the *y*-coordinates of the stair step line.

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

getControlLimit

```
public double getControlLimit()
```

Description

Returns the value of the attribute “ControlLimit”. This is the *i*-coordinate at which the line is drawn.

getMaximumValue

```
public double getMaximumValue()
```

Description

Returns the maximum value of this control limit line.

Returns

the maximum value for this control limit line. The default maximum value is positive infinity (no upper limit).

getMinimumValue

```
public double getMinimumValue()
```

Description

Returns the minimum value of this control limit line.

Returns

the minimum value for this control limit line. The default minimum value is negative infinity (no lower limit).

getValue

```
public double[] getValue()
```

Description

Returns the value of this control limit line.

Returns

the *y*-coordinate at which this control limit line is drawn.

paint

```
public void paint(Draw draw)
```

Description

Paints the horizontal control limit line as wide as the window.

setControlLimit

```
public void setControlLimit(double controlLimit)
```

Description

Sets the attribute “ControlLimit”.

Parameter

`controlLimit` – is the value of the “ControlLimit” attribute. This is the *y*-coordinate at which the line is drawn. Its default value is zero.

setMaximumValue

```
public void setMaximumValue(double maximumValue)
```

Description

Set the maximum value of this control limit line. The default maximum value is positive infinity (no upper limit).

Parameter

`maximumValue` – is the maximum value for this control limit line.

setMinimumValue

```
public void setMinimumValue(double minimumValue)
```

Description

Set the minimum value of this control limit line. The default minimum value is negative infinity (no lower limit).

Parameter

`minimumValue` – is the minimum value for this control limit line.

setValue

```
public void setValue(double y)
```

Description

Sets the value of this control limit line. The actual value used is subject to minimum and maximum values set as attributes to this object.

Parameter

`y` – is the *y*-coordinate at which this control limit line is drawn.

setValue

```
public void setValue(double[] y)
```

Description

Sets the value of this control limit line to an array of values.

Parameter

`y` – is an array containing the *y*-coordinates of a stair step line.

XbarR class

```
public class com.imsl.chart.qc.XbarR extends
com.imsl.chart.qc.ShewhartControlChart
```

`XbarR` is an *X-bar* chart for monitoring a process using sample ranges.

The control limits are at

$$\bar{\bar{x}} + k\frac{\bar{R}}{d_{2,n}\sqrt{n}}$$

where $\bar{\bar{x}}$ is the grand mean of all of the observations, $\bar{R}$ is the mean of the observed ranges, n is the sample size, and k is the value of the "ControlLimit" attribute for the limit. Additionally, $d_{2,n} = E[R]/\sigma$, where R is the range of data from a Gaussian distribution. Therefore $\bar{R}/d_{2,n}$ is an estimate of the within sample standard deviation. By default, the chart contains an upper control limit with k=3, a lower control limit with k=-3, and a central line equal to $\bar{\bar{x}}$. Additional control limits can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2.

Constructors

XbarR

`public XbarR(AxisXY axis, double[][] x)`

Description

Creates an X-bar chart from sample data using sample ranges.

Parameters

`axis` – the `AxisXY` parent of this node.

`x` – is an array of arrays containing sample data. The data of the i-th sample is in `x[i]`. Each row must have between 2 and 50 entries.

XbarR

`public XbarR(AxisXY axis, int sampleSize, double[] xbar, double[] range)`

Description

Creates an X-bar control chart given the means and ranges for a series of equally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is the number of observations in each sample. It must be between 2 and 50.

`xbar` – is an array containing the mean values for a series of samples.

`range` – is an array containing the ranges of the samples. The arrays `xbar` and `range` must have the same lengths.

XbarR

`public XbarR(AxisXY axis, int[] sampleSize, double[] xbar, double[] range)`

Description

Creates an X-bar control chart given the means and ranges for a series of unequally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is an array containing the nuumber of observations in each sample. Each sample must contain between 2 and 50 observations.

`xbar` – is an array containing the mean values for a series of samples.

`range` – is an array containing the ranges of the samples.

Methods

capabilityIndexCp

```
public double capabilityIndexCp(double lowerSpecificationLimit, double
upperSpecificationLimit)
```

Description

Returns the capability index c_p

$$c_p = \frac{\mathrm{USL} - \mathrm{LSL}}{6\sigma}$$

where LSL and USL are the lower and upper specification limits and

$$\sigma = \frac{\bar{R}}{d_{2,n}\sqrt{n}}$$

.

Parameters

`lowerSpecificationLimit` – is the lower specification limit.

`upperSpecificationLimit` – is the upper specification limit.

Returns

the capability index.

capabilityIndexCpk

```
public double capabilityIndexCpk(double lowerSpecificationLimit, double
upperSpecificationLimit)
```

Description

Returns the capability index c_{pk}

$$c_{pk} = \min\left[\frac{\mathrm{USL} - \bar{x}}{3\sigma}, \frac{\bar{x} - \mathrm{LSL}}{3\sigma}\right]$$

where LSL and USL are the lower and upper specification limits, $\bar{x}$ is the center line, and

$$\sigma = \frac{\bar{R}}{d_{2,n}\sqrt{n}}$$

.

Parameters

`lowerSpecificationLimit` – is the lower specification limit.

`upperSpecificationLimit` – is the upper specification limit.

Returns

the capability index.

createCharts

```
static public ShewhartControlChart[] createCharts(Chart chart, double[][] x)
```

Description

Creates a combined XbarR chart and RChart from data. The viewport of the XbarR chart is [0.2, 0.9] by [0.1, 0.4]. The viewport of the RChart chart is [0.2, 0.9] by [0.5, 0.8].

Parameters

chart – is the Chart object which is the parent of the two charts being created.

x – is an array of arrays containing sample data. The data of the *i*-th sample is in x[i]. Each sample must contain at least 2 and no more than 50 observations.

Returns

an array of length two containing the XBarR chart and the RChart.

createCharts

```
static public ShewhartControlChart[] createCharts(Chart chart, int sampleSize,
double[] xbar, double[] range)
```

Description

Creates a combined XbarR chart and RChart given the means and ranges for a series of equally sized samples. The viewport of the XbarR chart is [0.2, 0.9] by [0.1, 0.4]. The viewport of the RChart chart is [0.2, 0.9] by [0.5, 0.8].

Parameters

chart – is the Chart object which is the parent of the two charts being created.

sampleSize – is the number of observations in each sample. It must be between 2 and 50.

xbar – is an array containing the mean values for a series of samples.

range – is an array containing the ranges of the samples.

Returns

an array of length two containing the XBarR chart and the RChart.

createCharts

```
static public ShewhartControlChart[] createCharts(Chart chart, int[]
sampleSize, double[] xbar, double[] range)
```

Description

Creates a combined XbarR chart and RChart given the means and ranges for a series of unequally sized samples. The viewport of the XbarR chart is [0.2, 0.9] by [0.1, 0.4]. The viewport of the RChart chart is [0.2, 0.9] by [0.5, 0.8].

Parameters

chart – is the Chart object which is the parent of the two charts being created.

sampleSize – is an array containing the number of observations in each sample. Each sample must have between 2 and 50 observations.

xbar – is an array containing the mean values for a series of samples.

range – is an array containing the ranges of the samples.

Returns

an array of length two containing the XBarR chart and the `RChart`.

getRbar

```
public double getRbar()
```

Description

Returns the value of the “Rbar” attribute, the mean of the ranges for a series of samples.

Returns

the mean of the ranges for a series of samples

prePaint

```
public void prePaint()
```

Description

Setup chart with current settings.

setRbar

```
public void setRbar(double rbar)
```

Description

Sets the value of the “Rbar” attribute, the mean of the ranges for a series of samples.

Parameter

`rbar` – is the mean of the ranges for a series of samples.

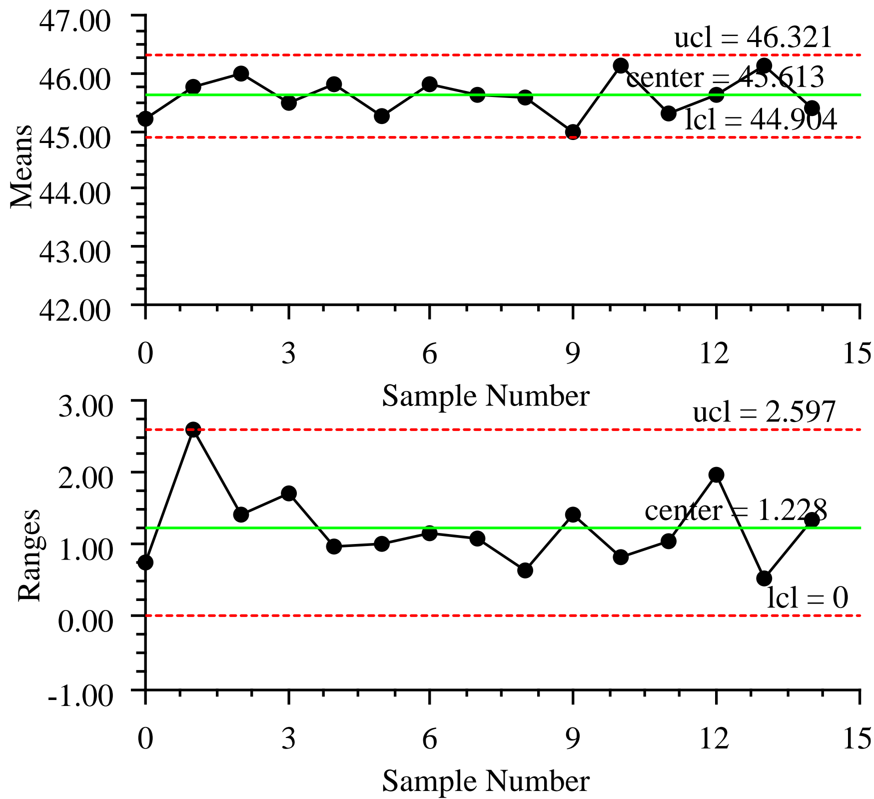
Example: XbarR Chart

During a manufacturing process 15 samples, each containing 5 items, were measured. An XbarR chart was contructed from the 15 sample ranges. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class XbarREx1 extends javax.swing.JApplet {

    static private final double data[][] = {
        {44.73, 45.47, 45.39, 45.33, 45.24},
        {45.57, 46.87, 45.40, 46.68, 44.29},
        {46.39, 45.31, 46.74, 46.06, 45.51},
        {45.54, 46.27, 44.57, 45.36, 45.72},
        {45.58, 45.59, 46.02, 45.45, 46.42},
        {45.91, 45.38, 44.98, 44.91, 45.17},
        {45.98, 45.29, 45.50, 45.77, 46.44},
        {46.30, 45.65, 45.21, 45.43, 45.57},
        {45.77, 45.38, 45.65, 45.25, 45.89},
        {44.10, 45.44, 45.27, 45.53, 44.65},
        {45.95, 46.22, 46.71, 45.92, 45.90},
        {44.87, 44.98, 45.91, 45.18, 45.64},
```

```
        {44.70, 45.89, 46.67, 45.84, 45.07},
        {45.90, 45.80, 46.30, 46.34, 46.34},
        {44.90, 46.23, 45.31, 45.29, 45.16}
    };

    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) {
        ShewhartControlChart control[] = XbarR.createCharts(chart, data);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        XbarREx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



RChart class

```
public class com.imsl.chart.qc.RChart extends
com.imsl.chart.qc.ShewhartControlChart
```

`RChart` is an R chart using sample ranges to monitor the variability of a process. Each sample must contain at least two observations. The range of a sample is the maximum observed value minus the

minimum observed value.

The control limits are at

$$\bar{R} + k\frac{d_{3,n}}{d_{2,n}}\bar{R}$$

$\bar{R}$ is the mean of the observed ranges, n is the sample size, and k is the value of the “ControlLimit” attribute for the line. Additionally, $d_{2,n}$ is the mean of the distribution of the ranges of n samples from the normal distribution with mean zero and standard deviation one. The standard deviation of this distribution is $d_{3,n}$. Therefore

$$\frac{d_{3,n}}{d_{2,n}}\bar{R}$$

is an estimator of the standard deviation of the ranges.

By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line with k=0. Additional control limit lines can be added. The method `addWeco` adds control limit lines with k = -2, -1, 1, 2.

Constructors

RChart

```
public RChart(AxisXY axis, double[][] x)
```

Description

Creates an R chart given sample data.

Parameters

`axis` – the `AxisXY` parent of this node

`x` – is an array of arrays containing sample data. The data of the i-th sample is in `x[i]`. Each sample must contain at least 2 and no more than 50 observations.

Exception

`IllegalArgumentException` is thrown if the number of samples is less than 2 or greater than 50.

RChart

```
public RChart(AxisXY axis, int sampleSize, double[] range)
```

Description

Creates an R chart given the ranges for a series of equally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is the number of observations in each sample. It must be at least 2 and no more than 50.

`range` – is an array containing the data ranges for a series of samples.

RChart

`public RChart(AxisXY axis, int[] sampleSize, double[] range)`

Description

Creates an R chart given the means for a series of equally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is an array containing the number of observations in each sample. It must be at least 2 and no more than 50.

`range` – is an array containing the data ranges for a series of samples.

Method

prePaint

`public void prePaint()`

Description

Setup chart with current settings.

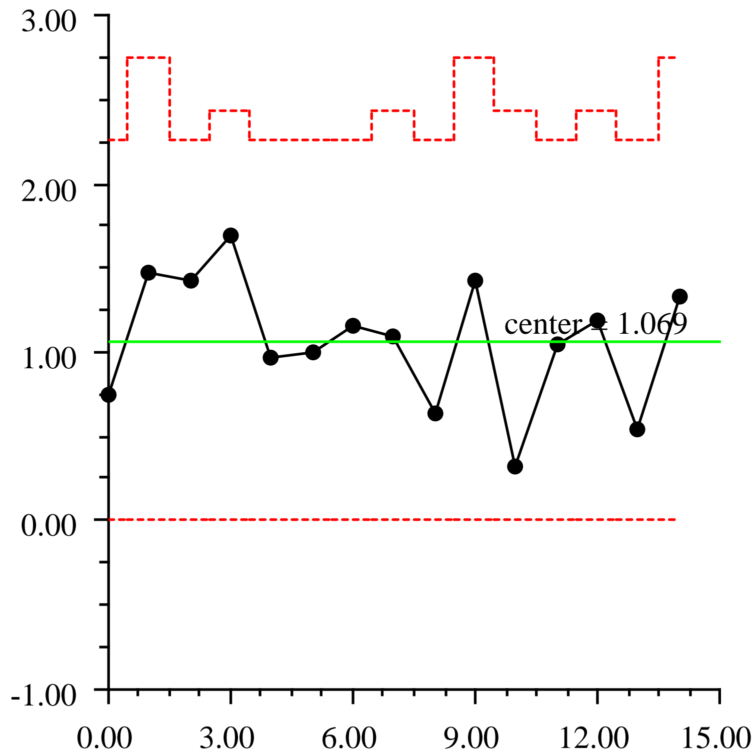
Example: R Chart

During a manufacturing process 15 samples, each containing 3 to 5 items, were measured. An R chart was contructed from the 15 sample ranges. Since the sample sizes are unequal the upper control limit is a stair step line. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class RChartEx1 extends javax.swing.JApplet {

    static private final double data[][] = {
        {44.73, 45.47, 45.39, 45.33, 45.24},
        {45.57, 46.87, 45.40},
        {46.39, 45.31, 46.74, 46.06, 45.51},
        {45.54, 46.27, 44.57, 45.36},
        {45.58, 45.59, 46.02, 45.45, 46.42},
        {45.91, 45.38, 44.98, 44.91, 45.17},
        {45.98, 45.29, 45.50, 45.77, 46.44},
        {46.30, 45.65, 45.21, 45.43},
        {45.77, 45.38, 45.65, 45.25, 45.89},
        {44.10, 45.53, 44.65},
        {45.95, 46.22, 45.92, 45.90},
        {44.87, 44.98, 45.91, 45.18, 45.64},
        {44.70, 45.89, 45.84, 45.07},
        {45.90, 45.80, 46.30, 46.34, 46.34},
        {44.90, 46.23, 45.16}
```

```
    };

    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) {
        AxisXY axis = new AxisXY(chart);
        new RChart(axis, data);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        RChartEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



XbarS class

```
public class com.imsl.chart.qc.XbarS extends
com.imsl.chart.qc.ShewhartControlChart
```

`XbarS` is an *X-bar* chart for monitoring a process using sample standard deviations.

The control limits are at

$$\bar{\bar{x}} + k\frac{\bar{s}}{c_{4,n}\sqrt{n}}$$

where $\bar{\bar{x}}$ is the grand mean of all of the observations, n is the sample size, and k is the value of the "ControlLimit" attribute for the limit. Additionally, $c_{4,n}$ is a factor such that $\bar{s}/c_{4,n}$ is an unbiased estimator of the within sample standard deviation. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{\bar{x}}$. Additional control limit lines can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2.

Constructors

XbarS

```
public XbarS(AxisXY axis, double[][] x)
```

Description

Creates an `XbarS` chart from sample data using within sample standard deviations.

Parameters

`axis` – the `AxisXY` parent of this node.

`x` – is an array of arrays containing sample data. The data of the i-th sample is in `x[i]`. Each row must have at least one entry.

XbarS

```
public XbarS(AxisXY axis, int sampleSize, double[] xbar, double[] w)
```

Description

Creates an `XbarS` chart given the means and standard deviations for a series of equally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is the number of observations in each sample. It must be at least one.

`xbar` – is an array containing the mean values for a series of samples.

`w` – is an array containing the within sample variation for a series of samples.

Exception

`IllegalArgumentException` is thrown if the two input arrays do not have the same length.

XbarS

```
public XbarS(AxisXY axis, int[] sampleSize, double[] xbar, double[] w)
```

Description

Creates an `XbarS` chart given the means and standard deviations for a series of unequally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is an array containing the number of observations in each sample. Each sample must have at least one observation.

`xbar` – is an array containing the mean values for a series of samples.

`w` – is an array containing the within sample variation for a series of samples.

Exception

`IllegalArgumentException` is thrown if the three input arrays do not all have the same length.

Methods

capabilityIndexCp

```
public double capabilityIndexCp(double lowerSpecificationLimit, double
upperSpecificationLimit)
```

Description

Returns the capability index c_p

$$c_p = \frac{\text{USL} - \text{LSL}}{6\sigma}$$

where LSL and USL are the lower and upper specification limits and

$$\sigma = \frac{\bar{s}}{c_{4,n}\sqrt{n}}$$

.

Parameters

`lowerSpecificationLimit` – is the lower specification limit.

`upperSpecificationLimit` – is the upper specification limit.

Returns

the capability index.

capabilityIndexCpk

```
public double capabilityIndexCpk(double lowerSpecificationLimit, double
upperSpecificationLimit)
```

Description

Returns the capability index c_{pk}

$$c_{pk} = \min\left[\frac{\text{USL} - \bar{x}}{3\sigma}, \frac{\bar{x} - \text{LSL}}{3\sigma}\right]$$

where LSL and USL are the lower and upper specification limits, $\bar{x}$ is the center line, and

$$\sigma = \frac{\bar{s}}{c_{4,n}\sqrt{n}}$$

.

Parameters

`lowerSpecificationLimit` – is the lower specification limit.

`upperSpecificationLimit` – is the upper specification limit.

Returns

the capability index.

createCharts

```
static public ShewhartControlChart[] createCharts(Chart chart, double[][] x)
```

Description

Creates a combined `XbarS` chart and `SChart` from data. The viewport of the `XbarS` chart is [0.2, 0.9] by [0.1, 0.4]. The viewport of the `SChart` chart is [0.2, 0.9] by [0.5, 0.8].

Parameters

`chart` – is the `Chart` object which is the parent of the two charts being created.

`x` – is an array of arrays containing sample data. The data of the *i*-th sample is in `x[i]`. Each row must have at least one entry.

Returns

an array of length two containing the XBarS chart and the `SChart`.

createCharts

```
static public ShewhartControlChart[] createCharts(Chart chart, int sampleSize,
double[] xbar, double[] w)
```

Description

Creates a combined `XbarS` chart and `SChart` given the means and in sample standard deviations for a series of equally sized samples. The viewport of the `XbarS` chart is [0.2, 0.9] by [0.1, 0.4]. The viewport of the `SChart` chart is [0.2, 0.9] by [0.5, 0.8].

Parameters

`chart` – is the `Chart` object which is the parent of the two charts being created.

`xbar` – is an array containing the mean values for a series of samples.

`sampleSize` – is the number of observations in each sample. It must be at least one.

`w` – is an array containing the in sample standard deviations of the samples.

Returns

an array of length two containing the XBarS chart and the SChart.

createCharts

```
static public ShewhartControlChart[] createCharts(Chart chart, int[]
sampleSize, double[] xbar, double[] w)
```

Description

Creates a combined X-bar chart and S-chart given the means and in sample standard deviations for a series of unequally sized samples.

Parameters

`chart` – is the `Chart` object which is the parent of the two charts being created.

`xbar` – is an array containing the mean values for a series of samples.

`sampleSize` – is an array containing the number of observations in each sample. Each sample must have at least one observation.

`w` – is an array containing the in sample standard deviations of the samples.

Returns

an array of length two containing the XBarS chart and the `SChart`.

getWbar

```
public double getWbar()
```

Description

Returns the value of the "Wbar" attribute, the within sample variation for a series of samples. Its default value is set by the constructor to an estimate of the within sample variation

Returns

the within sample variation for a series of samples.

prePaint

```
public void prePaint()
```

Description

Setup chart with current settings.

setWbar

```
public void setWbar(double wbar)
```

Description

Sets the value of the "Wbar" attribute, the within sample variation for a series of samples. Its default value is set by the constructor to an estimate of the within sample variation.

Parameter

`wbar` – is the within sample variation for a series of samples.

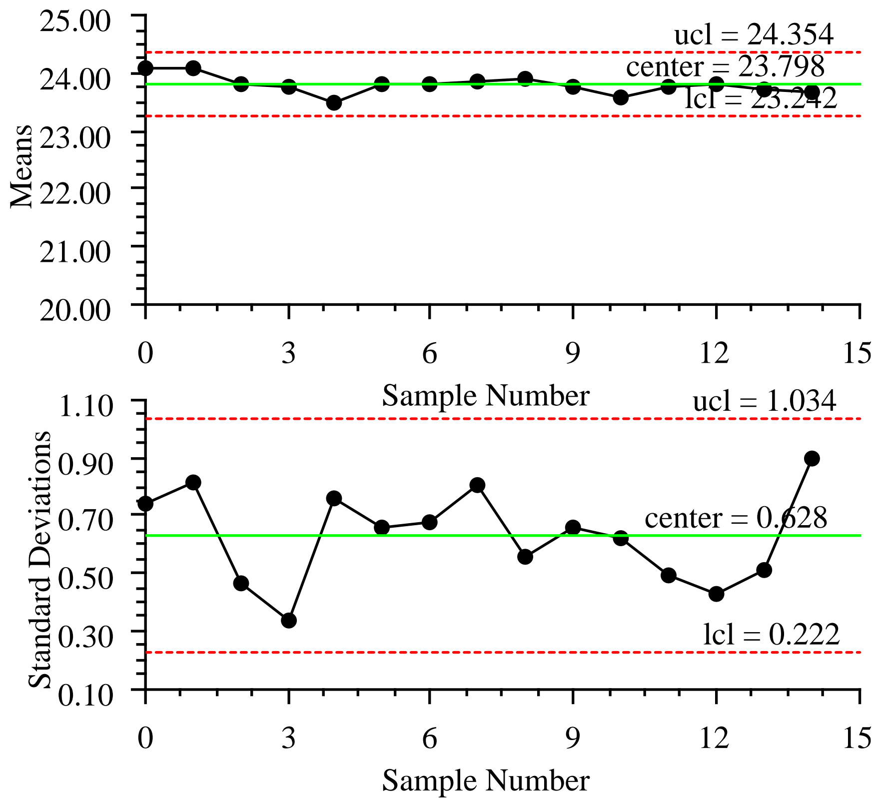
Example: XbarS Chart

During a manufacturing process 15 samples, each containing 12 items, were measured. An XbarS chart was contructed from the 15 sample standard deviations. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class XbarSEx1 extends javax.swing.JApplet {

    static private final double data[][] = {
        {23.97, 24.08, 23.16, 23.49, 24.73, 25.26,
            22.97, 23.12, 24.66, 24.20, 24.62, 24.56},
        {24.20, 24.50, 23.45, 22.22, 25.10, 24.41,
            24.05, 23.75, 23.89, 24.83, 25.21, 23.70},
        {23.73, 22.70, 23.54, 24.37, 24.08, 23.74,
            24.08, 23.95, 24.20, 23.43, 24.26, 23.61},
        {23.46, 23.14, 23.96, 23.37, 23.73, 24.29,
            24.13, 23.62, 24.08, 23.73, 23.91, 23.65},
        {23.95, 24.13, 22.95, 24.72, 24.40, 22.82,
            22.66, 22.71, 24.21, 23.39, 23.41, 22.56},
        {24.13, 24.28, 23.84, 24.55, 23.53, 23.77,
            24.38, 22.58, 24.47, 23.63, 22.64, 24.12},
        {23.69, 24.19, 24.76, 23.29, 24.84, 24.12,
            23.83, 22.60, 24.35, 22.96, 23.81, 23.46},
        {24.35, 23.11, 25.24, 24.10, 24.93, 22.93,
            23.47, 23.55, 23.91, 24.08, 22.45, 24.13},
        {24.98, 24.58, 23.52, 24.42, 23.90, 23.55,
            23.67, 24.25, 23.85, 23.08, 23.44, 23.43},
        {23.90, 24.04, 24.29, 23.62, 23.29, 23.16,
            24.34, 24.37, 24.19, 24.33, 22.17, 23.66},
        {23.51, 24.98, 24.34, 23.87, 23.29, 23.96,
            23.06, 23.47, 23.53, 22.87, 23.38, 22.86},
        {23.13, 23.17, 23.40, 23.68, 23.41, 23.67,
            23.37, 24.40, 24.64, 24.16, 24.17, 23.88},
        {23.52, 24.23, 24.25, 24.31, 23.89, 24.02,
            24.04, 23.83, 22.82, 23.93, 23.55, 23.40},
        {23.66, 23.59, 23.79, 24.07, 23.76, 23.34,
            23.65, 23.73, 25.12, 23.65, 23.23, 23.13},
        {23.04, 23.75, 22.84, 23.46, 21.72, 23.81,
            24.51, 24.01, 24.73, 23.88, 23.34, 24.98}
    };

    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) {
        ShewhartControlChart control[] = XbarS.createCharts(chart, data);
    }
```

```
    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        XbarSEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



SChart class

```
public class com.imsl.chart.qc.SChart extends
com.imsl.chart.qc.ShewhartControlChart
```

`SChart` is an S chart using sample standard deviations to monitor the variability of a process. This is normally used with sample sizes greater than 10. The control limits are at

$$\bar{s} + k\frac{\bar{s}}{c_{4,n}}\sqrt{1 - c_{4,n}^2}$$

where $\bar{s}$ is the mean of the within sample standard deviations, n is the sample size, and k is the value of the "ControlLimit" attribute for the line. Additionally,

$$c_{4,n} = \sqrt{\frac{2}{n-1}}\frac{\Gamma(\frac{n}{2})}{\Gamma(\frac{n-1}{2})}$$

is a factor such that $\bar{s}/c_{4,n}$ is an unbiased estimator of the within sample standard deviation. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{s}$. Additional control limit lines can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2.

Constructors

SChart

`public SChart(AxisXY axis, double[][] x)`

Description

Creates an S chart given sample data.

Parameters

`axis` – the `AxisXY` parent of this node

`x` – is an array of arrays containing sample data. The data of the i-th sample is in `x[i]`. Each row must have at least two entries.

SChart

`public SChart(AxisXY axis, int sampleSize, double[] s)`

Description

Creates an S chart given the within sample standard deviations for a series of equally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is the number of observations in each sample. It must be at least 2.

`s` – is an array containing the within sample standard deviations for a series of samples.

SChart

`public SChart(AxisXY axis, int[] sampleSize, double[] s)`

Description

Creates an S chart given the within sample standard deviations for a series of unequally sized samples.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is an array containing the number of observations in each sample. All samples must have at least two observations.

`s` – is an array containing the within sample standard deviations for a series of samples.

Exception

`IllegalArgumentException` is thrown if the two input arrays do not have the same length.

Method

prePaint

`public void prePaint()`

Description

Setup chart with current settings.

Example: S Chart

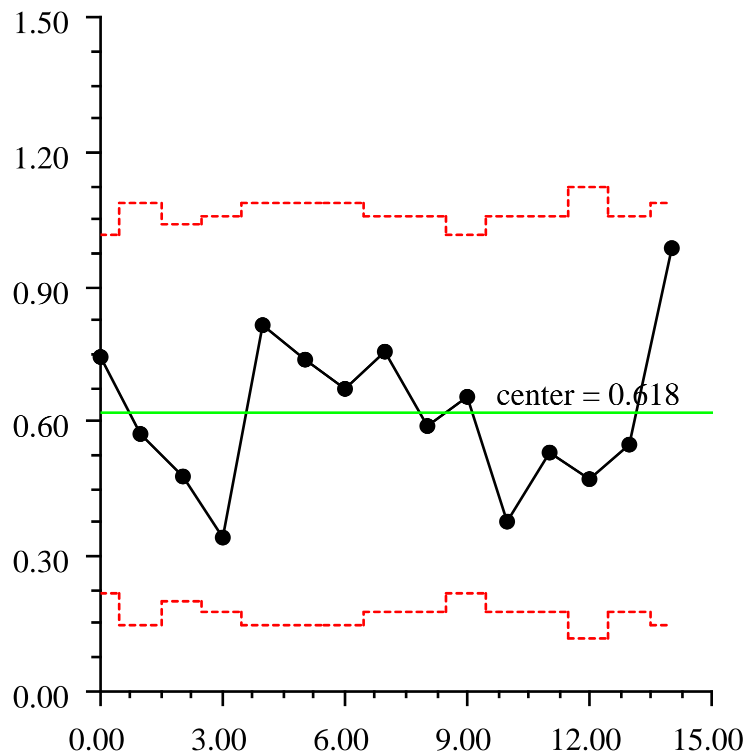
During a manufacturing process 15 samples, each containing up to 12 items, were measured. An S chart was contructed from the 15 sample standard deviations. Since the sample sizes are unequal the upper and lower control limit lines are stair step lines.

This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class SChartEx1 extends javax.swing.JApplet {

    static private final double data[][] = {
        {23.97, 24.08, 23.16, 23.49, 24.73, 25.26, 22.97,
            23.12, 24.66, 24.20, 24.62, 24.56},
        {24.20, 24.50, 23.45, 24.05, 23.75, 23.89, 24.83, 25.21, 23.70},
```

```
        {23.73, 22.70, 23.54, 24.37, 23.74, 24.08, 23.95,
            24.20, 23.43, 24.26, 23.61},
        {23.46, 23.14, 23.73, 24.29, 24.13, 23.62, 24.08,
            23.73, 23.91, 23.65},
        {23.95, 22.95, 24.72, 24.40, 22.82, 22.66, 22.71, 23.41, 22.56},
        {24.13, 24.28, 23.84, 24.55, 22.58, 24.47, 23.63, 22.64, 24.12},
        {23.69, 24.19, 24.76, 23.83, 22.60, 24.35, 22.96, 23.81, 23.46},
        {24.35, 23.11, 25.24, 24.10, 23.47, 23.55, 23.91,
            24.08, 22.45, 24.13},
        {24.98, 24.58, 23.52, 23.55, 23.67, 24.25, 23.85,
            23.08, 23.44, 23.43},
        {23.90, 24.04, 24.29, 23.62, 23.29, 23.16, 24.34,
            24.37, 24.19, 24.33, 22.17, 23.66},
        {23.51, 23.87, 23.29, 23.96, 23.06, 23.47, 23.53,
            22.87, 23.38, 22.86},
        {23.13, 23.17, 23.40, 23.68, 23.37, 24.40, 24.64,
            24.16, 24.17, 23.88},
        {23.52, 24.23, 24.25, 23.83, 22.82, 23.93, 23.55, 23.40},
        {23.66, 23.59, 23.79, 24.07, 23.65, 23.73, 25.12,
            23.65, 23.23, 23.13},
        {23.04, 23.75, 22.84, 23.46, 21.72, 24.73, 23.88, 23.34, 24.98}
    };

    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) {
        AxisXY axis = new AxisXY(chart);
        new SChart(axis, data);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        SChartEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



XmR class

```
public class com.imsl.chart.qc.XmR extends
com.imsl.chart.qc.ShewhartControlChart
```

`XmR` is an *XmR* chart for monitoring a process using moving ranges.

The moving range control chart uses the moving range of two successive observations to measure the process variability. This control chart is used for individual measurements (sample size = 1). The *moving range* is defined to be $\mathrm{MR}_i = |x_i - x_{i-1}|$. The control limits are at

$$\bar{x} + k\frac{\overline{MR}}{d_{2,2}}$$

where $\bar{x}$ is the mean of all of the individual observations, $\overline{MR}$ is the mean of the moving averages, and k is the value of the "ControlLimit" attribute for the line. Additionally, $d_{2,n} = E(R)/\sigma$ where R is the range of a Gaussian distribution. Therefore $\overline{MR}/d_{2,2}$ is an estimate of the standard deviation. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{x}$. Additionally control limits can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2.

Constructor

XmR

```
public XmR(AxisXY axis, double[] x)
```

Description

Creates an `XmR` chart given sample data.

Parameters

`axis` – the `AxisXY` parent of this node

`x` – is an array containing sample data.

Methods

getMRBar

```
public double getMRBar()
```

Description

Returns the expected mean of of all of the moving ranges of two observations.

Returns

the expected mean of the moving ranges.

prePaint

```
public void prePaint()
```

Description

Setup chart with current settings.

setMRBar

`public void setMRBar(double mrBar)`

Description

Sets the expected mean of all of the moving ranges of two observations. This defaults to the mean of the moving ranges computed by the constructor from the data. Its default value is the computed from the data passed to the constructor.

Parameter

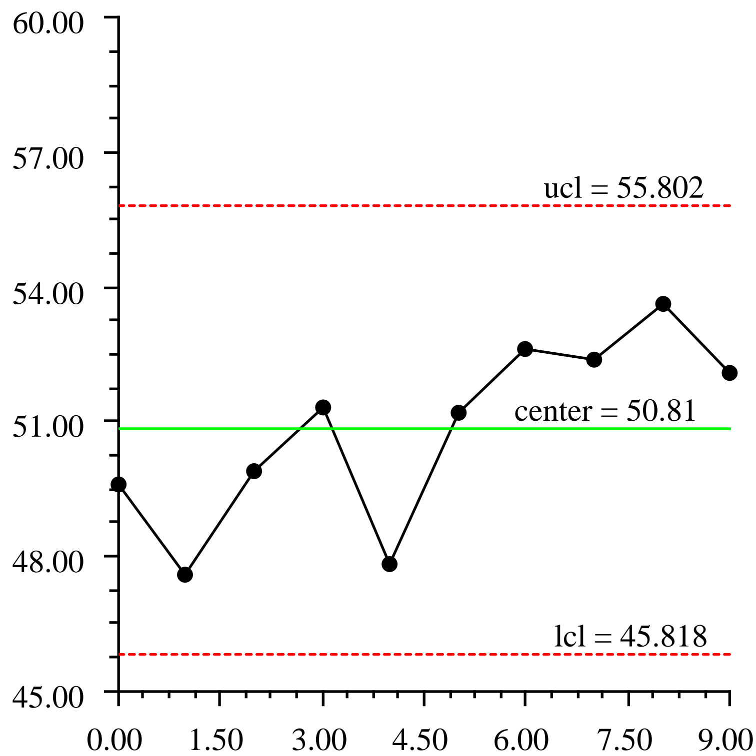
`mrBar` – is the expected mean of the moving ranges.

Example: Moving Range Chart

This moving range chart plots the flowrate for 10 batches. The data is from NIST Engineering Statistics Handbook: Individuals Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.XmR;

public class XmREx1 extends javax.swing.JApplet {

    static private final double flowrate[] = {
        49.6, 47.6, 49.9, 51.3, 47.8, 51.2, 52.6, 52.4, 53.6, 52.1
    };

    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) {
        AxisXY axis = new AxisXY(chart);
        XmR mr = new XmR(axis, flowrate);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        XmREx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



NpChart class

```
public class com.imsl.chart.qc.NpChart extends
com.imsl.chart.qc.ShewhartControlChart
```

`NpChart` is an *np*-chart for monitoring the number of defects when defects are not rare.

Control limits are computed using the binomial distribution. If defects are rare `com.imsl.chart.qc.CChart` (p. 1809) should be used instead. The control limits are at

$$np + k\sqrt{np(1-p)}$$

where p is the is the proportion of defective items, n is the sample size, and k is the value of the "ControlLimit" attribute for the line. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{c}$. If $p = \bar{c}/n$ is the proportion defective then the equation can be written as

$$np + k\sqrt{np(1-p)}$$

Additional control limits can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2.The lower control limit is forced to have a minimum value of zero. The y-axis labels are formatted as integers.

Constructors

NpChart

```
public NpChart(AxisXY axis, int sampleSize, int[] numberDefective)
```

Description

Creates an np-Chart given the number of defects in a series of samples.

Parameters

`axis` – the `AxisXY` parent of this node.

`sampleSize` – is the number of observations in each sample. It must be at least one.

`numberDefective` – is an array containing the number of defects in each of a series of samples. All of its entries must be nonnegative.

NpChart

```
public NpChart(AxisXY axis, int[] sampleSize, int[] numberDefective)
```

Description

Creates a np-Chart given the number of defects in a series of samples, where the number of observations per sample is not constant.

Parameters

`axis` – the `AxisXY` parent of this node.

`sampleSize` – is an array containg the number of observations in each sample. Each sample must contain at least one observation.

`numberDefective` – is an array containing the number of defects in each of a series of samples. All of its entries must be nonnegative.

Method

prePaint

`public void prePaint()`

Description

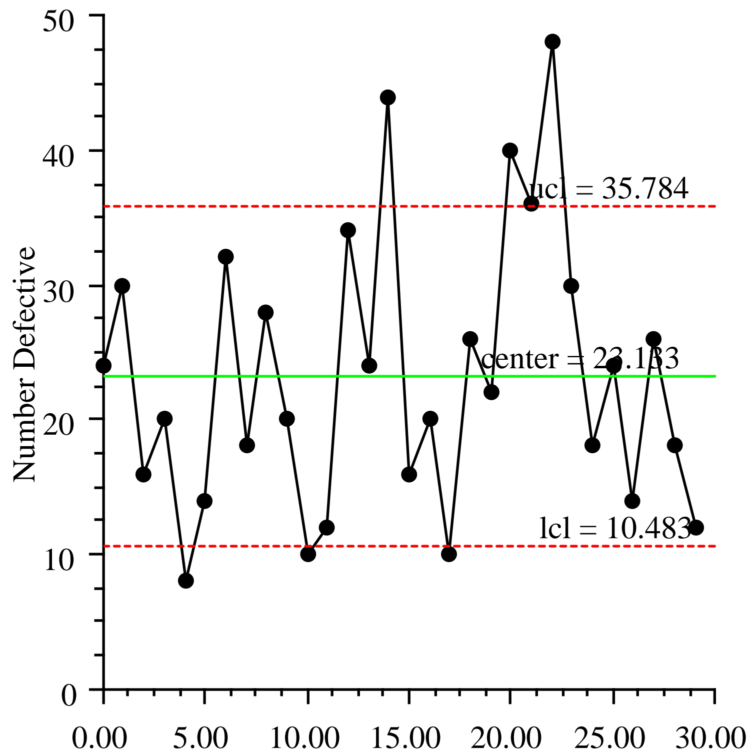
Setup chart with current settings.

Example: np-Chart

The location of 50 chips on each of 30 successive wafers is measured. The number of defects is the number of horizontal or vertical misregistrations of chips. The data is from NIST Engineering Statistics Handbook: Proportions Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.NpChart;

public class NpChartEx1 extends javax.swing.JApplet {

    static private final int numberDefective[] = {
        24, 30, 16, 20, 8, 14, 32, 18, 28, 20, 10, 12, 34,
        24, 44, 16, 20, 10, 26, 22, 40, 36, 48, 30, 18, 24,
        14, 26, 18, 12
    };
    static private final int sampleSize = 100;

    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) {
        AxisXY axis = new AxisXY(chart);
        axis.getAxisY().getAxisTitle().setTitle("Number Defective");
        NpChart np = new NpChart(axis, sampleSize, numberDefective);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        NpChartEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



PChart class

```
public class com.imsl.chart.qc.PChart extends
com.imsl.chart.qc.ShewhartControlChart
```

`PChart` is a *p*-chart for monitoring the defect rate when defects are not rare.

The defect rate is the number of defects found divided by the number of samples inspected. The number of defects are not assumed to be rare. Control limits are computed using the binomial distribution. If defects are rare, use `com.imsl.chart.qc.UChart` (p. 1812) instead. The control limits are at

$$\bar{p} + k\sqrt{\frac{\bar{p}(1-\bar{p})}{n}}$$

where $\bar{p}$ is the mean defect rate, n is the sample size, and k is the value of the "ControlLimit" attribute for the line. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{p}$. Additional control limits can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2. The true fraction conforming p can be used by setting the attribute "Center" to p. The lower control limit is forced to have a minimum value of zero. The upper control limit is forced to have a maximum value of one.

Constructors

PChart

`public PChart(AxisXY axis, int sampleSize, double[] defectRate)`

Description

Creates a p-Chart given the defect rates for a series of samples with equal sample sizes.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is the number of observations in each sample. It must be at least one.

`defectRate` – is an array containing defect rates of the samples. The defect rates must all be in the range [0,1].

PChart

`public PChart(AxisXY axis, int[] sampleSize, double[] defectRate)`

Description

Creates a p-Chart given the defect rates for a series of samples with varying sample sizes.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is an array containing the number of observations in each sample. It must be at least one.

`defectRate` – is an array containing defect rates of the samples. The defect rates must all be in the range [0,1]. The lengths of the arrays `sampleSize` and `defectRate` must be equal.

PChart

`public PChart(AxisXY axis, int sampleSize, int[] numberDefects)`

Description

Creates a p-Chart given the number of defects for a series of samples with equal sample sizes.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is the number of observations in each sample. Each sample must contain at least one observation.

`numberDefects` – is an array containing the number of defects in each of a series of samples. The number of defects should not be less than zero.

PChart

`public PChart(AxisXY axis, int[] sampleSize, int[] numberDefects)`

Description

Creates a p-Chart given the number of defects for a series of samples with varying sample sizes.

Parameters

`axis` – the `AxisXY` parent of this node

`sampleSize` – is an array containing the number of observations in each sample. It must be at least one.

`numberDefects` – is an array of arrays containing the number of defects in each of a series of samples. The number of defects should not be less than zero.

Method

prePaint

`public void prePaint()`

Description

Setup chart with current settings.

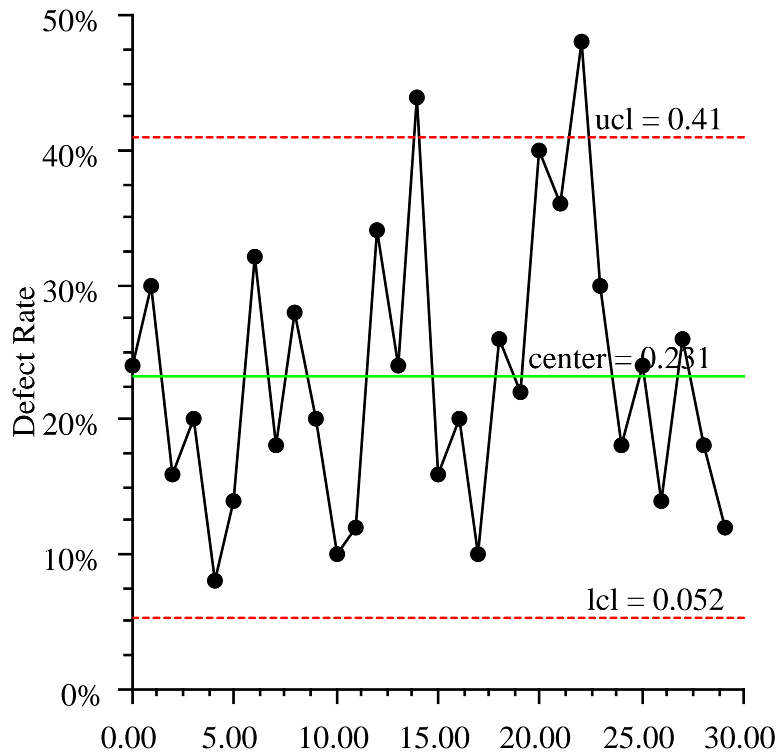
Example: p-Chart

The location of 50 chips on each of 30 successive wafers is measured. The number of defects is the number of horizontal or vertical misregistrations of chips. The data is from NIST Engineering Statistics Handbook: Proportions Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.PChart;

public class PChartEx1 extends javax.swing.JApplet {

    static private final double defectRate[] = {
        0.24, 0.3, 0.16, 0.2, 0.08, 0.14, 0.32, 0.18, 0.28, 0.2,
        0.1, 0.12, 0.34, 0.24, 0.44, 0.16, 0.2, 0.1, 0.26, 0.22,
```

```
        0.4, 0.36, 0.48, 0.3, 0.18, 0.24, 0.14, 0.26, 0.18, 0.12
    };
    static private final int sampleSize = 50;

    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) {
        AxisXY axis = new AxisXY(chart);
        axis.getAxisY().getAxisTitle().setTitle("Defect Rate");
        axis.getAxisY().getAxisLabel().setTextFormat("percent");
        PChart pc = new PChart(axis, sampleSize, defectRate);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        PChartEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



CChart class

```
public class com.imsl.chart.qc.CChart extends
com.imsl.chart.qc.ShewhartControlChart
```

`CChart` is a *c*-chart for monitoring the count of the number of defects when defects are rare. Control limits are computed using the Poisson distribution. If defects are not rare

`com.imsl.chart.qc.NpChart` (p. 1802) should be used instead. The control limits are at

$$\bar{c} + k\sqrt{\bar{c}}$$

where $\bar{c}$ is the mean number of defects per sample and k is the value of the “ControlLimit” attribute for the line. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{c}$. Additional control limits can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2. The lower control limit is computed using the Poisson distribution. First the probability of the number of defects being less than k (the “ControlLimit” value) standard deviations from the mean in a normal distribution is computed. The number of defects required to produce the same probability assuming a Poisson distribution, with the same mean, is then computed. The lower limit is set to be one more than this number of defects.

Constructor

CChart

`public CChart(AxisXY axis, int[] numberDefects)`

Description

Creates a C-chart given the number of defects in a series of samples.

Parameters

`axis` – the `AxisXY` parent of this node

`numberDefects` – is the number of defects in a series of samples. The number of defects should not be less than zero. There should be the same number of items in each sample.

Method

prePaint

`public void prePaint()`

Description

Setup chart with current settings.

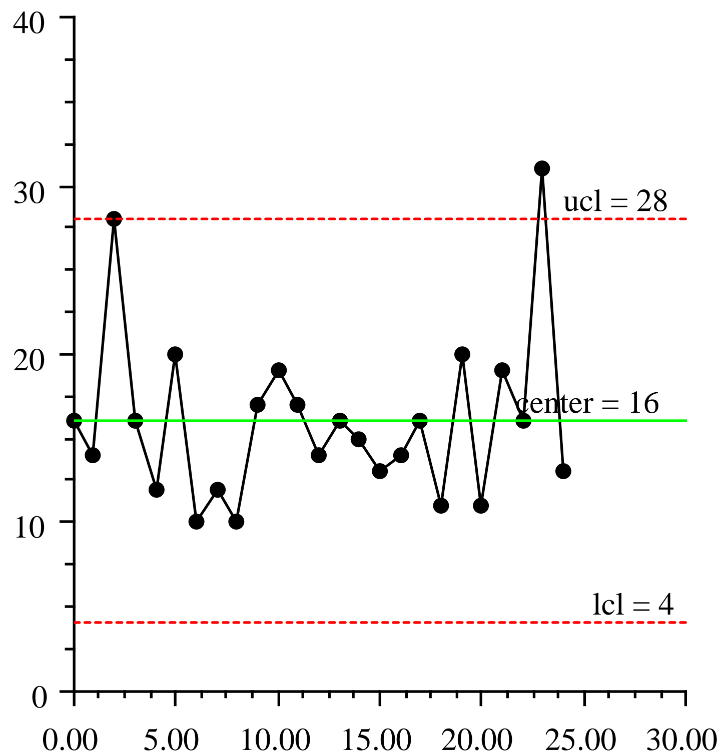
Example: c-Chart

The number of defects on each of 25 successive wafers is plotted. The data is from NIST Engineering Statistics Handbook: Counts Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.CChart;
```

```
public class CChartEx1 extends javax.swing.JApplet {

    static private final int numberOfDefects[] = {
        16, 14, 28, 16, 12, 20, 10, 12, 10, 17, 19, 17, 14, 16, 15, 13,
        14, 16, 11, 20, 11, 19, 16, 31, 13
    };

    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) {
        AxisXY axis = new AxisXY(chart);
        CChart control = new CChart(axis, numberOfDefects);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        CChartEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



UChart class

```
public class com.imsl.chart.qc.UChart extends
com.imsl.chart.qc.ShewhartControlChart
```

`UChart` is a u-chart for monitoring the defect rate when defects are rare.

The defect rate is the number of defects found divided by the number of samples inspected. The number of defects are assumed to be rare. Control limits are computed using the Poisson distribution. If defects are not rare, use `com.imsl.chart.qc.PChart` (p. 1805) instead. The sample sizes are not required to be equal. The control limits are at

$$\bar{p} + k\sqrt{\bar{p}/n}$$

where $\bar{p}$ is the mean defect rate, n is the sample size, and k is the value of the "ControlLimit" attribute for the line. By default, the chart contains an upper control limit line with k=3, a lower control limit line with k=-3, and a central line equal to $\bar{p}$. Additional control limits can be added. The method `addWeco` adds control limits with k = -2, -1, 1, 2.

Constructors

UChart

```
public UChart(AxisXY axis, double sizeSample, int[] numberDefects)
```

Description

Creates a u-Chart given the number of defects for a series of samples with equal sample sizes.

Parameters

`axis` – the `AxisXY` parent of this node

`sizeSample` – is the size of each sample.

`numberDefects` – is an array of arrays containing the number of defects. The number of defects must all be nonnegative.

UChart

```
public UChart(AxisXY axis, double[] sizeSample, int[] numberDefects)
```

Description

Creates a u-Chart given the number of defects rates for a series of samples with varying sample sizes.

Parameters

`axis` – the `AxisXY` parent of this node

`sizeSample` – is an array containing the size of each sample.

`numberDefects` – is an array of arrays containing the number of defects. The number of defects must all be nonnegative. The length of the `sizeSample` and `numberDefects` arrays must be equal.

Method

prePaint

```
public void prePaint()
```

Description

Setup chart with current settings.

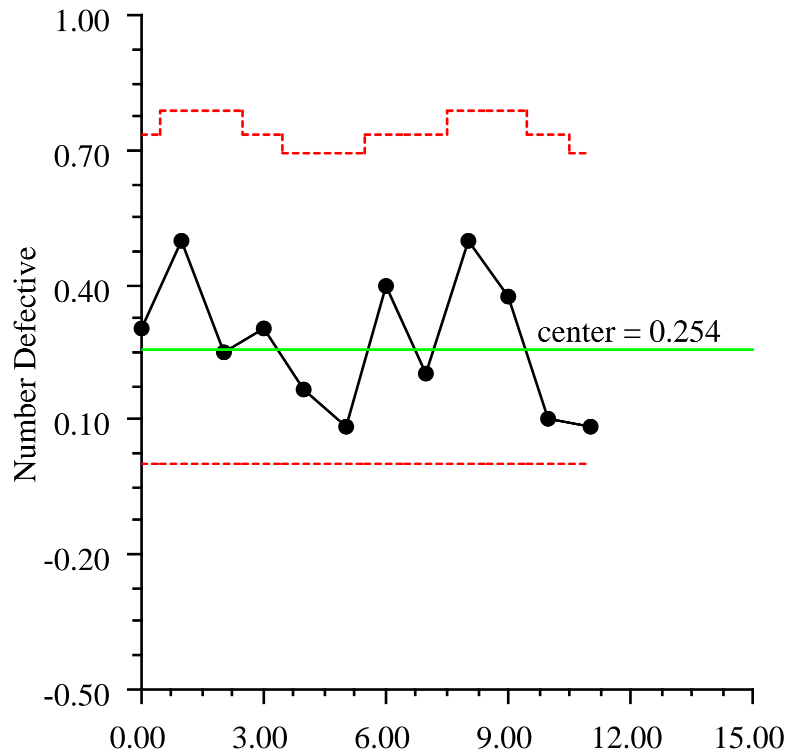
Example: u-Chart

The number of defects in each of 12 samples was counted. The number of items in the samples varied from 8 to 12.

This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.UChart;

public class UChartEx1 extends javax.swing.JApplet {

    static private final int numberDefects[] = {
        3, 4, 2, 3, 2, 1, 4, 2, 4, 3, 1, 1
    };
    static private final double sizeSample[] = {
        10, 8, 8, 10, 12, 12, 10, 10, 8, 8, 10, 12
    };

    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) {
        AxisXY axis = new AxisXY(chart);
        axis.getAxisY().getAxisTitle().setTitle("Number Defective");
        UChart pcc = new UChart(axis, sizeSample, numberDefects);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        UChartEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



EWMA class

```
public class com.imsl.chart.qc.EWMA extends
com.imsl.chart.qc.ShewhartControlChart
```

`EWMA` is an *exponentially weighted moving average* control chart.

The EWMA statistic is given by

$$\text{EWMA}_t = \lambda x_t + (1-\lambda)\text{EWMA}_{t-1}$$

where x_t is the observation at time t and $0 < \lambda \le 1$ is the decay parameter which determines the depth of memory of EWMA.

Constructors

EWMA

```
public EWMA(AxisXY axis, double[] x, double lambda)
```

Description

Creates an exponentially weighted moving average chart. The expected mean and standard deviation are computed from the sample data.

Parameters

`axis` – the `AxisXY` parent of this node

`x` – is an array containing the data.

`lambda` – is the decay parameter. It is usually between 0.2 and 0.3.

Exception

`IllegalArgumentException` is thrown if $0 < \lambda \le 1$ does not hold.

EWMA

```
public EWMA(AxisXY axis, double[] data, double lambda, double expectedMean,
double expectedStandardDeviation)
```

Description

Creates an exponentially weighted moving average chart using the given values for the expected mean and standard deviation.

Parameters

`axis` – the `AxisXY` parent of this node

`data` – is an array containing data.

`lambda` – is the decay parameter. It is usually between 0.2 and 0.3.

`expectedMean` – is the expected mean of the data.

`expectedStandardDeviation` – is the expected standard deviation of the data.

Exception

`IllegalArgumentException` is thrown if $0 < \lambda \le 1$ does not hold.

Methods

getLambda

`public double getLambda()`

Description

Returns the value of the attribute "Lambda". This is the decay parameter.

Returns

the value of the attribute "Lambda". Its default value is 0.25.

prePaint

`public void prePaint()`

Description

Setup chart with current settings.

setLambda

`public void setLambda(double lambda)`

Description

Sets the value of the attribute "Lambda". This is the decay parameter. Its default value is 0.25.

Parameter

`lambda` – is the value of the attribute "Lambda".

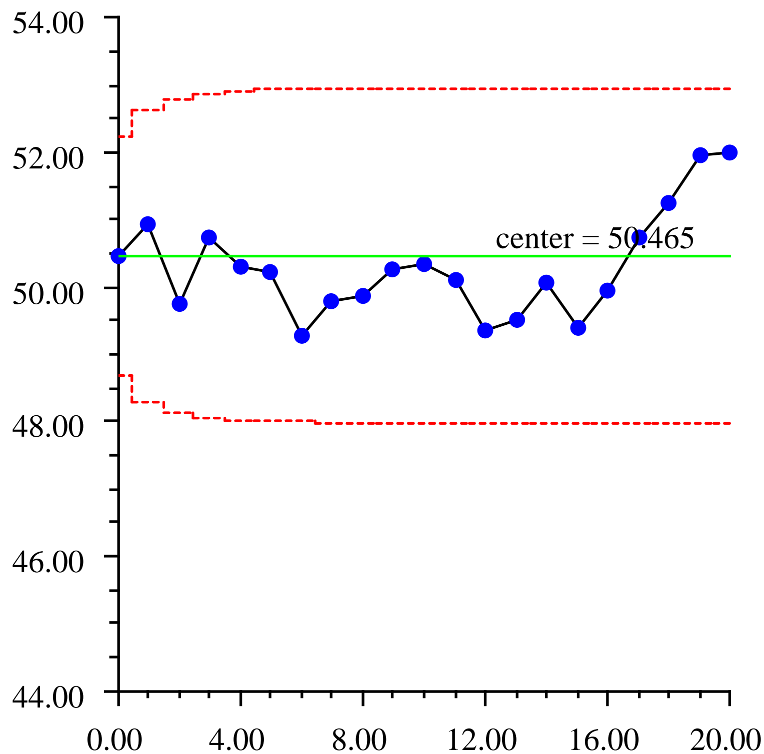
Example: EWMA Chart

A process is monitored using the EWMA control chart. The data is from NIST Engineering Statistics Handbook: EWMA Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.*;

public class EWMAEx1 extends javax.swing.JApplet {

    static private final double data[] = {
        52.0, 47.0, 53.0, 49.3, 50.1, 47.0,
        51.0, 50.1, 51.2, 50.5, 49.6, 47.6,
        49.9, 51.3, 47.8, 51.2, 52.6, 52.4,
        53.6, 52.1
    };

    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) {
        AxisXY axis = new AxisXY(chart);
        double lambda = 0.3;
        EWMA ewma = new EWMA(axis, data, lambda);
        ewma.getControlData().setMarkerColor(java.awt.Color.blue);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        EWMAEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



CuSum class

```
public class com.imsl.chart.qc.CuSum extends
com.imsl.chart.qc.ShewhartControlChart
```

`CuSum` is a *cumulative sum* chart. It is more efficient than a Shewhart chart for detecting small shifts in the mean of a process. CuSum plots the cumulative sum of the deviations of the expected value. If μ_0 is

the expected mean for a process and $\bar{x}_i$ are the sample means then the cumulative sum is

$$C_i = C_{i-1} + (\bar{x}_i - \mu_0)$$

Constructor

CuSum

```
public CuSum(AxisXY axis, double[] xbar)
```

Description

Creates a CuSum chart given the means of a series of samples.

Parameters

`axis` – the `AxisXY` parent of this node

`xbar` – are the means of the samples.

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

getExpectedMean

```
public double getExpectedMean()
```

Description

Returns the expected mean of all of the data from all of the samples. This is either the mean of the `xbar` data as computed by the constructor or the value set using `setExpectedMean`.

Returns

the expected mean of all of the data.

prePaint

```
public void prePaint()
```

setExpectedMean

```
public void setExpectedMean(double expectedMean)
```

Description

Sets the expected mean of all of the data from all of the samples. The value of the `ExpectedMean` attribute is either the expected mean value or the target mean. This defaults to the mean of the `xbar` data given to the constructor.

Parameter

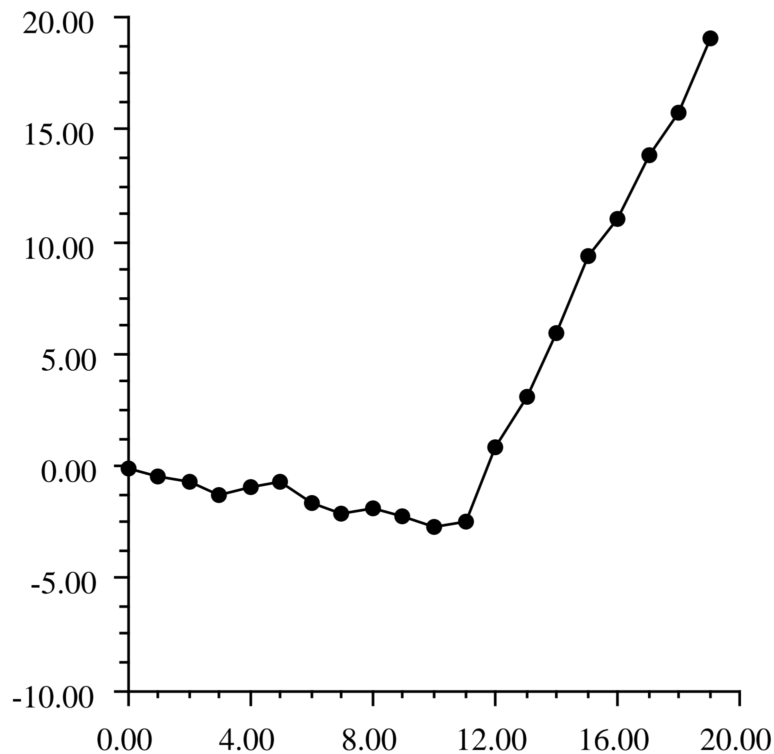
`expectedMean` – is the expected mean of all of the data.

Example: CuSum Chart

A CuSum chart is constructed from the data at NIST Engineering Statistics Handbook: CuSum Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.CuSum;

public class CuSumEx1 extends javax.swing.JApplet {

    static private final double data[] = {
        324.925, 324.675, 324.725, 324.350, 325.350, 325.225, 324.125,
        324.525, 325.225, 324.600, 324.625, 325.150, 328.325, 327.250,
        327.825, 328.500, 326.675, 327.775, 326.875, 328.350
    };

    public void init() {
        Chart chart = new Chart(this);
        JPanelChart panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    private void setup(final Chart chart) {
        AxisXY axis = new AxisXY(chart);
        double mean = 325;
        CuSum cusum = new CuSum(axis, data);
        cusum.setExpectedMean(mean);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        new CuSumEx1().setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



CuSumStatus class

```
public class com.imsl.chart.qc.CuSumStatus extends
com.imsl.chart.qc.ShewhartControlChart
```

`CuSumStatus` is a *cumulative sum status* chart. CuSumStatus plots the tabular CuSum results.

The one-sided upper and lower cusums, C^+ and C^- are computed. These are plotted as two bar charts. The C^+ are plotted as a bar chart above the x-axis and C^- is below the axis. By default, in-control bars are green and out-of-control bars are red.

Control limit lines are drawn at $\pm h$, where h is the value of the `getAbsoluteH`. This can be set either to an absolute number, using `setAbsoluteH`, or relative to the standard deviation of the data, using `setRelativeH`.

Constructor

CuSumStatus

```
public CuSumStatus(AxisXY axis, double[] data, double expectedMean, double
slackValue)
```

Description

Creates a CuSumStatus chart.

Parameters

`axis` – the `AxisXY` parent of this node

`data` – is an array of measurements of a continuous variable.

`expectedMean` – is the expected (or target) mean.

`slackValue` – is the slack value. This is also called K, or the reference value or the allowance.

Methods

addDataMarkers

```
public Data addDataMarkers()
```

Description

Adds the original data to the chart on a newly created axis. The new axis is stored as attribute "DataMarkersAxis" and the new Data node is stored as attribute "DataMarkers".

Returns

the `Data` object containing the markers.

addDataMarkers

```
public Data addDataMarkers(AxisXY axisDataMarkers)
```

Description

Adds the original data to the chart. The axis is stored as attribute "DataMarkersAxis" and the new Data node is stored as attribute "DataMarkers".

Parameter

`axisDataMarkers` – is the axis for the data markers.

Returns

the `Data` object containing the markers.

createDataAxis

```
public AxisXY createDataAxis()
```

Description

Creates a new axis to hold a cummulative line. The created axis is drawn on the right side. Its x-axis is not drawn, since it would overlap with the primary chart axis. The x-axis is set to have the same window size as the orignal axis and its autoscaling attribute is turned off.

Returns

the newly created axis.

getAbsoluteH

```
public double getAbsoluteH()
```

Description

Returns the value for H used for setting limits. There is no attribute "AbsoluteH". This method computes its value using the value of the attribute "RelativeH".

Returns

the value for H.

getBarMinus

```
public Bar getBarMinus()
```

Description

Returns the value of the attribute "BarMinus" containing the C^{-} bars. To access the C^{-} bars for where the process is in control, apply the method `getBarSet(0,0)` to the `Bar` object returned by this method. For the bars for where the process is not in control, apply the method `getBarSet(1,0)` to the `Bar` object returned by this method.

Returns

the Bar object for the negative CuSum bars.

getBarPlus

```
public Bar getBarPlus()
```

Description

Returns the value of the attribute "BarPlus" containing the C^{+} bars. To access the C^{+} bars for where the process is in control, apply the method `getBarSet(0,0)` to the `Bar` object returned by this method. For the bars for where the process is not in control, apply the method `getBarSet(1,0)` to the `Bar` object returned by this method.

Returns

the Bar object for the positive CuSum bars.

getCMinus

```
public double[] getCMinus()
```

Description

Returns the C^- values.

Returns

an array containing the C^- values.

getCPlus

```
public double[] getCPlus()
```

Description

Returns the C^+ values.

Returns

an array containing the C^+ values.

getDataMarkers

```
public Data getDataMarkers()
```

Description

Returns the “DataMarkers” attribute containing the data markers.

Returns

the data markers `Data` object.

getDataMarkersAxis

```
public AxisXY getDataMarkersAxis()
```

Description

Returns the “DataMarkersAxis” attribute containing the axis associated with the data markers. This is normally the right-side axis.

Returns

`AxisXY` associated with the data markers.

getExpectedMean

```
public double getExpectedMean()
```

Description

Returns the expected (target) mean value.

Returns

the expected (target) mean value.

getInitialCMinus

```
public double getInitialCMinus()
```

Description

Returns the initial value of C^-. This is used to initiate a fast initial response (headstart). Typically, this would be set to $H/2$ for a 50% headstart. Its default value is zero.

Returns

initialCMinus is the initial value for C^-.

getInitialCPlus

```
public double getInitialCPlus()
```

Description

Returns the initial value of C^+. This is used to initiate a fast initial response (headstart). Typically, this would be set to $H/2$ for a 50% headstart. Its default value is zero.

Returns

initialCPlus is the initial value for C^+.

getNMinus

```
public int[] getNMinus()
```

Description

Returns N^-, the number of consecutive periods that the cusums C_i^- have been nonzero.

Returns

an array containing the N^- values.

getNPlus

```
public int[] getNPlus()
```

Description

Returns N^+, the number of consecutive periods that the cusums C_i^+ have been nonzero.

Returns

an array containing the N^+ values.

getRelativeH

```
public double getRelativeH()
```

Description

Returns the value for relative h. This is the value of the attribute “RelativeH”. Its default value is 3.0.

Returns

the relative h value.

getSigma

```
public double getSigma()
```

Description

Returns the standard deviation of the data.

Returns

a `double` containing the standard deviation.

getSlackValue

```
public double getSlackValue()
```

Description

Returns the slack value. This is also called K, or the reference value or the allowance.

Returns

the slack value.

prePaint

```
public void prePaint()
```

Description

Setup chart with current settings.

print

```
public void print()
```

Description

Prints the tabular CuSum results. This prints the input data, C^+, N^+, C^-, and N^-.

setAbsoluteH

```
public void setAbsoluteH(double H)
```

Description

Sets the value for h used for setting limits. The relative h and absolute H are related by $H = h\sigma$. There is no attribute "AbsoluteH". This method sets the attribute "RelativeH".

Parameter

`H` – the value for absolute H.

setExpectedMean

```
public void setExpectedMean(double expectedMean)
```

Description

Sets the expected mean of all of the data from all of the samples. The value of the `ExpectedMean` attribute is either the expected mean value or the target mean. This defaults to the mean of the `xbar` data given to the constructor.

Parameter

`expectedMean` – is the expected mean of all of the data.

setInitialCMinus

```
public void setInitialCMinus(double initialCMinus)
```

Description

Sets the initial value of C^{-}. This is used to to initiate a fast initial response (headstart). Typically, this would be set to $H/2$ for a 50% headstart.

Parameter

`initialCMinus` – is the initial value for C^{-}.

setInitialCPlus

```
public void setInitialCPlus(double initialCPlus)
```

Description

Sets the initial value of C^{+}. This is used to initiate a fast initial response (headstart). Typically, this would be set to $H/2$ for a 50% headstart.

Parameter

`initialCPlus` – is the initial value for C^{+}.

setRelativeH

```
public void setRelativeH(double h)
```

Description

Sets the value for relative h. This sets the value of the attribute “RelativeH”.

Parameter

`h` – the value for h.

setSigma

```
public void setSigma(double sigma)
```

Description

Sets the standard deviation of the data. The default is the sample standard deviation of the data used to construct the chart.

Parameter

`sigma` – is the value to be used for the standard deviation.

setX

```
public void setX(double[] x)
```

Description

Sets the x-coordinates of the bars. This also sets the “BarWidth” attribute to (x[1]-x[0])/2.

Parameter

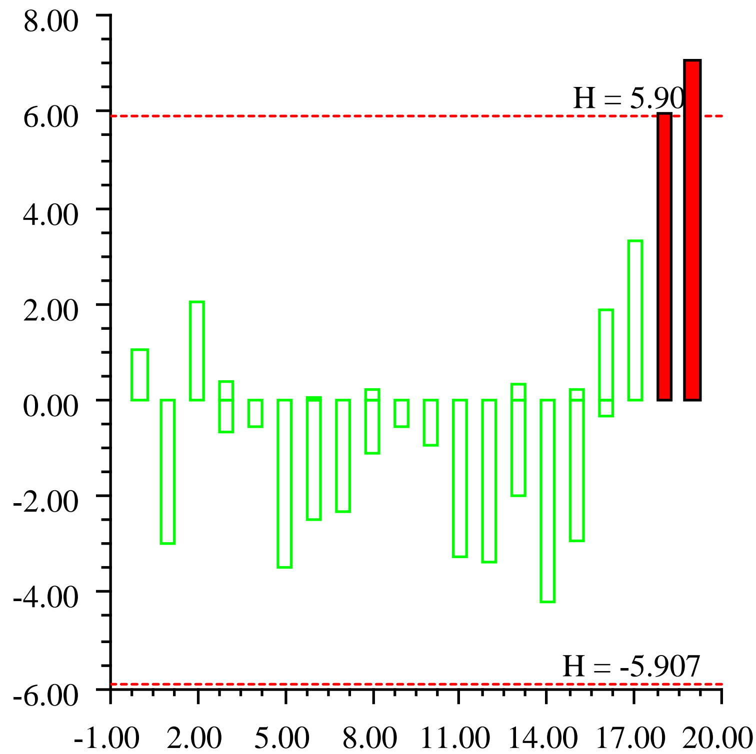
x – is an array containing the x-coordinates. Its length must equal the length of the data array used to construct this object.

Example: CuSumStatus Chart

A process is monitored using the CuSumStatus control chart. The solid red bars indicate the process is out-of-control. The hollow green bars indicate that the process is in control. The data is from NIST Engineering Statistics Handbook: CuSumStatus Control Charts. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.stat.Summary;
import com.imsl.chart.qc.CuSumStatus;

public class CuSumStatusEx1 extends javax.swing.JApplet {

    static private final double data[] = {
        52.0, 47.0, 53.0, 49.3, 50.1, 47.0,
        51.0, 50.1, 51.2, 50.5, 49.6, 47.6,
        49.9, 51.3, 47.8, 51.2, 52.6, 52.4,
        53.6, 52.1
    };

    public void init() {
        Chart chart = new Chart(this);
        JPanelChart panel = new JPanelChart(chart);
        getContentPane().add(panel, java.awt.BorderLayout.CENTER);
        setup(chart);
    }

    private void setup(final Chart chart) {
        AxisXY axis = new AxisXY(chart);
        double mean = Summary.mean(data);
        double slackValue = 0.5;
        CuSumStatus cusum = new CuSumStatus(axis, data, mean, slackValue);
        cusum.getBarPlus().getBarSet(0, 0).setFillType(cusum.FILL_TYPE_NONE);
        cusum.getBarMinus().getBarSet(0, 0).setFillType(cusum.FILL_TYPE_NONE);
        cusum.getBarPlus().getBarSet(0, 0).setFillOutlineColor(
                java.awt.Color.green);
        cusum.getBarMinus().getBarSet(0, 0).setFillOutlineColor(
                java.awt.Color.green);
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        new CuSumStatusEx1().setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



ParetoChart class

`public class com.imsl.chart.qc.ParetoChart extends com.imsl.chart.Bar`

`ParetoChart` is a *Pareto* bar chart. The bars are sorted into descending order. It is used in quality assurance tracking to identify and prioritize areas of greatest impact. It extends `Bar`.

The method `addCumulativeLine` adds a cumulative percentage line to the chart. This is the percent of defects accounted for by the current item and items to its left. If the cumulative percentage line is added, a second axis is created on the right. This is required because the units for this line are 0% to 100%. The units of the original axis (on the left) are the number of defects.

Constructors

ParetoChart

`public ParetoChart(AxisXY axisBar, String[] labels, int[] numberDefects)`

Description

Constructs a Pareto chart.

Parameters

`axisBar` – the `AxisXY` parent of this node. Its formatting is changed to integer formatting.

`labels` – a `String` array which contains the labels for the data values.

`numberDefects` – an `int` array which contains the number of defects. These data values must be in the same order as the values in `labels`, but they do not need to be sorted.

Exception

`IllegalArgumentException` is thrown if the length of the `labels` and `numberDefects` arrays are unequal or if any element of `numberDefects` is negative.

ParetoChart

`public ParetoChart(AxisXY axisBar, String[] labels, int[] numberDefects, double maximumFractionCategoriesPlotted, String otherLabel)`

Description

Constructs a Pareto chart showing only the most important bars. The numberDefects-axis formatting is changed to integer formatting.

Parameters

`axisBar` – the `AxisXY` parent of this node. Its formatting is changed to integer formatting.

`labels` – a `String` array which contains the labels for the data values.

`numberDefects` – an `int` array which contains the number of defects. These data values must be in the same order as the values in `labels`, but they do not need to be sorted.

`maximumFractionCategoriesPlotted` – is maximum cumulative fraction to be represented in separate categories. The remaining categories are consolidated into a single bar. A typical value for this argument is 0.80. This must be at least 0 and no more than 1.

`otherLabel` – is the label of the bar holding total defect count of the categories not plotted.

Exception

IllegalArgumentException is thrown if the length of the labels and numberDefects arrays are unequal or if any element of numberDefects is negative.

ParetoChart

```
public ParetoChart(AxisXY axisBar, String[] labels, int[] numberDefects, int
maximumCategoriesPlotted, String otherLabel)
```

Description

Constructs a Pareto chart showing only a limited number of bars. The numberDefects-axis formatting is changed to integer formatting.

Parameters

axisBar – the AxisXY parent of this node

labels – a String array which contains the labels for the data values.

numberDefects – an int array which contains the number of defects. These data values must be in the same order as the values in labels, but they do not need to be sorted.

maximumCategoriesPlotted – is the maximum number of categories to be plotted. Categories with smaller number of defects are consolidated into a single bar. The total number of bars will be maximumCategoriesPlotted+1. This must be at least 0 and no more than the length of numberDefects.

otherLabel – is the label of the bar holding total defect count of the categories not plotted.

Exception

IllegalArgumentException is thrown if the length of the labels and numberDefects arrays are unequal or if any element of numberDefects is negative.

Methods

addCumulativeLine

```
public Data addCumulativeLine()
```

Description

Creates a new right-side axis and adds a cumulative line to it.

Returns

a Data object containing the cumulative percentage line.

addCumulativeLine

```
public Data addCumulativeLine(AxisXY axisCumulativeLine)
```

Description

Adds a cumulative line to the specified axis.

Parameter

`axisCumulativeLine` – is the axis for the cumulative line.

createCumulativeLineAxis

```
public AxisXY createCumulativeLineAxis()
```

Description

Creates a new axis to hold a cumulative line. The created axis is drawn on the right side. Its x-axis is not drawn, since it would overlap with the primary Pareto chart axis. Its y-axis is labeled using the "percent" format. Its y-axis is scaled to [0,1], it is not autoscaled.

Returns

the newly created axis

getCumulativeAxis

```
public AxisXY getCumulativeAxis()
```

Description

Returns the "CumulativeAxis" attribute.

Returns

`AxisXY` object containing the cumulative line.

getCumulativeLine

```
public Data getCumulativeLine()
```

Description

Returns the "CumulativeLine" attribute.

Returns

the cumulative line `Data` object.

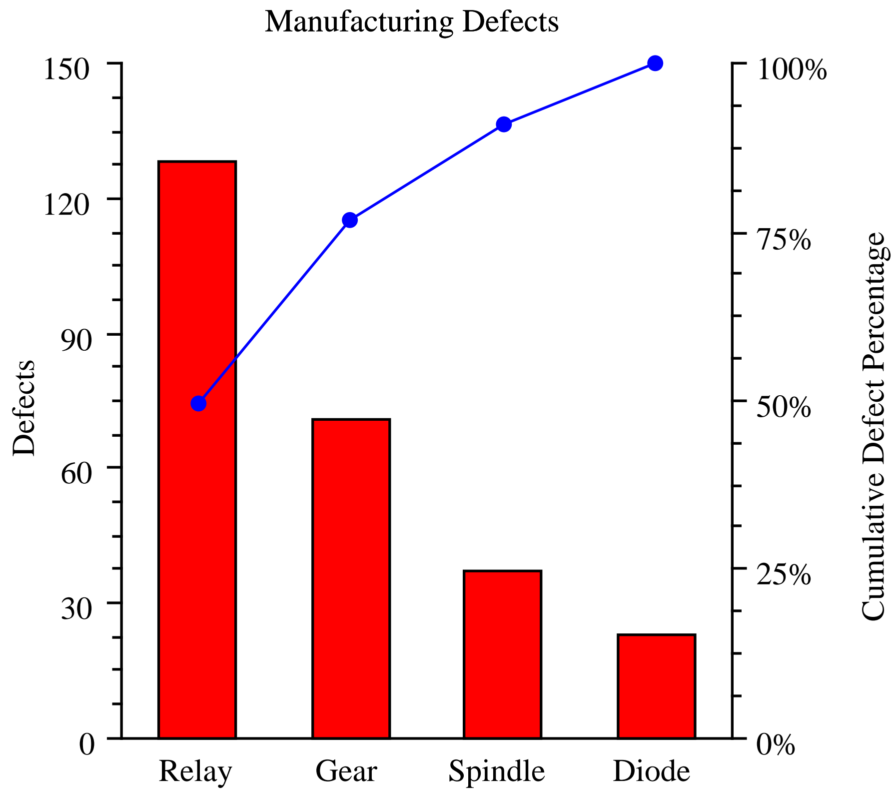
Example: Pareto Chart

The number of defects caused by four different factors were measured and plotted as a Pareto Chart. The class `ParetoChart` sorts the factors in order of number of defects. This class can be used either as an applet or as an application.

```
import com.imsl.chart.*;
import com.imsl.chart.qc.ParetoChart;

public class ParetoEx1 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) {
        AxisXY axisBar = new AxisXY(chart);

        int numberDefects[] = {71, 23, 128, 37};
        String labels[] = {"Gear", "Diode", "Relay", "Spindle"};
        ParetoChart pareto = new ParetoChart(axisBar, labels, numberDefects);
        pareto.setFillColor(java.awt.Color.red);
        pareto.addCumulativeLine();
        Data cumulativeLine = pareto.getCumulativeLine();
        cumulativeLine.setMarkerType(Data.MARKER_TYPE_FILLED_CIRCLE);
        cumulativeLine.setLineColor(java.awt.Color.blue);
        cumulativeLine.setMarkerColor(java.awt.Color.blue);

        chart.getChartTitle().setTitle("Manufacturing Defects");
        axisBar.getAxisY().getAxisTitle().setTitle("Defects");
        pareto.getCumulativeAxis().getAxisY().getAxisTitle().
                setTitle("Cumulative Defect Percentage");
    }

    public static void main(String argv[]) {
        JFrameChart frame = new JFrameChart();
        ParetoEx1.setup(frame.getChart());
        frame.setVisible(true);
    }
}
```

Output



Chapter 28: Chart 3D

Types

class Chart3D 1837
class JFrameChart3D 1840
class ChartNode3D 1842
class Background 1852
class Canvas3DChart 1852
class BufferedPaint 1856
class ChartLights 1857
class AmbientLight 1857
class DirectionalLight 1858
class PointLight 1860
class AxisXYZ 1861
class AxisBox 1863
class Axis3D 1865
class AxisLabel 1868
class AxisLine 1869
class AxisTitle 1869
class MajorTick 1870
class Surface 1870
class Data 1881
interface ColorFunction 1893
class ColormapLegend 1893

Chart3D class

```
public class com.imsl.chart3d.Chart3D extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

Root node of a 3d chart tree.

Constructor

Chart3D

`public Chart3D()`

Description

Creates a new instance of Chart3D

Methods

addToSceneGraph

`protected void addToSceneGraph(Group parent)`

cleanup

`public void cleanup()`

Description

Cleanup memory use and references used by the chart. Typically it should be invoked by an applet's destroy method.

clone

`public Object clone()`

Description

Returns a clone of the graphics tree.

Returns

an `Object` which is a clone of this graphics tree

clone

`protected Object clone(Map hashClonedNode)`

Description

Returns a clone of this node.

Parameter

`hashClonedNode` – the `Hashtable` to be cloned

Returns

an `Object` which is a clone of this node

finalize

`protected void finalize()`

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.

getCanvas

`public Canvas3D getCanvas()`

getKeyboard

`public boolean getKeyboard()`

Description

Returns the value of the “Keyboard” attribute. If true then the mouse can be used to zoom, translate and reset the chart. Its default value is true.

Returns

the value for the “Keyboard” attribute.

getOrbit

`public boolean getOrbit()`

Description

Returns the value of the “Orbit” attribute. If true then the mouse can be used to rotate, zoom and translate the chart. Its default value is true.

Returns

the value for the “Orbit” attribute.

getViewPlatformTransformation

`public void getViewPlatformTransformation(Transform3D t3d)`

Description

Sets the transformation for the view platform.

Parameter

`t3d` – is set to the ViewPlatform transformation.

resetViewPlatformTransformation

`public void resetViewPlatformTransformation()`

Description

Resets the view platform transformation to its default value.

setCanvas

```
public void setCanvas(Canvas3D canvas)
```

setKeyboard

```
public void setKeyboard(boolean keyboard)
```

Description

Sets the value of the “Keyboard” attribute. If true then the keyboard can be used to zoom, translate and reset the chart.

Parameter

`keyboard` – is the value for the “Keyboard” attribute.

setOrbit

```
public void setOrbit(boolean orbit)
```

Description

Sets the value of the “Orbit” attribute. If true then the mouse can be used to rotate, zoom and translate the chart.

Parameter

`orbit` – is the value for the “Orbit” attribute.

setViewPlatformTransformation

```
public void setViewPlatformTransformation(Transform3D t3d)
```

Description

Sets the transformation for the view platform.

Parameter

`t3d` – is the new ViewPlatform transformation.

JFrameChart3D class

```
public class com.imsl.chart3d.JFrameChart3D extends javax.swing.JFrame
implements Serializable
```

JFrameChart3D 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.

Constructors

JFrameChart3D

```
public JFrameChart3D()
```

Description

Creates new JFrameChart3D to display a chart.

JFrameChart3D

```
public JFrameChart3D(Chart3D chart)
```

Description

Creates new JFrameChart3D to display a given chart.

Parameter

`chart` – is the Chart to be displayed

Methods

getCanvas

```
public Canvas3DChart getCanvas()
```

Description

Returns the Canvas3DChart into which the chart is drawn.

getChart3D

```
public Chart3D getChart3D()
```

Description

Return the Chart object.

Returns

the chart being displayed by this container

render

```
public void render()
```

Description

Renders the 3D chart node tree into a Java 3D scene graph.

ChartNode3D class

```
abstract public class com.imsl.chart3d.ChartNode3D extends
com.imsl.chart.AbstractChartNode implements Serializable
```

The base class of all of the nodes in the 3D chart tree.

Fields

AXIS_TITLE_AT_END

```
static final public int AXIS_TITLE_AT_END
```

Value for attribute "AxisTitlePosition" indicating that the axis title should be placed at the end of the axis.

AXIS_TITLE_PARALLEL

```
static final public int AXIS_TITLE_PARALLEL
```

Value for attribute "AxisTitlePosition" indicating that the axis title should be placed parallel to the axis.

DATA_TYPE_LINE

```
static final public int DATA_TYPE_LINE
```

Value for attribute "DataType" indicating that the data points should be connected with line segments. This is the default setting.

DATA_TYPE_MARKER

```
static final public int DATA_TYPE_MARKER
```

Value for attribute "DataType" indicating that a marker should be drawn at each data point.

DATA_TYPE_PICTURE

```
static final public 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.

DATA_TYPE_TUBE

```
static final public int DATA_TYPE_TUBE
```

Value for attribute "DataType" indicating that a tube connecting the data points should be drawn. Tubes are similar to lines, but tubes are shaded. The diameter of the tube is controlled by the attribute "LineWidth". Tube color is controlled by the attribute "LineColor".

MARKER_TYPE_CUBE

```
static final public int MARKER_TYPE_CUBE
```

Flag for a cube data marker.

MARKER_TYPE_CUSTOM

```
static final public int MARKER_TYPE_CUSTOM
```

Flag for a custom marker

MARKER_TYPE_PLUS

```
static final public int MARKER_TYPE_PLUS
```

Flag for a 3D plus sign data marker.

MARKER_TYPE_SIMPLE_CUBE

```
static final public int MARKER_TYPE_SIMPLE_CUBE
```

Flag for a simple cube (no edge) data marker.

MARKER_TYPE_SIMPLE_PLUS

```
static final public int MARKER_TYPE_SIMPLE_PLUS
```

Flag for a simple 2D plus sign (no edge) data marker.

MARKER_TYPE_SIMPLE_TETRAHEDRON

```
static final public int MARKER_TYPE_SIMPLE_TETRAHEDRON
```

Flag for a simple tetrahedron (no edge) data marker.

MARKER_TYPE_SPHERE

```
static final public int MARKER_TYPE_SPHERE
```

Flag for a sphere data marker.

MARKER_TYPE_TETRAHEDRON

```
static final public int MARKER_TYPE_TETRAHEDRON
```

Flag for a tetrahedron data marker.

Constructor

ChartNode3D

```
public ChartNode3D(ChartNode3D parent)
```

Description

Construct a `ChartNode3D` object.

Parameter

`parent` – the `ChartNode3D` parent of this object

Methods

addToSceneGraph

```
abstract protected void addToSceneGraph(Group parent)
```

Description

Called to add this object to the scene graph.

Parameter

`parent` – is the node in the scene graph at which this object is to be added.

getAxisTitlePosition

```
public int getAxisTitlePosition()
```

Description

Returns the value of the “AxisTitlePosition” attribute.

Returns

The `int` value of the “AxisTitlePosition” attribute, if defined. Otherwise, AXIS_TITLE_AT_END is returned.

getBoundingSphere

```
public BoundingSphere getBoundingSphere()
```

Description

Gets the spherical bounding region object `BoundingSphere`.

Returns

a `BoundingSphere` object which is defined by a centerpoint and a radius.

getChildren

```
final public ChartNode3D[] getChildren()
```

Description

Returns an array of the children of this node. If there are no children, a 0-length array is returned.

Returns

a `ChartNode3D` array which contains the children of this node

getColorFunction

```
public ColorFunction getColorFunction()
```

Description

Returns the value of the “ColorFunction” attribute.

Returns

The `ColorFunction` value of the “ColorFunction” attribute, if defined. If not defined 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, zmin, zmax

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_MARKER` is returned.

getLightingEnabled

`public boolean getLightingEnabled()`

Description

Returns the value of the “LightingEnabled” attribute.

Returns

true if lights are to be used. Its default value is `true`.

getMarkerPulsingCycle

`public double getMarkerPulsingCycle()`

Description

Returns the value of the “MarkerPulsingCycle” attribute.

Returns

The `double` value of the “MarkerPulsingCycle” attribute, if defined. Otherwise, a default of 0.0 is returned.

getMarkerPulsingCycleOffset

`public double getMarkerPulsingCycleOffset()`

Description

Returns the value of the “MarkerPulsingCycleOffset” attribute.

Returns

The `double` value of the “MarkerPulsingCycleOffset” attribute, if defined. Otherwise, a default of 0.0 is returned.

getMarkerPulsingMaximumScale

`public double getMarkerPulsingMaximumScale()`

Description

Returns the value of the “MarkerPulsingMaximumScale” attribute.

Returns

The `double` value of the "MarkerPulsingMaximumScale" attribute, if defined. Otherwise, a default of 2.0 is returned.

getMarkerPulsingMinimumScale

```
public double getMarkerPulsingMinimumScale()
```

Description

Returns the value of the "MarkerPulsingMinimumScale" attribute.

Returns

The `double` value of the "MarkerPulsingMinimumScale" attribute, if defined. Otherwise, a default of 0.0 is returned.

getMarkerRotatingAxis

```
public double[] getMarkerRotatingAxis()
```

Description

Returns the value of the "MarkerRotatingAxis" attribute.

Returns

The `double` value of the "MarkerRotatingAxis" attribute, if defined. Otherwise, a default of 0.0 is returned.

getMarkerRotatingCycle

```
public double getMarkerRotatingCycle()
```

Description

Returns the value of the "MarkerRotatingCycle" attribute.

Returns

The `double` value of the "MarkerRotatingCycle" attribute, if defined. Otherwise, a default of 0.0 is returned.

getMarkerRotatingCycleOffset

```
public double getMarkerRotatingCycleOffset()
```

Description

Returns the value of the "MarkerRotatingCycleOffset" attribute.

Returns

The `double` value of the "MarkerRotatingCycleOffset" attribute, if defined. Otherwise, a default of 0.0 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_CUBE is returned.

getMaterial

`public Material getMaterial()`

Description

Returns the value of the “Material” attribute.

Returns

The value of the “Material” attribute, if defined. Otherwise, a default of material is returned.

getParent

`public ChartNode3D 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 `ChartNode3D` 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.

getTitle

`public String getTitle()`

Description

Returns the value of the “Title” attribute.

Returns

the `String` value of the “Title” attribute

getViewport

`public double[] getViewport()`

Description

Returns the value of the “Viewport” attribute.

Returns

a `double`[6] array containing xmin, xmax, ymin, ymax, zmin, zmax

getVirtualUniverse

`public VirtualUniverse getVirtualUniverse()`

Description

Returns the value of the “Universe” attribute.

Returns

The value of the "Universe" attribute.

getZ

```
public double[] getZ()
```

Description

Returns the value of the "Z" attribute.

Returns

the `double` array which contains the value of the "Z" attribute

setAxisTitlePosition

```
public void setAxisTitlePosition(int value)
```

Description

Sets the value of the "AxisTitlePosition" attribute.

Parameter

`value` – "AxisTitlePosition" value. This should be `AXIS_TITLE_AT_END` or `AXIS_TITLE_PARALLEL`. `AXIS_TITLE_AT_END` is the default value.

setBoundingSphere

```
public void setBoundingSphere(BoundingSphere bounds)
```

Description

Sets the spherical bounding region object `BoundingSphere`.

Parameter

`bounds` – a `BoundingSphere` object which is defined by a centerpoint and a radius.

setColorFunction

```
public void setColorFunction(ColorFunction colorFunction)
```

Description

Sets the value of the "ColorFunction" attribute. ColorFunction defines a value-dependent coloring.

Parameter

`colorFunction` – defines a mapping from x,y,z to a color.

setDataType

```
public void setDataType(int value)
```

Description

Sets the value of the "DataType" attribute.

Parameter

`value` – "DataType" value. This should be some xor-ed combination of DATA_TYPE_LINE, DATA_TYPE_MARKER.

setLightingEnabled

```
public void setLightingEnabled(boolean lightingEnabled)
```

Description

Sets the value of the "LightingEnabled" attribute.

Parameter

`lightingEnabled` – is true if lights are to be used. Its default value is `true`.

setMarkerPulsingCycle

```
public void setMarkerPulsingCycle(double time)
```

Description

Sets the value of the "MarkerPulsingCycle" attribute. The default marker cycle time is zero. If "MarkerPulsingCycle" is greater then zero then markers pulse with the specified cycle time.

Parameter

`time` – a `double` which specifies the "MarkerPulsingCycle" time in seconds.

setMarkerPulsingCycleOffset

```
public void setMarkerPulsingCycleOffset(double offset)
```

Description

Sets the value of the "MarkerPulsingCycleOffset" attribute.

Parameter

`offset` – a `double` which specifies the "MarkerPulsingCycleOffset". This is the time, in seconds, by which a pulsing marker starting time is offset from the initial time. This allows different markers to pulse with different phases.

setMarkerPulsingMaximumScale

```
public void setMarkerPulsingMaximumScale(double max)
```

Description

Sets the value of the "MarkerPulsingMaximumScale" attribute.

Parameter

`max` – a `double` which specifies the "MarkerPulsingMaximumScale". This is the amount by which a pulsing marker is scaled at the top of a pulse. Its default value is 2.0.

setMarkerPulsingMinimumScale

```
public void setMarkerPulsingMinimumScale(double min)
```

Description

Sets the value of the "MarkerPulsingMinimumScale" attribute.

Parameter

`min` – a `double` which specifies the "MarkerPulsingMinimumScale". This is the amount by which a pulsing marker is scaled at the bottom of a pulse. Its default value is 0.0.

setMarkerRotatingAxis

```
public void setMarkerRotatingAxis(double x, double y, double z)
```

Description

Sets the value of the "MarkerRotatingAxis" attribute. The default marker cycle time is zero. If "MarkerRotatingAxis" is greater then zero then markers rotate with the specified cycle time.

Parameters

`x` – is the *x*-coordinate of the rotation axis.

`y` – is the *y*-coordinate of the rotation axis.

`z` – is the *z*-coordinate of the rotation axis.

setMarkerRotatingCycle

```
public void setMarkerRotatingCycle(double time)
```

Description

Sets the value of the "MarkerRotatingCycle" attribute. The default marker cycle time is zero. If "MarkerRotatingCycle" is greater then zero then markers rotate with the specified cycle time.

Parameter

`time` – a `double` which specifies the "MarkerRotatingCycle" time in seconds.

setMarkerRotatingCycleOffset

```
public void setMarkerRotatingCycleOffset(double offset)
```

Description

Sets the value of the "MarkerRotatingCycleOffset" attribute.

Parameter

`offset` – a `double` which specifies the "MarkerRotatingCycleOffset". This is the time, in seconds, by which a rotating marker starting time is offset from the initial time. This allows different markers to rotate with different phases.

setMarkerType

```
public void setMarkerType(int type)
```

Description

Sets the value of the "MarkerType" attribute. This indicates which marker is to be drawn.

Parameter

`type` – the `int` “MarkerType” value.

---

setMaterial

```
public void setMaterial(Material material)
```

Description

Sets the value of the “Material” attribute. This indicates which matherial is to be used when lighting a surface.

Parameter

`material` – is a Java 3D Material value.

---

setTitle

```
public void setTitle(String value)
```

Description

Sets the value of the “Title” attribute.

Parameter

`value` – a `String` which contains the “Title” value

---

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] by [0,1] This attribute affects only Axis nodes, since they contain the mappings to device space.

Parameter

`value` – A `double` array of length 6 which contains the “Viewport” values for xmin, xmax, ymin, ymax, zmin, zmax. The value saved is a copy of the input array.

---

setViewport

```
public void setViewport(double xmin, double xmax, double ymin, double ymax,
double zmin, double zmax)
```

Description

Sets the value of the “Viewport” attribute.

Parameters

`xmin` – a `double`, the minimum x-coordinate of the viewport

`xmax` – a `double`, the maximum x-coordinate of the viewport

`ymin` – a `double`, the minimum y-coordinate of the viewport

`ymax` – a `double`, the maximum y-coordinate of the viewport

`zmin` – a `double`, the minimum z-coordinate of the viewport

`zmax` – a `double`, the maximum z-coordinate of the viewport

setZ

```
public void setZ(Object value)
```

Description

Sets the value of the “Z” attribute.

Parameter

`value` – the `Object` which contains the “Z” value

Background class

```
public class com.imsl.chart3d.Background extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

Background of the chart. The chart’s background is a solid color defined by this node’s “FillColor” attribute value. The default background color is white.

This node is created by the `Chart3D` node. To disable this node, set its “Paint” attribute value to `false`.

More complex background’s can be implemented by registering `com.imsl.chart3d.Canvas3DChart.Paint` (p. 1855) objects with `com.imsl.chart3d.Canvas3DChart` (p. 1852) . These objects can be used to draw to the background either in front or behind the 3D chart.

Method

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

Canvas3DChart class

```
public class com.imsl.chart3d.Canvas3DChart extends javax.media.j3d.Canvas3D
```

A canvas for rendering a 3D chart.

Constructors

Canvas3DChart

`public Canvas3DChart()`

Description

Creates a Canvas3DChart with a new Chart3D object.

Canvas3DChart

`public Canvas3DChart(Chart3D chart)`

Description

Creates a Canvas3DChart with a given Chart3D object.

Parameter

`chart` – is the `Chart3D` object associated with this canvas.

Methods

addPostRenderPaint

`public void addPostRenderPaint(Canvas3DChart.Paint paint)`

Description

Adds a Paint object to draw to the canvas after the the 3D image is rendered.

Parameter

`paint` – is the `Paint` object to be removed.

addPreRenderPaint

`public void addPreRenderPaint(Canvas3DChart.Paint paint)`

Description

Adds a Paint object to draw to the canvas before the the 3D image is rendered.

Parameter

`paint` – implements the `paint` method to be called before the 3D image is rendered.

getChart3D

`public Chart3D getChart3D()`

Description

Returns the Chart3D associated with this canvas.

Returns

the Chart3D associated with this canvas.

paint

`public void paint(Graphics g)`

Description

Paint method overriden to correct a problem in JDK 1.4. See bug ID Bug ID 4374079 for details.

postRender

`public void postRender()`

Description

Calls the `Paint` objects added to the post-render list. This routine is called by the Java 3D rendering loop after completing all rendering to the canvas for this frame and before the buffer swap.

NOTE: Applications should *not* call this method.

postSwap

`public void postSwap()`

Description

Writes the chart to a file as a bitmap image. Use the `write` method to trigger writing of the image.

NOTE: Applications should *not* call this method.

preRender

`public void preRender()`

Description

Calls the `Paint` objects added to the pre-render list. This routine is called by the Java 3D rendering loop after clearing the canvas and before any rendering has been done for this frame.

NOTE: Applications should *not* call this method.

removePostRenderPaint

`public void removePostRenderPaint(Canvas3DChart.Paint paint)`

Description

Removes a Paint object from the list of post-render Paint objects.

Parameter

`paint` – is the `Paint` object to be removed.

removePreRenderPaint

`public void removePreRenderPaint(Canvas3DChart.Paint paint)`

Description

Removes a Paint object from the list of pre-render Paint objects.

Parameter

`paint` – implements the `paint` method to be called before the 3D image is rendered.

render

`public void render()`

Description

Creates a scene graph from the chart tree and starts rendering the scene graph into this canvas. This method must be called after the chart tree has been created and associated with this canvas.

write

`public void write(String filename, String format)`

Description

Write the canvas as an image file after it is next redrawn.

Parameters

`filename` – is the name of the file to which the image is to be written.

`format` – is the image format name, such as “PNG” or “JPEG”. The supported formats are the same as for ImageIO.write.

Canvas3DChart.Paint interface

`public interface com.imsl.chart3d.Canvas3DChart.Paint`

Interface for 2D drawing on the canvas before or after the the 3D image is drawn.

Method

paint

`public void paint(Graphics graphics)`

Parameter

`graphics` – is a `java.awt.Graphics2D` object.

BufferedPaint class

```
public class com.imsl.chart3d.BufferedPaint implements
com.imsl.chart3d.Canvas3DChart.Paint, Cloneable
```

A Paint object cached into an image.

This is used to cache a static image that will be painted into the canvas containing a 3D chart. Since the 3D chart canvas will be repainted many times each second, it is faster to compose the image once.

Constructor

BufferedPaint

```
public BufferedPaint(Canvas3DChart.Paint paint, int x, int y, int width, int
height, Component component)
```

Description

The `paint` method in `Canvas3DChart.Paint` is written into an image of size `width` by `height`. Any whitespace around the image is trimmed. The trimmed image is then used to paint onto the canvas.

Parameters

`paint` – is the `Canvas3DChart.Paint` object to be cached.

`x` – is the pixel position in the canvas of the left edge of the image. If x is negative then |x| is the distance from the right edge of the image to the right edge of the component.

`y` – is the pixel position in the canvas of the top edge of the image. If y is negative then |y| is the distance from the bottom edge of the image to the bottom edge of the component.

`width` – is the maximum width of the image.

`height` – is the maximum height of the image.

`component` – is the `Component` in which the image is to be painted.

Methods

clone

```
public Object clone() throws CloneNotSupportedException
```

paint

```
public void paint(Graphics g)
```

Description

Paint the image onto the canvas. This method should be called by the canvas, not by any application.

Parameter

g – is the `Graphics` object.

trim

```
public void trim()
```

Description

Returns a subimage with the white space trimmed off.

ChartLights class

```
public class com.imsl.chart3d.ChartLights extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

Default set of lights.

`ChartLights` defines a default set of lights for the chart. If customized lights are desired, then this node can be disabled by setting its “Paint” attribute to false and explicitly adding lights to the scene.

Method

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

AmbientLight class

```
public class com.imsl.chart3d.AmbientLight extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

An ambient light. Ambient light is light that seems to come from all directions.

Constructor

AmbientLight

```
public AmbientLight(Chart3D parent)
```

Description

Creates an ambient light.

Parameter

`parent` – is the `Chart3D` parent of this node.

Method

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

DirectionalLight class

```
public class com.imsl.chart3d.DirectionalLight extends
com.imsl.chart3d.ChartNode3D implements Serializable
```

A directional light.

A directional light is an oriented light with an origin at infinity. The direction is defined by the attribute "Direction".

The light's position is in a coordinate system in which the default viewport is the cube [-1,1] by [-1,1] by [-1,1].

Constructors

DirectionalLight

```
public DirectionalLight(Chart3D parent)
```

Description

Creates a directional light pointing in the negative z direction.

Parameter

`parent` – is the `Chart3D` parent of this node.

DirectionalLight

```
public DirectionalLight(Chart3D parent, double x, double y, double z)
```

Description

Creates a directional light pointing with a specified direction.

Parameters

`parent` – is the `Chart3D` parent of this node.

`x` – is the *x*-component of the direction vector.

`y` – is the *y*-component of the direction vector.

`z` – is the *z*-component of the direction vector.

Methods

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

getDirection

```
public Vector3f getDirection()
```

Description

Returns the value of the “Direction” attribute.

Returns

The `Vector3f` value of the “Direction” attribute, if defined. Otherwise, (0, 0, -1) is returned.

setDirection

```
public void setDirection(Vector3f direction)
```

Description

Sets the value of the “Direction” attribute to a light direction.

Parameter

`direction` – `Vector3f` direction.

setDirection

```
public void setDirection(double x, double y, double z)
```

Description

Sets the value of the “Direction” attribute to a light direction.

Parameters

x – is the *x*-component of the direction vector.

y – is the *y*-component of the direction vector.

z – is the *z*-component of the direction vector.

PointLight class

`public class com.imsl.chart3d.PointLight extends com.imsl.chart3d.ChartNode3D implements Serializable`

A point light source.

A point light source is at a fixed point in space and radiates light equally in all directions away from the light source. The light's position is defined by the attribute "Position".

The light's position is in a coordinate system in which the default viewport is the cube [-1,1] by [-1,1] by [-1,1].

Constructors

PointLight

`public PointLight(Chart3D parent)`

Description

Creates a point light source at the origin.

Parameter

`parent` – is the `Chart3D` parent of this node.

PointLight

`public PointLight(Chart3D parent, double x, double y, double z)`

Description

Creates a point light at a specified position.

Parameters

`parent` – is the `Chart3D` parent of this node.

x – is the *x*-component of the position.

y – is the *y*-component of the position.

z – is the *z*-component of the position.

Methods

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

getPosition

```
public Point3f getPosition()
```

Description

Returns the value of the “Position” attribute.

Returns

The `Point3f` value of the “Position” attribute, if defined. Otherwise, (0, 0, 0) is returned.

setPosition

```
public void setPosition(Point3f position)
```

Description

Sets the value of the “Point” attribute to a light point.

Parameter

`position` – is the location of the light.

setPosition

```
public void setPosition(double x, double y, double z)
```

Description

Sets the value of the “Point” attribute to a light point.

AxisXYZ class

```
public class com.imsl.chart3d.AxisXYZ extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

The axes for an x-y-z chart.

This node is used when the mapping to and from user and device space can be decomposed into an x, a y and a z mapping.

Constructor

AxisXYZ

`public AxisXYZ(Chart3D chart)`

Description

Create an `AxisXYZ`. This also creates three `Axis3D` nodes as children of this node. They hold the decomposed mapping.

Parameter

`chart` – the `Chart3D` parent of this node

Methods

addToSceneGraph

`protected void addToSceneGraph(Group parent)`

getAxisBox

`public AxisBox getAxisBox()`

Description

Return the axis box node.

Returns

the `AxisBox` node

getAxisX

`public Axis3D getAxisX()`

Description

Return the x-axis node.

Returns

the `Axis3D` x-axis node

getAxisY

`public Axis3D getAxisY()`

Description

Return the y-axis node.

Returns

the `Axis3D` y-axis node

getAxisZ

`public Axis3D getAxisZ()`

Description

Return the z-axis node.

Returns

the `Axis3D` z-axis node

mapCubeToUser

```
public void mapCubeToUser(double cubeX, double cubeY, double cubeZ, double[]
userXYZ)
```

Description

Map the cube coordinates to user coordinates.

Parameters

`cubeX` – an `int`, the cube x-coordinate

`cubeY` – an `int`, the cube y-coordinate

`cubeZ` – an `int`, the cube z-coordinate

`userXYZ` – a `double`[3] array on input. On output, the user coordinates.

mapUserToCube

```
public void mapUserToCube(double userX, double userY, double userZ, double[]
cubeXYZ)
```

Description

Map the user coordinates (userX,userY) to the cube coordinates cubeXYZ.

Parameters

`userX` – a `double`, the user x-coordinate

`userY` – a `double`, the user y-coordinate

`userZ` – a `double`, the user y-coordinate

`cubeXYZ` – an `int`[3] array on input. On output, the cube coordinates.

AxisBox class

```
public class com.imsl.chart3d.AxisBox extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

Box behind the axis.

The axis box is drawn behind the axis. The color is defined by this node's "FillColor" attribute value. The default color is a transparent gray.

The box also includes grid lines. They are drawn with this node's "LineColor" attribute.

This node is created by the `Chart3D` node. To disable this node, set its "Paint" attribute value to `false`.

Fields

FACE_XA

`static final public int FACE_XA`

Show the x = a face of the box.

FACE_XB

`static final public int FACE_XB`

Show the x = b face of the box.

FACE_YA

`static final public int FACE_YA`

Show the y = a face of the box.

FACE_YB

`static final public int FACE_YB`

Show the y = b face of the box.

FACE_ZA

`static final public int FACE_ZA`

Show the z = a face of the box.

FACE_ZB

`static final public int FACE_ZB`

Show the z = b face of the box.

Methods

addToSceneGraph

`protected void addToSceneGraph(Group parent)`

getVisibleFaces

`public int getVisibleFaces()`

Description

Returns the flag indicating which faces of the box are to be drawn. The default value is `FACE_XB | FACE_YB | FACE_ZA`.

setVisibleFaces

`public void setVisibleFaces(int visibleFaces)`

Description

Sets the "VisibleFaces" attribute indicating which faces of the box are to be drawn.

Parameter

`visibleFaces` – is an or-ed combination of the flags `FACE_XA`, `FACE_YA`, `FACE_ZA`, `FACE_XB`, `FACE_YB`, `FACE_ZB`.

Axis3D class

`public class com.imsl.chart3d.Axis3D extends com.imsl.chart3d.ChartNode3D implements Serializable`

An x-axis, y-axis or a z-axis.

Axis3D is created by `com.imsl.chart3d.AxisXYZ` (p. 1861) as its child. It can be retrieved using the method `com.imsl.chart3d.AxisXYZ.getAxisX` (p. 1862) or `com.imsl.chart3d.AxisXYZ.getAxisY` (p. 1862) or `com.imsl.chart3d.AxisXYZ.getAxisZ` (p. 1862) .

It in turn creates the following child nodes: `com.imsl.chart3d.AxisLine` (p. 1869) , `com.imsl.chart3d.AxisLabel` (p. 1868) , `com.imsl.chart3d.AxisTitle` (p. 1869) and `com.imsl.chart3d.MajorTick` (p. 1870) .

The number of tick marks ("Number" attribute) is set to 4, but autoscaling can change this value.

Methods

addToSceneGraph

`protected void addToSceneGraph(Group parent)`

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

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.

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

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`, or `AXIS_Z`

getWindow

```
public double[] getWindow()
```

Description

Returns the window for an Axis1D.

Returns

an array of length two containing the range of this axis.

setFirstTick

```
public void setFirstTick(double firstTick)
```

Description

Convenience routine to set the "FirstTick" attribute.

Parameter

`firstTick` – a `double`, the location of the first tick

setTickInterval

```
public void setTickInterval(double tickInterval)
```

Description

Sets the tick interval.

Parameter

`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.

Parameter

`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.

setWindow

```
public void setWindow(double[] window)
```

Description

Sets the window for an Axis1D.

Parameter

`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` – is the value of the left/bottom end of the axis.

`max` – is the value of the right/top end of the axis.

AxisLabel class

```
public class com.imsl.chart3d.AxisLabel extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

The labels on an axis.

`AxisLabel` is created by `com.imsl.chart3d.Axis3D` (p. 1865) as its child. It can be retrieved using the method `com.imsl.chart3d.Axis3D.getAxisLabel` (p. 1865) .

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”.

Methods

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

getLabels

```
public String[] getLabels()
```

Description

Returns the "Labels" attribute.

Returns

a `String` array containing the axis labels, if set. Otherwise, `null` is returned.

setLabels

```
public void setLabels(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`.

Parameter

`value` – a `String` array containing the labels for the major tick marks

AxisLine class

```
public class com.imsl.chart3d.AxisLine extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

The axis line.

`AxisLine` is created by `com.imsl.chart3d.Axis3D` (p. 1865) as its child. It can be retrieved using the method `com.imsl.chart3d.Axis3D.getAxisLine` (p. 1865) .

Method

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

AxisTitle class

```
public class com.imsl.chart3d.AxisTitle extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

Axis title. The position of the axis title is controlled by the attribute “AxisTitlePosition”. It can be either parallel to the axis line or at the end of the axis line.

This node is created by the `Axis3D` node. To disable this node, set its “Paint” attribute value to `false`.

Method

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

MajorTick class

```
public class com.imsl.chart3d.MajorTick extends com.imsl.chart3d.ChartNode3D
implements Serializable
```

Major ticks marks.

This node is created by the `Axis3D` node. To disable this node, set its “Paint” attribute value to `false`.

Method

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

Surface class

```
public class com.imsl.chart3d.Surface extends com.imsl.chart3d.Data implements
Serializable
```

Surface from a function or from a set of scattered data points.

Fields

SURFACE_TYPE_FLAT

`static final public int SURFACE_TYPE_FLAT`

Draws the surface using flat shading.

In a flat shaded surface, each polygon has a uniform color.

SURFACE_TYPE_GOURAUD

`static final public int SURFACE_TYPE_GOURAUD`

Draws the surface using Gouraud shading. In a Gouraud shaded surface, colors are interpolated across each polygon.

SURFACE_TYPE_MESH

`static final public int SURFACE_TYPE_MESH`

Draws the surface as a mesh.

SURFACE_TYPE_NICEST

`static final public int SURFACE_TYPE_NICEST`

Draws the surface using the best shading available.

Constructors

Surface

`public Surface(AxisXYZ parent, double[] x, double[] y, double[] z)`

Description

Creates a surface from a scattered set of 3D points.

A surface is created from a scattered set of points. A function is interpolated to the given point over the rectangular area determined by that point's ranges.

Parameters

`parent` – an AxisXYZ object, the parent of this node.

`x` – is the array of x values.

`y` – is the array of y values.

`z` – is the array of z values.

Surface

`public Surface(AxisXYZ parent, double[] x, double[] y, double[][] z)`

Description

Creates a surface from a gridded data set. A surface is created from a grid of points in a rectangular area. The point `z[i][j]` is the *z*-value at (`x[i]`, `y[j]`).

Parameters

`parent` – an AxisXYZ object, the parent of this node.

`x` – is the array of *x* values.

`y` – is the array of *y* values.

`z` – is the two-dimensional array of *z* values of size `x.length` by `y.length`.

Surface

```
public Surface(AxisXYZ parent, double[] x, double[] y, double[] z, Color[]
color)
```

Description

Creates a surface from a scattered set of 3D points with a color given at each point.

A surface is created from a scattered set of points. A function is interpolated to the given point over the rectangular area determined by that point's ranges.

Parameters

`parent` – an AxisXYZ object, the parent of this node.

`x` – is the array of *x* values.

`y` – is the array of *y* values.

`z` – is the array of *z* values.

`color` – is array of color values at each point.

Surface

```
public Surface(AxisXYZ parent, double[] x, double[] y, double[][] z, Color[][]
color)
```

Description

Creates a colored surface from a gridded data set.

A surface is created from a grid of points in a rectangular area. The point `z[i][j]` is the *z*-value at (`x[i]`, `y[j]`).

Parameters

`parent` – an AxisXYZ object, the parent of this node.

`x` – is the array of *x* values.

`y` – is the array of *y* values.

`z` – is the two-dimensional array of *z* values.

`color` – is the two-dimensional array of color values. The array must have the same size as the array `z`.

Surface

```
public Surface(AxisXYZ parent, Surface.ZFunction zFunction, double xmin, double
xmax, double ymin, double ymax)
```

Description

Creates a surface from a function. A surface is created by evaluation of the function on a grid of points in a rectangular area, [xmin,xmax] by [ymin,ymax], of the *xy*-plane.

Parameters

`parent` – an AxisXYZ object, the parent of this node.

`zFunction` – the function, $z = f(x,y)$.

`xmin` – the minimum x-value of the function rectangle.

`xmax` – the maximum x-value of the function rectangle.

`ymin` – the minimum y-value of the function rectangle.

`ymax` – the maximum y-value of the function rectangle.

Methods

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

getNumberGridPointsX

```
public int getNumberGridPointsX()
```

Description

Returns the value of the “NumberGridPointsX” attribute.

This is the grid points in the x-direction for surfaces defined by a function.

Returns

The number of grid points in the x-direction. Default is 40.

getNumberGridPointsY

```
public int getNumberGridPointsY()
```

Description

Returns the value of the “NumberGridPointsY” attribute.

This is the grid points in the y-direction for surfaces defined by a function.

Returns

The number of grid points in the y-direction. Default is 40.

getSurfaceType

```
public int getSurfaceType()
```

Description

Returns the attribute "SurfaceType".

Returns

one of SURFACE_TYPE_MESH, SURFACE_TYPE_FLAT, SURFACE_TYPE_GOURAUD, SURFACE_TYPE_NICEST or SURFACE_TYPE_MESH or-ed with one of the other types. Default value is SURFACE_TYPE_NICEST.

setNumberGridPointsX

```
public void setNumberGridPointsX(int nx)
```

Description

Sets the value of the "NumberGridPointsX" attribute.

This is the grid points in the x-direction for surfaces defined by a function.

Parameter

nx – The number of grid points in the x-direction. Default is 40.

setNumberGridPointsY

```
public void setNumberGridPointsY(int ny)
```

Description

Sets the value of the "NumberGridPointsY" attribute.

This is the grid points in the y-direction for surfaces defined by a function.

Parameter

ny – The number of grid points in the y-direction. Default is 40.

setSurfaceType

```
public void setSurfaceType(int surfaceType)
```

Description

Sets the attribute "SurfaceType".

Parameter

surfaceType – is one of SURFACE_TYPE_MESH, SURFACE_TYPE_FLAT, SURFACE_TYPE_GOURAUD, SURFACE_TYPE_NICEST or SURFACE_TYPE_MESH or-ed with one of the other types.

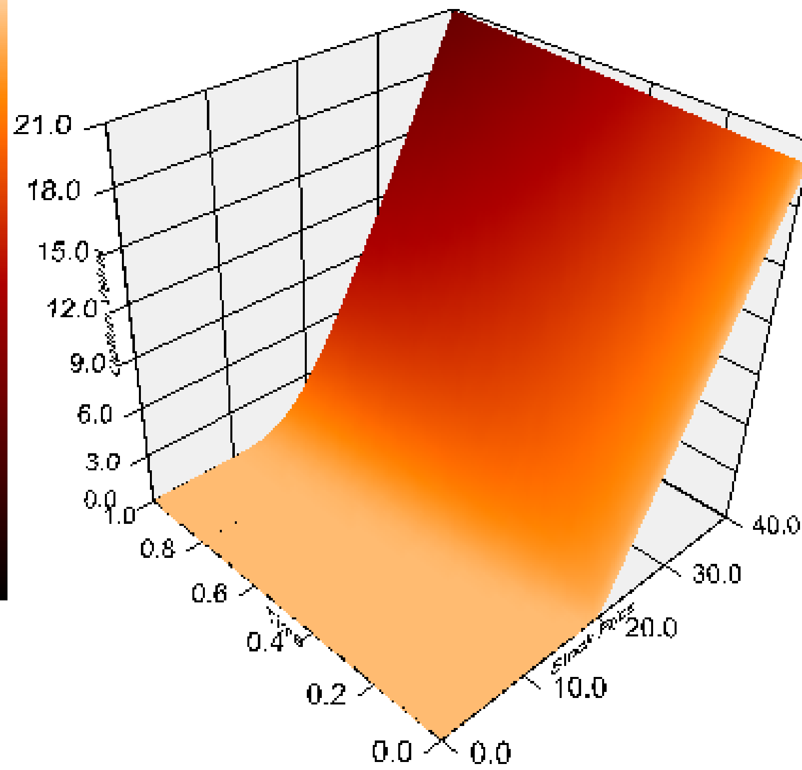
Example: Call Option Surface shaded by Vega

A surface chart of call option values shaded by vega is rendered. The X, Y, and Z axes represent Stock Price, Time, and Option Value respectively.

```
import com.imsl.chart3d.*;
import com.imsl.chart.Colormap;
import com.imsl.stat.Cdf;
import java.awt.Color;

// Surface chart of call option value shaded by vega.
public class SurfaceEx1 extends JFrameChart3D {

    // Creates new form CallOptionSurface
    public SurfaceEx1() {
        Chart3D chart = getChart3D();
        chart.setTextFormat("0.0000");

        AxisXYZ axis = new AxisXYZ(chart);
        axis.setAxisTitlePosition(AxisXYZ.AXIS_TITLE_PARALLEL);
        axis.setTextFormat("0.0");

        axis.getAxisX().getAxisTitle().setTitle("Stock Price");
        axis.getAxisY().getAxisTitle().setTitle("Time");
        axis.getAxisZ().getAxisTitle().setTitle("Option Value");

        double strike = 20.0;
        double rate = 0.045;
        double sigma = 0.25;
        CallOption callOption = new CallOption(strike, rate, sigma);

        double minStock = 0.0;
        double maxStock = 2.0 * strike;
        double minTime = 0.0;
        double maxTime = 1.0;
        Surface surface = new Surface(axis, callOption, minStock,
                maxStock, minTime, maxTime);
        surface.setColorFunction(callOption);
        surface.setSurfaceType(Surface.SURFACE_TYPE_MESH
                | Surface.SURFACE_TYPE_NICEST);

        ColormapLegend colormapLegend = new ColormapLegend(chart,
                Colormap.RED_TEMPERATURE, -10., 60.);
        colormapLegend.setTitle("Vega");
        colormapLegend.setTextFormat("0.00");
        colormapLegend.setNumber(25);
        colormapLegend.setAutoscaleInput(ColormapLegend.AUTOSCALE_WINDOW);
        colormapLegend.setAutoscaleOutput(ColormapLegend.AUTOSCALE_NUMBER);

        this.setSize(375, 375);
        render();
    }

    public class CallOption implements Surface.ZFunction, ColorFunction {
```

```
        private double strike, rate, sigma;

        //Compute call option value using the Black-Scholes formula.
        public CallOption(double strike, double rate, double sigma) {
            this.strike = strike;
            this.rate = rate;
            this.sigma = sigma;

        }

        public double f(double stock, double time) {
            double d1 = (Math.log(stock / strike)
                    + (rate + 0.5 * sigma * sigma) * time)
                    / (sigma * Math.sqrt(time));
            double d2 = d1 - sigma * Math.sqrt(time);
            return stock * Cdf.normal(d1) - strike
                    * Math.exp(-rate * time) * Cdf.normal(d2);
        }

        public double delta(double stock, double time) {
            double d1 = (Math.log(stock / strike)
                    + (rate + 0.5 * sigma * sigma) * time)
                    / (sigma * Math.sqrt(time));
            return Cdf.normal(d1);
        }

        public double vega(double stock, double time) {
            double d1 = (Math.log(stock / strike)
                    + (rate + 0.5 * sigma * sigma) * time)
                    / (sigma * Math.sqrt(time));
            return stock * Math.sqrt(time) * Cdf.normal(d1);
        }

        public Color color(double stock, double time, double optionValue) {
            double vega = vega(stock, time);
            double s = (vega + 10.0) / (60.0 + 10.);
            return Colormap.RED_TEMPERATURE.color(s);
        }
    }

    public static void main(String args[]) {
        new SurfaceEx1().setVisible(true);
    }
}
```

Output



Example: Daily Carbon Monoxide Levels by Time of Day

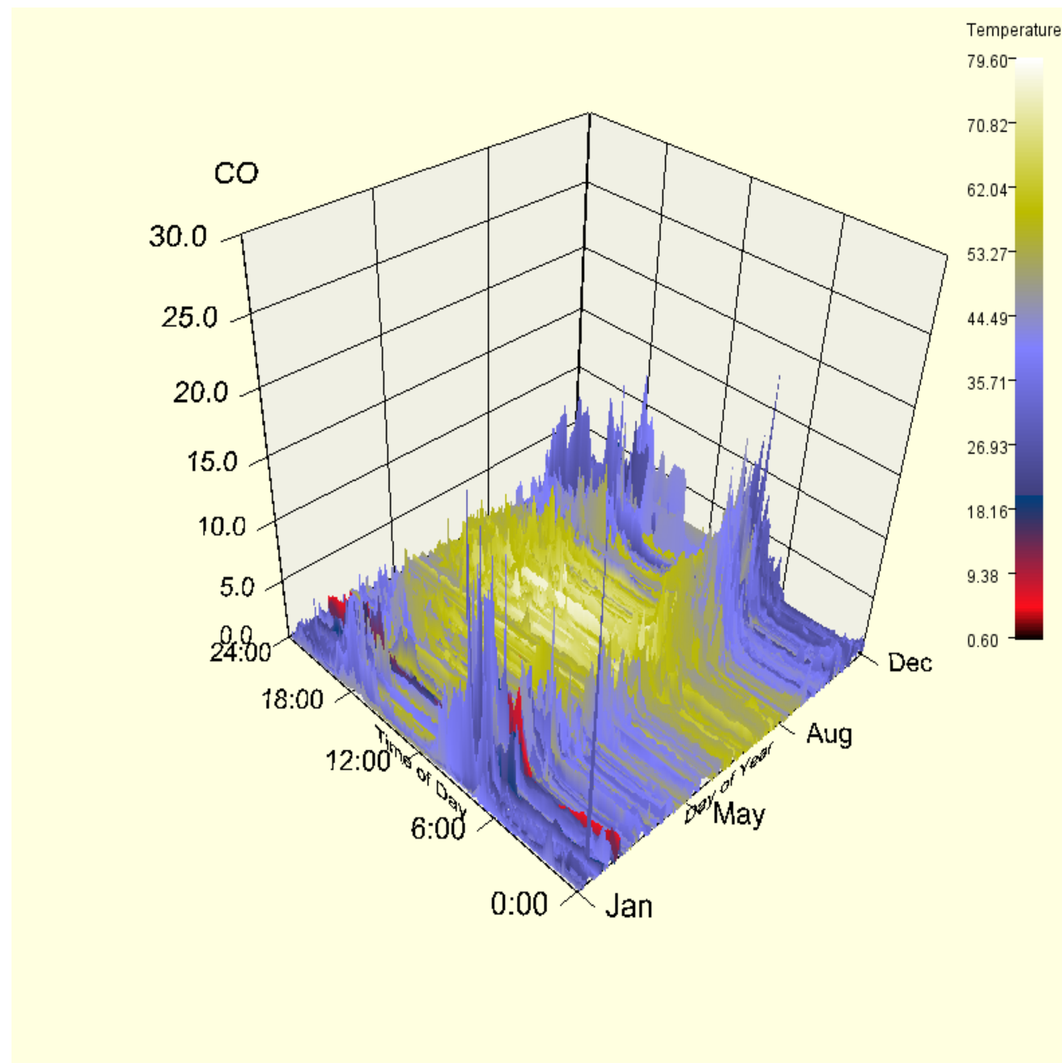
A surface plot is rendered to show the carbon monoxide levels in a metropolitan area over the course of a year by time of day.

```
import com.imsl.chart3d.*;
import com.imsl.chart.Colormap;
import com.imsl.io.*;
import java.awt.Color;
import java.io.*;
import java.sql.SQLException;
import java.util.GregorianCalendar;
```

```
// CO surface shaded by temperature
public class SurfaceEx2 extends JFrameChart3D {

    static private final int xMax = 365;
    static private final int yMin = 0;
    static private final int yMax = 144;

    private double temp[][];
    private double co[][];

    private double tempMin, tempMax;

    private Colormap colormap = Colormap.SPECTRAL;

    // Creates new form COSurface
    public SurfaceEx2() throws IOException, SQLException {
        temp = readData("temp.csv");
        co = readData("co.csv");

        tempMin = temp[0][0];
        tempMax = temp[0][0];
        for (int i = 0; i < xMax; i++) {
            for (int j = 0; j < yMax; j++) {
                tempMin = Math.min(temp[i][j], tempMin);
                tempMax = Math.max(temp[i][j], tempMax);
            }
        }

        Chart3D chart = getChart3D();
        chart.getBackground().setFillColor("lightyellow");
        AxisXYZ axis = new AxisXYZ(chart);
        axis.setAxisTitlePosition(AxisXYZ.AXIS_TITLE_PARALLEL);

        axis.getAxisX().getAxisTitle().setTitle("Day of Year");
        final GregorianCalendar initialDate = new GregorianCalendar(2000,
                GregorianCalendar.JANUARY, 1);
        axis.getAxisX().setAutoscaleOutput(0);
        GregorianCalendar lastDate = (GregorianCalendar) initialDate.clone();
        lastDate.add(GregorianCalendar.DATE, 365);
        axis.getAxisX().setWindow(initialDate.getTimeInMillis(),
                lastDate.getTimeInMillis());
        axis.getAxisX().setTextFormat(new java.text.SimpleDateFormat("MMM"));

        axis.getAxisY().getAxisTitle().setTitle("Time of Day");
        axis.getAxisY().setAutoscaleOutput(0);
        axis.getAxisY().setWindow(yMin, yMax);
        String labelsY[] = {"0:00", "6:00", "12:00", "18:00", "24:00"};
        axis.getAxisY().getAxisLabel().setLabels(labelsY);

        axis.getAxisZ().getAxisTitle().setTitle("CO");
        axis.getAxisZ().getAxisTitle().
                setAxisTitlePosition(axis.AXIS_TITLE_AT_END);
        axis.getAxisZ().setTextFormat("0.0");

        GregorianCalendar date = (GregorianCalendar) initialDate.clone();
        double x[] = new double[xMax];
```

```
        for (int i = 0; i < xMax; i++) {
            x[i] = date.getTimeInMillis();
            date.add(GregorianCalendar.DATE, 1);
        }

        double y[] = new double[yMax];
        for (int j = 0; j < yMax; j++) {
            y[j] = j - 1;
        }

        Color color[][] = new Color[xMax][yMax];
        for (int i = 0; i < xMax; i++) {
            for (int j = 0; j < yMax; j++) {
                double t = (tempMax - temp[i][j]) / (tempMax - tempMin);
                color[i][j] = colormap.color(t);
            }
        }

        Surface surface = new Surface(axis, x, y, co, color);
        surface.setSurfaceType(Surface.SURFACE_TYPE_NICEST);
        int nTicks = 10;
        double ticks[] = new double[nTicks];
        for (int i = 0; i < nTicks; i++) {
            ticks[i] = tempMax - i * (tempMax - tempMin) / (nTicks - 1);
        }
        ColormapLegend colormapLegend
                = new ColormapLegend(chart, colormap, ticks);
        colormapLegend.setPosition(-1, 10);
        colormapLegend.setTitle("Temperature");

        setSize(375, 375);
        render();
    }

    static private double[][] readData(String name)
            throws IOException, SQLException {
        InputStream is = SurfaceEx2.class.getResourceAsStream(name);
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        FlatFile ff = new FlatFile(br);
        double data[][] = new double[xMax][yMax];
        for (int j = 0; j < yMax; j++) {
            if (!ff.next()) {
                throw new IOException("Error in file " + name);
            }
            for (int i = 0; i < xMax; i++) {
                data[i][j] = ff.getDouble(i + 1);
            }
        }
        is.close();
        return data;
    }

    public static void main(String args[]) throws IOException, SQLException {
        new SurfaceEx2().setVisible(true);
    }
}
```

Output



Surface.ZFunction interface

```
public interface com.imsl.chart3d.Surface.ZFunction
```

Functional representation of a surface.

Method

f

`public double f(double x, double y)`

Description

Define the surface function.

Parameters

x – is the *x* value.

y – is the *y* value.

Returns

the *z* value.

Data class

`public class com.imsl.chart3d.Data extends com.imsl.chart3d.ChartNode3D implements Serializable`

Draws a 3D 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 `com.imsl.chart3d.ChartNode3D.DATA_TYPE_MARKER` (p. 1842) bit is set, the marker attributes are active. If the `com.imsl.chart3d.ChartNode3D.DATA_TYPE_LINE` (p. 1842) bit is set, the points are connected by lines using the line attributes. If the `com.imsl.chart3d.ChartNode3D.DATA_TYPE_TUBE` (p. 1842) bit is set, the points are connected by tubes using the line attributes. Tubes are similar to lines, but are fully 3d objects and so can be shaded.

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.

Constructors

Data

`public Data(AxisXYZ parent)`

Description

Creates a data node.

Parameter

`parent` – the `AxisXYZ` parent of this data node

Data

```
public Data(AxisXYZ parent, double[] x, double[] y, double[] z)
```

Description

Creates a data node with x, y and z values.

Parameters

`parent` – the `AxisXYZ` 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

`z` – a `double` array which contains the value for the attribute “Z” in this node

Methods

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

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. The argument `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.

Parameter

`range` – a `double` array which contains the updated range, {xmin,xmax,ymin,ymax}

getCustomMarkerFactory

```
public Data.CustomMarkerFactory getCustomMarkerFactory()
```

Description

Returns a custom marker factory.

setCustomMarker

```
public void setCustomMarker(Data.CustomMarkerFactory customMarkerFactory)
```

Description

Sets a custom marker factory. This factory is used when the "MarkerType" attribute is set to MARKER_TYPE_CUSTOM.

update

```
public void update()
```

Description

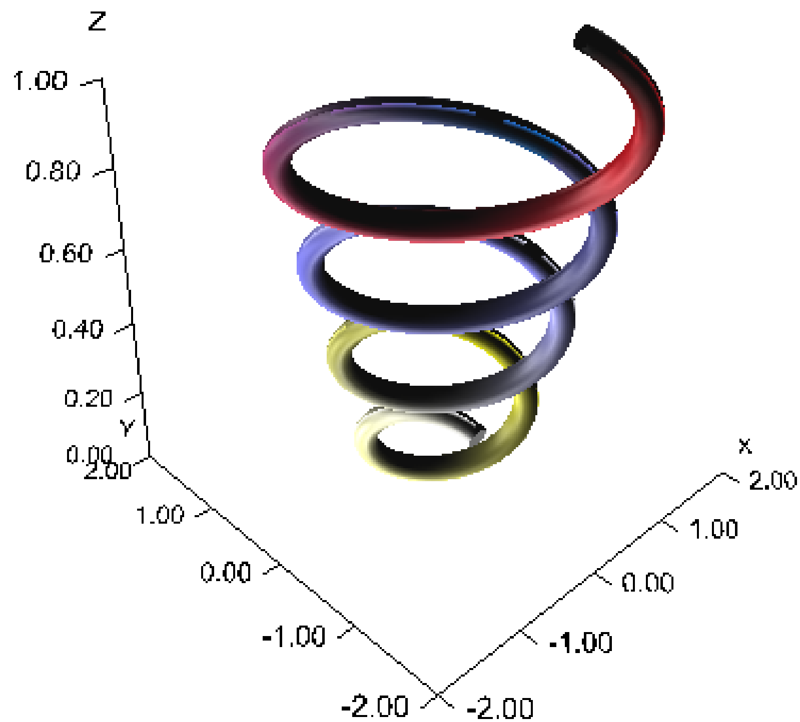
Update the surface by reevaluation of the z-function and the color function.

Example: Spiral Data connected with Tubes

A spiral data set is charted with tubes connecting the data points.

```
import com.imsl.chart3d.*;
import com.imsl.chart3d.ColorFunction;
import java.awt.Color;

public class DataEx1 extends JFrameChart3D implements ColorFunction {

    public DataEx1() {
        Chart3D chart = getChart3D();
        AxisXYZ axis = new AxisXYZ(chart);

        axis.getAxisBox().setPaint(false);

        int nSpiral = 400;
        double xSpiral[] = new double[nSpiral];
        double ySpiral[] = new double[nSpiral];
        double zSpiral[] = new double[nSpiral];
        for (int i = 0; i < nSpiral; i++) {
            double t = 8.0 * Math.PI * i / (double) (nSpiral - 1);
            double r = 0.6 + (double) i / (double) (nSpiral - 1);
            xSpiral[i] = r * Math.cos(t);
            ySpiral[i] = r * Math.sin(t);
            zSpiral[i] = (double) i / (double) (nSpiral - 1);
        }
        Data spiral = new Data(axis, xSpiral, ySpiral, zSpiral);
        spiral.setDataType(Data.DATA_TYPE_TUBE);
        spiral.setLineWidth(2);
        spiral.setColorFunction(this);
        this.setSize(375, 375);
        render();
    }

    public Color color(double x, double y, double z) {
        return com.imsl.chart.Colormap.SPECTRAL.color(z);
    }

    public static void main(String args[]) {
        new DataEx1().setVisible(true);
    }
}
```

Output



Example: Fisher Iris Data marked by spheres

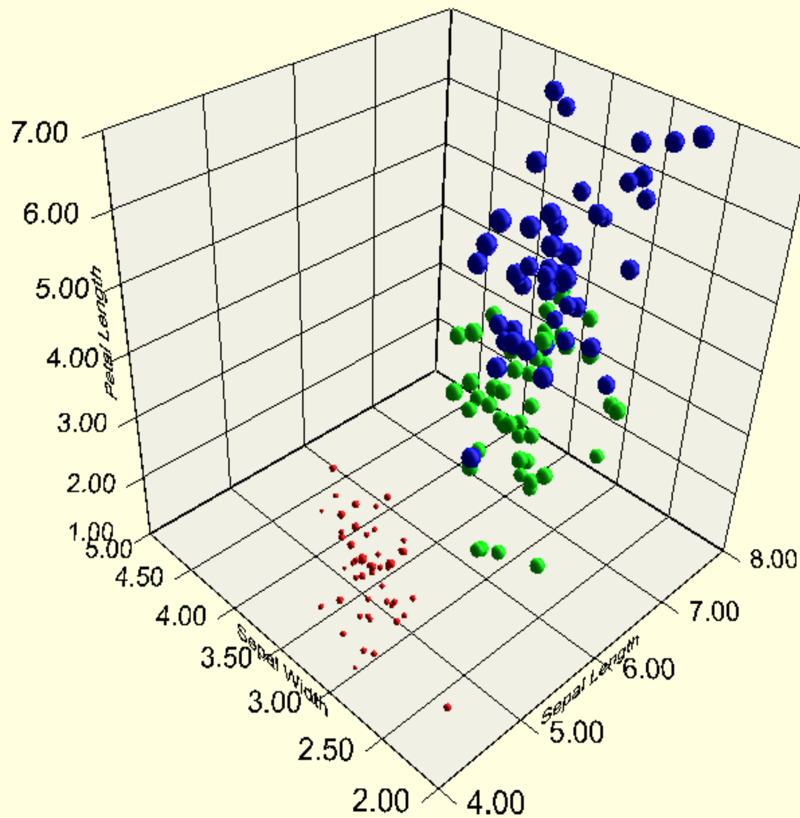
The classic Fisher iris data is plotted in this chart. This example shows use of the 3D marker type. A sphere has been chosen in this example to highlight data points.

```
import com.imsl.chart3d.*;
import com.imsl.io.FlatFile;
import java.awt.Color;
import java.io.*;
import java.sql.SQLException;
import java.util.StringTokenizer;

public class DataEx2 extends JFrameChart3D {
```

```
    private int species[];
    private double sepalLength[];
    private double sepalWidth[];
    private double petalLength[];
    private double petalWidth[];

    public DataEx2() throws IOException, SQLException {
        read();

        Chart3D chart = getChart3D();
        chart.getBackground().setFillColor("lightyellow");
        AxisXYZ axis = new AxisXYZ(chart);
        axis.setAxisTitlePosition(AxisXYZ.AXIS_TITLE_PARALLEL);

        axis.getAxisX().getAxisTitle().setTitle("Sepal Length");
        axis.getAxisY().getAxisTitle().setTitle("Sepal Width");
        axis.getAxisZ().getAxisTitle().setTitle("Petal Length");

        axis.setDataType(Data.DATA_TYPE_MARKER);
        axis.setMarkerType(Data.MARKER_TYPE_SPHERE);
        Color color[] = {java.awt.Color.red, java.awt.Color.green,
            java.awt.Color.blue};

        for (int k = 0; k < species.length; k++) {
            // marker type = Species
            // x = Sepal Length
            // y = Sepal Width
            // z = Petal Length
            // marker size = Petal Width
            double xp[] = {sepalLength[k]};
            double yp[] = {sepalWidth[k]};
            double zp[] = {petalLength[k]};
            Data data = new Data(axis, xp, yp, zp);
            data.setMarkerSize(Math.sqrt(petalWidth[k]));
            data.setMarkerColor(color[species[k] - 1]);
        }
        setSize(375, 375);
        render();
    }

    private void read() throws IOException, SQLException {
        InputStream is = getClass().getResourceAsStream("FisherIris.csv");
        FisherIrisReader fisherIrisReader = new FisherIrisReader(is);
        int nObs = 150;
        species = new int[nObs];
        sepalLength = new double[nObs];
        sepalWidth = new double[nObs];
        petalLength = new double[nObs];
        petalWidth = new double[nObs];
        for (int k = 0; fisherIrisReader.next(); k++) {
            species[k] = fisherIrisReader.getInt("Species");
            sepalLength[k] = fisherIrisReader.getDouble("Sepal Length");
            sepalWidth[k] = fisherIrisReader.getDouble("Sepal Width");
            petalLength[k] = fisherIrisReader.getDouble("Petal Length");
            petalWidth[k] = fisherIrisReader.getDouble("Petal Width");
```

```
        }
    }

    static private class FisherIrisReader extends FlatFile {

        public FisherIrisReader(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 IOException, SQLException {
        new DataEx2().setVisible(true);
    }
}
```

Output



Example: Heart Data

A multivariate data set is charted. This example shows how multiple variables can be encoded into a single chart. The data set is from: Afifi, A.A. and S.P. Azen (1979), *Statistical Analysis: A Computer Oriented Approach*, Second Edition, Academic Press, New York.

Each observation in the data set is represented by a marker. The encoding of the variables into the chart is described in the following table:

Representation	Variable
x-coordinate	Age (years)
y-coordinate	Height (cm)
z-coordinate	Initial Body Surface Area (m$\hat{2}$)
Marker Type	Survival?
Marker Size	Initial Mean Circulation Time (sec)
Color	Initial Cardiac Index (liters/min-m$\hat{2}$)
Min Pulse Size	Initial Hemoglobin (gm/100 ml)
Max Pulse Size	Final Hemoglobin (gm/100 ml)
Rotation Direction	Sex (Gender)

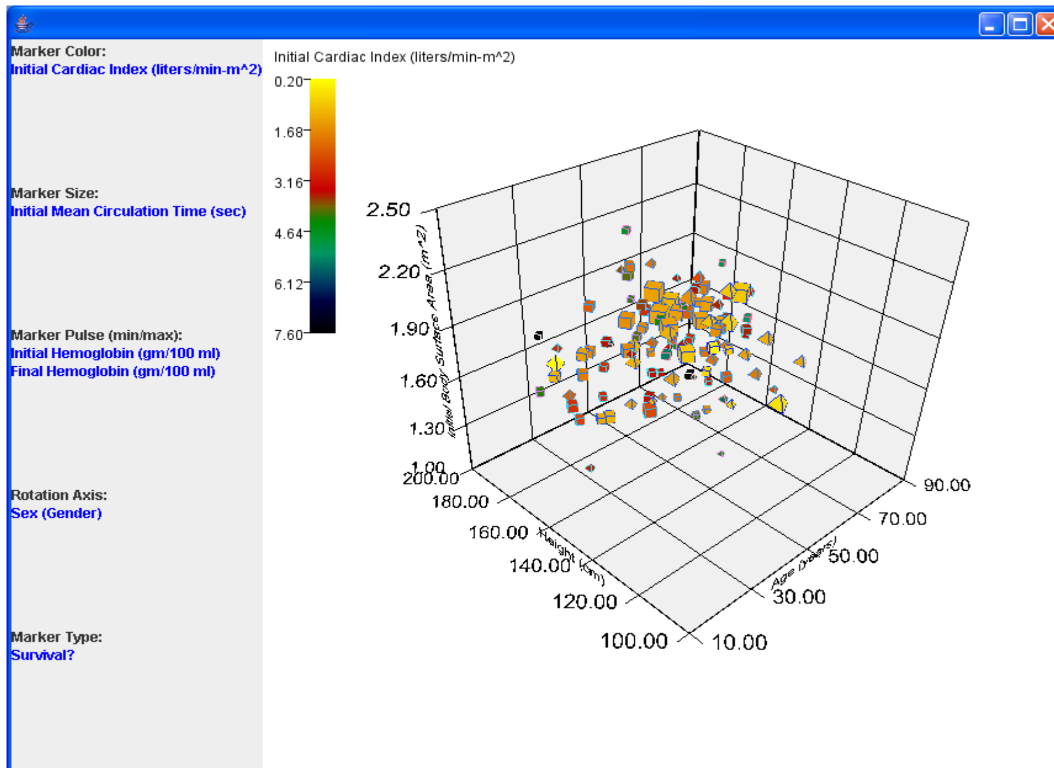
```
import com.imsl.chart.Colormap;
import com.imsl.chart3d.*;
import com.imsl.io.*;
import com.imsl.stat.Summary;
import java.awt.Color;
import java.io.*;
import java.sql.*;
import java.util.StringTokenizer;
import javax.swing.*;

public class DataEx3 extends javax.swing.JFrame {

    static private final int nVariables = 34;
    static private final int nObs = 113;

    static private final Colormap colormap = Colormap.BLUE_GREEN_RED_YELLOW;

    // Age (years)
    static private final int ivarX = 1;

    // Height (cm)
    static private final int ivarY = 2;

    // Initial Body Surface Area (m^2)
    static private final int ivarZ = 11;

    // Survival?
    static private final int ivarMarkerType = 4;

    // Initial Mean Circulation Time (sec)
    static private final int ivarMarkerSize = 14;

    // Initial Cardiac Index (liters/min-m^2)
    static private final int ivarMarkerColor = 12;

    // Initial Hemoglobin (gm/100 ml)
    static private final int ivarMarkerPulseMin = 18;

    // Final Hemoglobin (gm/100 ml)
    static private final int ivarMarkerPulseMax = 32;

    // Sex (Gender)
```

```
static private final int ivarRotationAxis = 3;

private ResultSetMetaData meta;
private JPanel jPanelLegend;

public DataEx3() throws IOException, SQLException {
    InputStream is = DataEx3.class.getResourceAsStream("AfifiAzen.csv");
    AfifiAzenReader reader = new AfifiAzenReader(is);
    double data[][] = reader.readData();
    is.close();

    Chart3D chart = new Chart3D();
    AxisXYZ axis = new AxisXYZ(chart);
    axis.setAxisTitlePosition(AxisXYZ.AXIS_TITLE_PARALLEL);

    meta = reader.getMetaData();
    axis.getAxisX().getAxisTitle().setTitle(meta.getColumnName(ivarX + 1));
    axis.getAxisY().getAxisTitle().setTitle(meta.getColumnName(ivarY + 1));
    axis.getAxisZ().getAxisTitle().setTitle(meta.getColumnName(ivarZ + 1));

    int markerTypes[]
            = {Data.MARKER_TYPE_CUBE, Data.MARKER_TYPE_TETRAHEDRON};

    double minMarkerSize = 0.0;
    double maxMarkerSize = 0.0;
    if (ivarMarkerSize >= 0) {
        Summary summary = getSummary(data, ivarMarkerSize);
        minMarkerSize = summary.getMinimum();
        maxMarkerSize = summary.getMaximum();
    }

    double minColor = 0.0;
    double maxColor = 0.0;
    if (ivarMarkerColor >= 0) {
        Summary summary = getSummary(data, ivarMarkerColor);
        minColor = summary.getMinimum();
        maxColor = summary.getMaximum();
    }

    double maxPulse = 0.0;
    if (ivarMarkerColor >= 0) {
        maxPulse = getSummary(data, ivarMarkerPulseMax).getMaximum();
    }

    axis.setDataType(Data.DATA_TYPE_MARKER);
    for (int i = 0; i < data.length; i++) {
        double xp[] = {data[i][ivarX]};
        double yp[] = {data[i][ivarY]};
        double zp[] = {data[i][ivarZ]};
        Data data3D = new Data(axis, xp, yp, zp);
        double size = (data[i][ivarMarkerSize] - minMarkerSize)
                / (maxMarkerSize - minMarkerSize);
        data3D.setMarkerSize(1.0 + size);
        double t = (data[i][ivarMarkerColor] - minColor)
                / (maxColor - minColor);
        data3D.setMarkerColor(colormap.color(t));
```

```
            data3D.setMarkerPulsingMinimumScale(data[i][ivarMarkerPulseMin]
                    / maxPulse);
            data3D.setMarkerPulsingMaximumScale(data[i][ivarMarkerPulseMax]
                    / maxPulse);
            data3D.setMarkerPulsingCycle(1.0);

            double zaxis = (data[i][ivarRotationAxis] == 1 ? 1.0 : -1.0);
            data3D.setMarkerRotatingAxis(0.0, 0.0, zaxis);
            data3D.setMarkerRotatingCycle(8.0);

            data3D.setMarkerType(data[i][ivarMarkerType]
                    == 1.0 ? markerTypes[0] : markerTypes[1]);
        }

        Canvas3DChart canvas = new Canvas3DChart(chart);
        canvas.setSize(375, 375);
        getContentPane().add(canvas, java.awt.BorderLayout.CENTER);

        jPanelLegend = new JPanel(new java.awt.GridBagLayout());
        setupLegend();
        getContentPane().add(jPanelLegend, java.awt.BorderLayout.WEST);
        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        pack();

        ColormapLegend colormapLegend
                = new ColormapLegend(chart, colormap, minColor, maxColor);
        colormapLegend.setPosition(10, 10);
        colormapLegend.setTitle(meta.getColumnName(ivarMarkerColor + 1));

        canvas.render();
    }

    private class AfifiAzenReader extends FlatFile {

        AfifiAzenReader(InputStream is) throws IOException {
            super(new BufferedReader(new InputStreamReader(is)));
            String line = readLine();
            line = readLine();
            line = readLine();
            StringTokenizer st = new StringTokenizer(line, ",");
            for (int j = 0; st.hasMoreTokens(); j++) {
                setColumnName(j + 1, st.nextToken().trim());
                setColumnClass(j, Double.class);
            }
        }

        double[][] readData() throws IOException, java.sql.SQLException {
            double data[][] = new double[nObs][nVariables];
            for (int i = 0; i < nObs; i++) {
                if (!next()) {
                    throw new IOException("Error in file");
                }
                for (int j = 0; j < nVariables; j++) {
                    data[i][j] = getDouble(j + 1);
                }
```

```
            }
            return data;
        }
    }

    static Summary getSummary(double data[][], int ivar) {
        Summary summary = new Summary();
        for (int i = 0; i < nObs; i++) {
            summary.update(data[i][ivar]);
        }
        return summary;
    }

    private void setupLegend() throws SQLException {
        addLegendTitle("Marker Color:");
        addLegendValue(ivarMarkerColor, 1.0);

        addLegendTitle("Marker Size:");
        addLegendValue(ivarMarkerSize, 1.0);

        addLegendTitle("Marker Pulse (min/max):");
        addLegendValue(ivarMarkerPulseMin, 0.0);
        addLegendValue(ivarMarkerPulseMax, 1.0);

        addLegendTitle("Rotation Axis:");
        addLegendValue(ivarRotationAxis, 1.0);

        addLegendTitle("Marker Type:");
        addLegendValue(ivarMarkerType, 1.0);
    }

    private void addLegendTitle(String title) {
        JLabel jLabel = new JLabel(title);
        java.awt.GridBagConstraints gridBagConstraints
                = new java.awt.GridBagConstraints();
        gridBagConstraints.gridwidth = java.awt.GridBagConstraints.REMAINDER;
        gridBagConstraints.anchor = java.awt.GridBagConstraints.WEST;
        jPanelLegend.add(jLabel, gridBagConstraints);
    }

    private void addLegendValue(int ivar, double weight) throws SQLException {
        JLabel jLabel = new JLabel(meta.getColumnName(ivar + 1));
        java.awt.GridBagConstraints gridBagConstraints
                = new java.awt.GridBagConstraints();
        gridBagConstraints.gridwidth = java.awt.GridBagConstraints.REMAINDER;
        gridBagConstraints.anchor = java.awt.GridBagConstraints.NORTHWEST;
        gridBagConstraints.weighty = weight;
        jPanelLegend.add(jLabel, gridBagConstraints);
        jLabel.setForeground(Color.BLUE);
    }

    public static void main(String args[])
            throws IOException, java.sql.SQLException {
        new DataEx3().setVisible(true);
    }
}
```

Output



Data.CustomMarkerFactory interface

```
public interface com.imsl.chart3d.Data.CustomMarkerFactory
```

Factory to create customized markers.

Method

createCustomMarker

```
public Node createCustomMarker()
```

Description

Returns a custom marker.

ColorFunction interface

```
public interface com.imsl.chart3d.ColorFunction
```

Interface to define value dependent colors.

Method

color

```
public Color color(double x, double y, double z)
```

ColormapLegend class

```
public class com.imsl.chart3d.ColormapLegend extends
com.imsl.chart3d.ChartNode3D implements Serializable
```

Adds a legend for a Colormap gradient to the background of the canvas.

Constructors

ColormapLegend

```
public ColormapLegend(Chart3D chart, Colormap colormap, double[] ticks)
```

Description

Creates a legend for a Colormap and adds it to the canvas. If set, the attribute “Title” is used to provide a title for the legend.

The `paint` method in `Canvas3DChart.Paint` is written into an image of size `width` by `height`. Any whitespace around the image is trimmed. The trimmed image is then used to paint onto the canvas.

Parameters

`chart` – is the `Chart3D` object on which the legend is to be painted.

`colormap` – is the `Colormap` for the legend.

`ticks` – is an array of values used to label the legend. These should be equally spaced.

ColormapLegend

```
public ColormapLegend(Chart3D chart, Colormap colormap, double min, double max)
```

Methods

addToSceneGraph

```
protected void addToSceneGraph(Group parent)
```

getPosition

```
public int[] getPosition()
```

Description

Returns the position of the legend.

Returns

an array containing the legend's position.

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.

getWindow

```
public double[] getWindow()
```

Description

Returns the window for a ColormapLegend.

Returns

window an array of length two containing the range of this colormap.

remove

```
public void remove()
```

Description

Removes the node from its parents list of children.

setPosition

```
public void setPosition(int x, int y)
```

Description

Sets the position of the legend. The default position is (0,0).

Parameters

`x` – is the pixel position in the canvas of the left edge of the legend. If *x* is negative then |x| is the distance from the right edge of the legend to the right edge of the component.

`y` – is the pixel position in the canvas of the top edge of the legend. If *y* is negative then |y| is the distance from the bottom edge of the legend to the bottom edge of the component.

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.

Parameter

`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.

setWindow

```
public void setWindow(double[] window)
```

Description

Sets the window for a ColormapLegend.

Parameter

`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 a ColormapLegend.

Parameters

`min` – is the value of the bottom end of the colormap legend labels.

`max` – is the value of the top of the colormap legend labels.

Chapter 29: Data Mining

Types

class NaiveBayesClassifier 1902
class Itemsets 1923
class AssociationRule 1925
class Apriori 1926
class KohonenSOM 1934
class KohonenSOMTrainer 1942
class PredictiveModel 1945
class BootstrapAggregation 1962
class CrossValidation 1969
class GradientBoosting 1976

Usage Notes

Data Mining - An Overview

Data mining refers to the process of using statistical and analytical methods to extract useful information from large databases. The problem of extracting information from data is prevalent everywhere, in government, business, education, industry, engineering and sciences. The methods and algorithms used in data mining have been developed and invented, with considerable overlap, within fields known as *machine learning*, *statistical learning*, and statistics. While there may be theoretical or philosophical differences between the fields of study, these nuances are not important from a practical standpoint. They all have the same goal: learning from data.

Data Types

In general, data fall into two major categories: continuous and categorical. A continuous variable can assume any real number within a certain range. Examples of continuous variables include temperature, height, weight, circumference, body mass index, rate of return, etc. Count data are often treated as continuous variables in data mining algorithms. Even though they only assume discrete values, their set of possible values is infinite. Examples of count data include the number of accidents per year, the number of units sold, the number of insurance claims, and so on.

Categorical variables take on values from a finite list of categories. There are two types of categorical data: ordinal and nominal. Ordinal data have a natural ordering among the categories, such as a school grade. Nominal data are categories without a natural ordering, such as eye color.

Sometimes continuous variables are binned into a finite set, for example, income level: {less than 25K, between 25K and 50K, over 50K}, or body weight: {underweight, normal, overweight, obese}. For modeling purposes, these are often treated as ordinal categorical, especially when, for whatever reason, the raw observations on the subjects are no longer available.

Other types of data deserve a special mention: transaction or invoice data and text data. A transaction (think of a grocery store receipt) has attributes such as date and time, total amount, the set of products that were purchased, their quantities and prices, and possibly attributes of the individual customer making the purchases. Text data is discrete and not ordered, but the association of text or words in sentences, forming context, makes text data an important subtype for data mining applications.

Data Mining Problem Types

The primary types of data mining problems are *pattern recognition* and *prediction*. Prediction includes the subtypes *classification*, *regression*, and *forecasting*.

Pattern recognition algorithms are designed to detect patterns in large, high-dimensional, and complex data sets. Pattern recognition problems fall under two broad categories: supervised and unsupervised. In supervised problems, the number of groups or categories is known and each example (observation) in the training data has a known outcome (or response). The set of attributes or predictor variables measured on each example may relate to the response variable, and may then be used to predict the outcomes of future or new examples. Supervised learning algorithms try to detect the relationship between the set of attributes and the outcome of the response variable.

Prediction problems are supervised problems concerned with predicting an outcome of a variable using known attributes as inputs into a statistical model. In prediction problems, there is a single variable of interest called a *dependent* variable, or a *response* variable, or sometimes a *label* when the variable is categorical. The set of attributes consists of other variables that may have some relationship with the variable of interest. These variables are variously referred to as *independent*, *explanatory*, *predictor* variables, *attributes*, or *features*.

Classification is a prediction problem in which the response is categorical; regression is a prediction problem in which the response is continuous; and forecasting is a prediction problem in which the response variable and predictor variables are indexed by time. Most algorithms in this chapter have methods for either classification or regression. For time series, neural networks and support vector machines have both been used successfully.

In unsupervised problems, there is no known outcome or response. Each example in the training data is a vector of measurements on a number (often a very large number) of variables. The problem is to detect any patterns or structure that might exist in the high-dimensional space spanned by the variables. With a smaller set of natural groupings (clusters) or structures in lower dimensions, stronger inferences can be made about the population or the distribution of the variables. In this chapter, the Kohonen self-organizing map (`KohonenSOM`) is an example of an unsupervised learning algorithm. Many of the algorithms described in the Multivariate Analysis chapter are unsupervised learning algorithms.

For any data mining model to be useful, it must be given data it can learn from. This data has examples of known values of predictor variables and known value of the dependent variable. This data is called

training data. Once the model is *trained* or *fitted* to the training data, new examples (ones not used to train the model) are run through the model to obtain an estimated (predicted) value of the dependent variable. If the true value of the dependent variable is known, the predicted value is compared to the known value and a prediction error can be recorded. New examples with known (or realized) values of the dependent variable comprise what is often called a *test* data set, which is used to evaluate how accurately the model predicts the dependent variable. For the purpose of such model assessment, a complete data set can be randomly partitioned into a training set and a test set. See below the summary of *cross-validation*.

Data Filtering and Scaling

Regardless of the type or the source of the data, it must be filtered from its raw form into formats required by data mining and analytical algorithms. Categorical data must be mapped into a corresponding numerical representation. Some algorithms, for example `DecisionTrees`, treat categorical data appropriately, while other algorithms interpret all data as continuous. In these cases, categorical variables must be transformed.

Dummy variables are used to indicate the distinct values of categorical variables without implying order. To illustrate, suppose in a survey of consumer preferences, the categorical variable x indicates the type of chocolate, {White, Milk, Dark} and that these values are encoded to {1, 2, 3} in the data. If x is used directly, the algorithm presumes a scale ($1 < 2 < 3$) that is not really true or meaningful, and will lead to invalid inferences about the relationships between the variables. The reason for creating dummy variables is to represent a categorical variable with a set of indicators that puts each distinct value on the same numerical level. In the table, the binary encoding of x into 3 dummy variables is shown.

Chocolate	x-value	dummy1	dummy2	dummy3
White	1	1	0	0
Milk	2	0	1	0
Dark	3	0	0	1

The class `UnsupervisedNominalFilter` uses binary encoding to map nominal data into a matrix of zeros and ones. The resulting columns are often referred to as *dummy* variables.

If the categorical variable is ordinal, so that order between the levels is meaningful, the class `UnsupervisedOrdinalFilter` encodes and decodes ordinal data into the range [0, 1] using cumulative percentages.

Continuous data may need to be scaled. Many algorithms, such as neural networks and support vector machines, perform better if continuous data are mapped into a common scale. Class `ScaleFilter` implements several techniques for automatically scaling continuous data, including several variations of z-score scaling. If the continuous data represent a time series, `TimeSeriesFilter` and `TimeSeriesClassFilter` can be used to create a matrix of lagged values required as input to neural networks. More details on these methods can be found in the `Neural Networks` chapter.

Apriori Market Basket Analysis

Market basket analysis is an unsupervised data mining problem for detecting strong associations between products or items in transactional data. The problem arose especially with the advent of digital scanners and scanner data in supermarkets, hence the name “market basket”. The class `Apriori` performs the Apriori algorithm for finding strong association rules. Association rules are statements of

the form, “if X, then Y”, given with some measure of confidence. For example, in a supermarket X and Y are different products such as bread and butter. Learning which products are strongly associated helps managers make more profitable marketing decisions, such as product placement or sales promotions. There are other applications for association rule discovery, such as in text mining and bioinformatics.

Kohonen Self-organizing Map

A self-organizing map (SOM), also known as a Kohonen map or Kohonen SOM, is a technique for gathering high-dimensional data into clusters that are constrained to lie in low dimensional space, usually two dimensions. It is a widely used technique for the purpose of feature extraction and visualization for very high dimensional data. The Kohonen SOM is equivalent to a neural network having inputs linked to every node in the network. The creation of a Kohonen map involves two steps: training and forecasting. Training builds the map using input examples, and forecasting classifies new input.

Naive Bayes

Naive Bayes is a classification algorithm. A classifier is trained using `NaiveBayesClassifier` with known classifications. Once the classifier is trained, the classifier can be used to classify patterns with unknown classifications through `NaiveBayesClassifier` methods `predictClass` and `probabilities`.

Classification problems can be solved using other algorithms such as discriminant analysis and neural networks. In general, these alternatives have smaller classification error rates, but they are too slow for large classification problems. During training, `NaiveBayesClassifier` uses the non-missing training data to estimate two-way correlations among the attributes. Higher order correlations are assumed to be zero. This can increase the classification error rate, but it significantly reduces the time needed to train the classifier.

Neural Networks

An artificial neural network, or neural network, is a very flexible modeling framework that can be used in many of the applications and problem areas in data mining. The elements of a neural network (terminology inspired by the biological brain) are nodes and layers and activation functions. There are many options for setting up a network. Once the architecture of the network is specified, it can be trained on the training data and evaluated on test data, similar to other supervised learning algorithms. For more details, see the `Neural Networks` chapter.

Predictive Models

Class `PredictiveModel` is the abstract base class for predictive models like decision trees. It contains methods and members common to different predictive models in regression or classification problems. Users can leverage the abstract class and its methods to create customized predictive models. Presently, JMSL includes two major packages that extend `PredictiveModel`: `DecisionTrees` and `SupportVectorMachine`.

Decision Trees

Decision trees are predictive models for classification or regression. They are so named because they mimic a branching decision making process. The outcome of a decision tree algorithm is a set of decision rules, making them often preferable to other algorithms which are difficult to interpret, even though they may be accurate for a given problem. The `DecisionTrees` chapter includes 4 specific

algorithms, ALACART, C45, CHAID, and QUEST, for generating a decision tree. For more details and examples, see the DecisionTrees chapter.

Support Vector Machines

Support Vector Machines (SVMs) are a widely used machine learning method for regression, classification and other learning tasks. Class SVRegression implements the ν- and ε-SVMs for regression, ν-SVR and ε-SVR. Class SVClassification contains implementations of the C- and ν-SVMs for classification, C-SVC and ν-SVC. Class SVOneClass estimates the support of a high-dimensional distribution, using the One-class SVM. All three classes are subclasses of class SupportVectorMachine. This parent class contains methods which can be used by all SVM formulations. Furthermore, all SVMs can use different kernels for optimization and the calculation of predictions. The kernels are implemented in classes LinearKernel,PolynomialKernel, RadialBasisKernel and SigmoidKernel. More information on the available SVMs and kernels can be found in the Support Vector Machines chapter.

Cross-Validation

Cross-validation is an important resampling method that can be used for model assessment or model selection. Class CrossValidation performs *k*-fold cross-validation on predictive models, like decision trees or support vector machines. In *k*-fold cross-validation, the set of observations is randomly split into *k* disjoint folds of approximately equal size. The first fold is then treated as a test set, and the model trained on the remaining *k*-1 folds.This procedure is repeated *k* times, with each of the folds serving once as a test set. Applying the fitted models to their test sets results in *k* estimates for the test error. The cross-validated error is computed by averaging these values. For classification problems, stratified cross-validation can be performed via the setStratifiedCrossValidation method.

Ensemble Methods

An ensemble method involves fitting a collection of predictive models and combining their collective outputs. The approach helps reduce variability and overfitting and improves predictive accuracy. In particular, decision trees have been shown to dramatically improve when used in ensembles.

Bootstrap Aggregation

Bootstrap aggregation (bagging) is an important statistical tool that helps to improve the accuracy of predictions from predictive models, like decision trees. Given a specific predictive model and a single training data set of size *N*, class BootstrapAggregation performs bagging by taking repeated bootstrap samples of size *N* from the training data set. The samples are drawn randomly with replacement from the training set so that an observation can occur more than once in the bootstrap data set. The predictive model is then trained on each bootstrap sample separately, and predictions are generated. The predictions are finally combined into a single value by averaging (for regression problems) or majority vote (for classification problems).

Gradient Boosting

Like bagging, gradient boosting is another approach to iteratively improve the predictions from a predictive model, specifically a decision tree. Class GradientBoosting implements a special form of gradient boosting, the stochastic gradient tree boosting algorithm of Friedman (1999). Class GradientBoosting can be applied to regression and classification problems. For classification problems, the binomial or multinomial deviance loss function must be used.

Random Decision Trees

Class `RandomTrees` implements the ensemble method called *random forest* (Breiman, 2001). A random forest is a collection of decision trees on bootstrap samples. In addition, the set of predictor variables is randomized before each branching or splitting decision within the decision tree algorithms. This extra randomization reduces correlation among the different trees in the ensemble. The class `RandomTrees` is in the `DecisionTrees` chapter.

NaiveBayesClassifier class

`public class com.imsl.datamining.NaiveBayesClassifier implements Serializable`

Trains a Naive Bayes Classifier

`NaiveBayesClassifier` trains a Naive Bayes classifier for classifying data into one of `nClasses` target classes. Input attributes can be a combination of both nominal and continuous data. Ordinal data can be treated as either nominal attributes or continuous. If the distribution of the ordinal data is known or can be approximated using one of the continuous distributions, then associating them with continuous attributes allows a user to specify that distribution. Missing values are allowed.

Before training the classifier the input attributes must be specified. For each nominal attribute, use method `createNominalAttribute` to specify the number of categories in each `nNominal` attribute. Specify the input attributes in the same column order that they will be supplied to the `train` method. For example, if the input attribute in the first two columns of the nominal input data, `nominalData`, represent the first two nominal attributes and have two and three categories respectively, then the first call to the `createNominalAttribute` method would specify two categories and the second call to `createNominalAttribute` would specify three categories.

Likewise, for each continuous attribute, the method `createContinuousAttribute` can be used to specify a `ProbabilityDistribution` other than the default `NormalDistribution`. A second `createContinuousAttribute` is provided to allow specification of a different distribution for each target class (see Example 3). Create each continuous attribute in the same column order they will be supplied to the `train` method. If `createContinuousAttribute` is not invoked for all `nContinuous` attributes, the `NormalDistribution` `ProbabilityDistribution` will be used. For example, if five continuous attributes have been specified in the constructor, but only three calls to `createContinuousAttribute` have been invoked, the last two attributes, or columns of `continuousData` in the `train` method, will use the `NormalDistribution` `ProbabilityDistribution`.

Nominal only, continuous only, and a combination of both nominal and continuous input attributes are allowed. The three `train` methods allow for a combination of input attribute types.

Let C be the classification attribute with target categories $0, 1, \ldots, \mathrm{nClasses} - 1$, and let $X = \{x_1, x_2, \ldots, x_k\}$ be a vector valued array of k=`nNominal`+`nContinuous` input attrtibutes, where `nNominal` is the number of nominal attributes and `nContinuous` is the number of continuous attributes. See methods `createNominalAttribute` to specify the number of categories for each nominal attribute

and `createContinuousAttribute` to specify the distribution for each continuous attribute. The classification problem simplifies to estimate the conditional probability *P(C|X)* from a set of training patterns. The Bayes rule states that this probability can be expressed as the ratio:

$$P(C=c|X=\{x_1,x_2,\ldots,x_k\}) = \frac{P(C=c)P(X=\{x_1,x_2,\ldots,x_k\})|C=c)}{P(X=\{x_1,x_2,x,\ldots,x_k\})}$$

where c is equal to one of the target classes $0,1,\ldots,\text{nClasses}-1$. In practice, the denominator of this expression is constant across all target classes since it is only a function of the given values of X. As a result, the Naive Bayes algorithm does not expend computational time estimating $P(X=\{x_1,x_2,x,\ldots,x_k\})$ for every pattern. Instead, a Naive Bayes classifier calculates the numerator $P(C=c)P(X=\{x_1,x_2,\ldots,x_k\})|C=c)$ for each target class and then classifies X to the target class with the largest value, i.e.,

$$X \xleftarrow[\max(c=0,1,\ldots,\text{nClasses}-1)]{} P(C=c)P(X|C=c)$$

The classifier simplifies this calculation by assuming conditional independence. That is it assumes that:

$$P(X=\{x_1,x_2,\ldots,x_k\}|C=c) = \prod_{j=1}^{k} P(x_j|C=c)$$

This is equivalent to assuming that the values of the input attributes, given C, are independent of one another, i.e.,

$$P(x_i|x_j,C=c) = P(x_i|C=c),\ \text{for all } i \neq j$$

In real world data this assumption rarely holds, yet in many cases this approach results in surprisingly low classification error rates. Since, the estimate of $P(C=c|X=\{x_1,x_2,\ldots,x_k\})$ from a Naive Bayes classifier is generally an approximation, classifying patterns based upon the Naive Bayes algorithm can have acceptably low classification error rates.

For nominal attributes, this implementation of the Naive Bayes classifier estimates conditional probabilities using a smoothed estimate:

$$P(x_j|C=c) = \frac{\#N\{x_j \cap C=c\}+\lambda}{\#N\{C=c\}+\lambda j},$$

where $\#N\{Z\}$ is the number of training patterns with attribute Z and j is equal to the number of categories associated with the j-th attribute.

The probability *P(C=c)* is also estimated using a smoothed estimate:

$$P(C=c) = \frac{\#N\{C=c\}+\lambda}{\text{nPatterns}+\lambda(\text{nClasses})}\ .$$

These estimates correspond to the maximum a priori (MAP) estimates for a Dirichelet prior assuming equal priors. The smoothing parameter can be any non-negative value. Setting $\lambda = 0$ corresponds to no smoothing. The default smoothing used in this algorithm, $\lambda = 1$, is commonly referred to as Laplace smoothing. This can be specified using the optional `setDiscreteSmoothingValue`.

For continuous attributes, the same conditional probability $P(x_j|C=c)$ in the Naive Bayes formula is replaced with the conditional probability density function $f(x_j|C=c)$. By default, the density function

for continuous attributes is the normal (Gaussian) probability density function (see NormalDistribution):

$$f(x_j|C=c) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}}$$

where μ and σ are the conditional mean and standard deviation, i.e. the mean and standard deviation of x_j when $C = c$. For convenience, methods getMeans and getStandardDeviations are provided to calculate the conditional mean and standard deviations of the training patterns.

In addition to the default normal pdf, users can select any continuous distribution to model the continuous attribute by providing an implementation of the com.imsl.stat.ProbabilityDistribution interface. See NormalDistribution, LogNormalDistribution, GammaDistribution, and PoissonDistribution for classes that implement the ProbabilityDistribution interface.

Smoothing conditional probability calculations for continuous attributes is controlled by the methods setContinuousSmoothingValue and setZeroCorrection. By default, conditional probability calculations for continuous attributes are unadjusted for calculations near zero. The value specified in the setContinuousSmoothingValue method will be added to each continuous probability calculation. This is similar to the effect of using setDiscreteSmoothingValue for the corresponding discrete calculations.

The value specified in the setZeroCorrection method is used when $(f(x|C=c)+\lambda)=0$, where λ is the smoothing parameter setting. If this condition occurs, the conditional probability is replaced with the value set in setZeroCorrection.

Methods getClassificationErrors, getPredictedClass, getProbabilities, and getTrainingErrors provide information on how well the trained NaiveBayesClassifier predicts the known target classifications of the training patterns.

Methods probabilities and predictClass estimate classification probabilities and predict classification of the input pattern using the trained Naive Bayes Classifier. The predicted classification returned by predictClass is the class with the largest estimated classification probability. Method classError predicts the classification from the trained Naive Bayes classifier and compares the predicted classifications with the known target classification provided. This allows verification of the classifier with a set of patterns other than the training patterns.

Constructor

NaiveBayesClassifier

```
public NaiveBayesClassifier(int nContinuous, int nNominal, int nClasses)
```

Description

Constructs a NaiveBayesClassifier

Parameters

nContinuous – an int containing the number of continuous attributes

`nNominal` – an `int` containing the number of nominal attributes

`nClasses` – an `int` containing the number of target classifications

Methods

classError

```
public double classError(double[] continuous, int[] nominal, int
classification)
```

Description

Returns the classification probability error for the input pattern and known target classification.

Parameters

`continuous` – a `double` array of length `nContinuous` containing an input pattern of continuous attributes. If `nContinuous` = 0, a `null` is allowed.

`nominal` – an `int` array of length `nNominal` containing an input pattern of nominal attributes. If `nNominal` = 0, a `null` is allowed.

`classification` – an `int` containing the target classification.

Returns

a `double` containing the classification probability error for the input pattern. The classification error for the input pattern is equal to 1-p, where p is the predicted class probability of input `classification`. The predicted class probability of input `classification` can be obtained by the method `probabilities`. If p = `probabilities` and k is equal to `classification`, then the classification error is 1 - p[k].

createContinuousAttribute

```
public void createContinuousAttribute(ProbabilityDistribution pdf)
```

Description

Create a continuous variable and the associated distribution function.

Parameter

`pdf` – a `ProbabiltyDistribution` to be applied to the continuous attribute. The distribution function will be applied to all classes. By default, `NormalDistribution` is used.

createContinuousAttribute

```
public void createContinuousAttribute(ProbabilityDistribution[] pdf)
```

Description

Create a continuous variable and the associated distribution functions for each target classification.

Parameter

`pdf` – an array of `ProbabilityDistributions` containing `nClasses` distribution functions for a continuous attribute. This allows a different distribution function to be applied to each classification. By default, `NormalDistribution` is used.

createNominalAttribute

`public void createNominalAttribute(int nCategories)`

Description

Create a nominal attribute and the number of categories

Parameter

`nCategories` – an `int` containing the number of categories in the nominal attribute. The category values are expected to be encoded with integers ranging from 0 to `nCategories` -1. No default is used for `nCategories`. If `nNominal` is not zero, and `createNominalAttribute` is not invoked for each `nNominal` attribute, an `IllegalStateException` will be thrown when the `train` method is invoked.

getClassCounts

`public int[] getClassCounts(int[] classificationData)`

Description

Returns the number of patterns for each target classification.

Parameter

`classificationData` – an `int` array containing the target classifications for the training patterns. These must be encoded from zero to `nClasses`-1. Any value outside this range is considered a missing value. In this case, the data in that pattern are not used to train the Naive Bayes classifier. However, any pattern with missing values is still classified after the classifier is trained.

Returns

an `int` array containing the class counts.

getClassificationErrors

`public double[] getClassificationErrors()`

Description

Returns the classification probability errors for each pattern in the training data.

Returns

a `double` array containing the classification probability errors for each pattern in the training data. The classification error for the *i*-th training pattern is equal to 1-*predictedClassProbability*`[i][k]`, where *predictedClassProbability* is returned from `getProbabilities` and *k* is equal to `classificationData[i]`.

getMeans

`public double[][] getMeans(double[][] continuousData, int[] classificationData)`

Description

Returns a table of means for each continuous attribute in `continuousData` segmented by the target classes in `classificationData`.

This method is provided as a utility, prior training is not necessary.

Parameters

`continuousData` – a `double` matrix containing training values for the continuous attributes.

`classificationData` – an `int` array containing the target classifications for the training patterns.

Returns

a `continuousData[0].length` by `nClasses` `double` matrix, *means*, containing the means segmented by the target classes. The *i*-th row contains the means of the *i*-th continuous attribute for each value of the target classification. That is, *means*[i][j] is the mean for the *i*-th continuous attribute when the target classification equals *j*, unless there are no training patterns for this condition.

getPredictedClass

```
public int[] getPredictedClass()
```

Description

Returns the predicted classification for each training pattern.

Returns

an `int` array containing the predicted classification for each training pattern.

getProbabilities

```
public double[][] getProbabilities()
```

Description

Returns the predicted classification probabilities for each target class.

Returns

a `double` matrix, *prob*, of size *nPatterns* by `nClasses` containing the predicted classification probabilities for each target class, where *nPatterns* is the number of patterns trained. *prob*[i][j] is the estimated probability that the *i*-th pattern belongs to the *j*-th target class.

getStandardDeviations

```
public double[][] getStandardDeviations(double[][] continuousData, int[]
classificationData)
```

Description

Returns a table of standard deviations for each continuous attribute in `continuousData` segmented by the target classes in `classificationData`.

This method is provided as a utility, prior training is not necessary.

Parameters

`continuousData` – a `double` matrix containing training values for the continuous attributes.

`classificationData` – an `int` array containing the target classifications for the training patterns.

Returns

a `continuousData[0].length` by `nClasses` `double` matrix, *stdev*, containing the standard deviations segmented by the target classes. The *i*-th row contains the standard deviation of the *i*-th continuous attribute for each value of the target classification. That is, *stdev*[i][j] is the standard deviations for the *i*-th continuous attribute when the target classification equals *j*, unless there are no training patterns for this condition.

getTrainingErrors

`public int[][] getTrainingErrors()`

Description

Returns a table of classification errors of non-missing classifications for each target classification plus the overall total of classification errors.

Returns

an `int` matrix containing `nClasses` + 1 rows and two columns. The first column contains the number of misclassifications and the second column contains the total number of classifications for the *i*-th row target class. The last row of the matrix contains the total number of misclassifications in column one and the total non-missing classifications in column two.

ignoreMissingValues

`public void ignoreMissingValues(boolean ignoreMissing)`

Description

Specifies whether or not missing values will be ignored during the training process.

Parameter

`ignoreMissing` – a `boolean` specifying whether or not to ignore patterns during training when one or more input attributes are missing. By default, both missing and non-missing values are used to train the classifier. Classification predictions are still returned for all patterns even when set to true. By default, `ignoreMissing` = `false`.

predictClass

`public int predictClass(double[] continuous, int[] nominal)`

Description

Predicts the classification for the input pattern using the trained Naive Bayes classifier.

Parameters

`continuous` – a `double` array containing an input pattern of `nContinuous` continuous attributes. If `nContinuous` = 0, a `null` is allowed.

`nominal` – an `int` array of length `nNominal` containing an input pattern of nominal attributes. If `nNominal` = 0, a `null` is allowed.

Returns

an `int` containing the predicted classification for the input pattern using the trained Naive Bayes Classifier. The predicted classification returned is the class with the largest estimated classification probability. The classification probabilities can be predicted using the `probabilities` method.

probabilities

```
public double[] probabilities(double[] continuous, int[] nominal)
```

Description

Predicts the classification probabilities for the input pattern using the trained Naive Bayes classifier.

Parameters

`continuous` – a `double` array containing an input pattern of `nContinuous` continuous attributes. If `nContinuous` = 0, a `null` is allowed.

`nominal` – an `int` array of length `nNominal` containing an input pattern of nominal attributes. If `nNominal` = 0, a `null` is allowed.

Returns

a `double` array of length `nClasses` containing the predicted classification probabilities for each target class.

setContinuousSmoothingValue

```
public void setContinuousSmoothingValue(double clambda)
```

Description

Parameter for calculating smoothed estimates of conditional probabilities for continuous attributes.

Parameter

`clambda` – a `double` containing the smoothing parameter to be used for calculating smoothed estimates of conditional probabilities for continuous attributes. `clambda` must be non-negative. By default, `clambda=0`, i.e. no smoothing is done.

setDiscreteSmoothingValue

```
public void setDiscreteSmoothingValue(double dlambda)
```

Description

Parameter for calculating smoothed estimates of conditional probabilities for discrete (nominal) attributes.

Parameter

`dlambda` – a `double` containing the smoothing parameter to be used for calculating smoothed estimates of conditional probabilities for discrete attributes. `dlambda` must be non-negative. By default, `dlambda` = 1.0, i.e. Laplace smoothing of conditional probabilities.

setZeroCorrection

```
public void setZeroCorrection(double zeroCorrection)
```

Description

Specifies the replacement value to be used for conditional probabilities equal to zero.

Parameter

`zeroCorrection` – a `double` containing the value to replace conditional probabilities equal to zero. `zeroCorrection` must be non-negative. By default, no correction will be performed.

---

train

```
public void train(double[][] continuousData, int[] classificationData)
```

Description

Trains a Naive Bayes classifier for classifying data into one of `nClasses` target classifications.

Parameters

`continuousData` – a `double` matrix containing the training values for the `nContinuous` continuous attributes. The *i*-th row contains the input attributes for the *i*-th training pattern. The *j*-th column contains the values for the *j*-th continuous attribute. Missing values should be set to `Double.NaN`. Patterns with both non-missing and missing values are used to train the classifier unless the `ignoreMissingValues` method has been set to `true`.

`classificationData` – an `int` array containing the target classifications for the training patterns. These must be encoded from zero to `nClasses`-1. Any value outside this range is considered a missing value. In this case, the data in that pattern are not used to train the Naive Bayes classifier. However, any pattern with missing values is still classified after the classifier is trained.

---

train

```
public void train(int[][] nominalData, int[] classificationData)
```

Description

Trains a Naive Bayes classifier for classifying data into one of `nClasses` target classifications.

Parameters

`nominalData` – an `int` matrix containing the training values for the `nNominal` nominal attributes. The *i*-th row contains the input attributes for the *i*-th training pattern. The *j*-th column contains the classifications for the *j*-th nominal attribute. The values for the *j*-th nominal attribute are expected to be encoded with integers starting from 0 to *nCategories* - 1, where *nCategories* is specified in the `createNominalAttribute` method. Any value outside this range is treated as a missing value. Patterns with both non-missing and missing values are used to train the classifier unless the `ignoreMissingValues` method has been set to `true`.

`classificationData` – an `int` array containing the target classifications for the training patterns. These must be encoded from zero to `nClasses`-1. Any value outside this range is considered a missing value. In this case, the data in that pattern are not used to train the Naive Bayes classifier. However, any pattern with missing values is still classified after the classifier is trained.

---

train

```
public void train(double[][] continuousData, int[][] nominalData, int[]
classificationData)
```

Description

Trains a Naive Bayes classifier for classifying data into one of `nClasses` target classifications.

Parameters

`continuousData` – a `double` matrix containing the training values for the `nContinuous` continuous attributes. The *i*-th row contains the input attributes for the *i*-th training pattern. The *j*-th column contains the values for the *j*-th continuous attribute. Missing values should be set to `Double.NaN`. Patterns with both non-missing and missing values are used to train the classifier unless the `ignoreMissingValues` method has been set to `true`.

`nominalData` – an `int` matrix containing the training values for the `nNominal` nominal attributes. The *i*-th row contains the input attributes for the *i*-th training pattern. The *j*-th column contains the classifications for the *j*-th nominal attribute. The values for the *j*-th nominal attribute are expected to be encoded with integers starting from 0 to *nCategories* - 1, where *nCategories* is specified in the `createNominalAttribute` method. Any value outside this range is treated as a missing value. Patterns with both non-missing and missing values are used to train the classifier unless the `ignoreMissingValues` method has been set to `true`.

`classificationData` – an `int` array containing the target classifications for the training patterns. These must be encoded from zero to `nClasses`-1. Any value outside this range is considered a missing value. In this case, the data in that pattern are not used to train the Naive Bayes classifier. However, any pattern with missing values is still classified after the classifier is trained.

Example 1: Continuous Attribute Example

Fisher's (1936) Iris data is often used for benchmarking classification algorithms. It consists of the following continuous input attributes and a classification target:

- *Continuous Attributes Usage*
  - Sepal Length
  - Sepal Width
  - Petal Length
  - Petal Width
- Classification of Iris Type
  - Setosa
  - Versicolour
  - Virginica

This example trains a Naive Bayes classifier using 140 of the 150 continuous patterns, then classifies ten unknown plants using their sepal and petal measurements.

```
import com.imsl.datamining.*;
import com.imsl.stat.NormalDistribution;

public class NaiveBayesClassifierEx1 {

    private static double[][] irisData = {
        {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}
    };

    public static void main(String[] args) throws Exception {
        /* Data corrections described in the KDD data mining archive */
        irisData[34][4] = 0.1;
        irisData[37][2] = 3.1;
        irisData[37][3] = 1.5;

        /*  Train first 140 patterns of the iris Fisher Data */
        int[] irisClassificationData = new int[irisData.length - 10];
        double[][] irisContinuousData
                = new double[irisData.length - 10][irisData[0].length - 1];

        for (int i = 0; i < irisData.length - 10; i++) {
            irisClassificationData[i] = (int) irisData[i][0] - 1;
            System.arraycopy(irisData[i], 1,
                    irisContinuousData[i], 0, irisData[0].length - 1);
        }

        int nNominal = 0;   /* no nominal input attributes      */

        int nContinuous = 4;   /* four continuous input attributes */

        int nClasses = 3;   /* three classification categories  */

        NaiveBayesClassifier nbTrainer
                = new NaiveBayesClassifier(nContinuous, nNominal, nClasses);

        for (int i = 0; i < nContinuous; i++) {
            nbTrainer.createContinuousAttribute(new NormalDistribution());
        }
        nbTrainer.train(irisContinuousData, irisClassificationData);

        int[][] classErrors = nbTrainer.getTrainingErrors();

        System.out.println("     Iris Classification Training Error Rates");
        System.out.println("------------------------------------------------");
        System.out.println("  Setosa   Versicolour   Virginica    |   Total");
        System.out.println("  " + classErrors[0][0] + "/" + classErrors[0][1]
```

```
                + "          " + classErrors[1][0] + "/" + classErrors[1][1]
                + "         " + classErrors[2][0] + "/" + classErrors[2][1]
                + "       |   " + classErrors[3][0]
                + "/" + classErrors[3][1]);
        System.out.println(
                "-------------------------------------------------\n\n\n");

        /*  Classify last  10 iris data patterns with the trained classifier */
        double[] continuousInput = new double[(irisData[0].length - 1)];
        double[] classifiedProbabilities = new double[nClasses];

        System.out.println("Probabilities for Incorrect Classifications");
        System.out.println(" Predicted   ");
        System.out.println(
                "   Class     |  Class       |   P(0)     P(1)     P(2) ");
        System.out.println(
                "--------------------------------------------------------");
        for (int i = 0; i < 10; i++) {
            int targetClassification
                    = (int) irisData[(irisData.length - 10) + i][0] - 1;
            System.arraycopy(irisData[(irisData.length - 10) + i],
                    1, continuousInput, 0, (irisData[0].length - 1));

            classifiedProbabilities
                    = nbTrainer.probabilities(continuousInput, null);
            int classification = nbTrainer.predictClass(continuousInput, null);
            if (classification == 0) {
                System.out.print("Setosa       |");
            } else if (classification == 1) {
                System.out.print("Versicolour  |");
            } else if (classification == 2) {
                System.out.print("Virginica    |");
            } else {
                System.out.print("Missing      |");
            }
            if (targetClassification == 0) {
                System.out.print(" Setosa       |");
            } else if (targetClassification == 1) {
                System.out.print(" Versicolour  |");
            } else if (targetClassification == 2) {
                System.out.print(" Virginica    |");
            } else {
                System.out.print(" Missing      |");
            }

            for (int j = 0; j < nClasses; j++) {
                Object[] pArgs = {new Double(classifiedProbabilities[j])};
                System.out.printf("   %2.3f ", pArgs);
            }
            System.out.println();
        }
    }
}
```

Output

The Naive Bayes classifier incorrectly classifies 6 of the 150 training patterns.

```
     Iris Classification Training Error Rates
------------------------------------------------
  Setosa   Versicolour   Virginica    |   Total
  0/50        0/50        20/40      |   20/140
------------------------------------------------



Probabilities for Incorrect Classifications
 Predicted
   Class     |  Class       |   P(0)     P(1)     P(2)
-------------------------------------------------------
Virginica    | Virginica    |   0.000    0.436    0.564
Virginica    | Virginica    |   0.000    0.466    0.534
Versicolour  | Virginica    |   0.000    0.542    0.458
Virginica    | Virginica    |   0.000    0.441    0.559
Virginica    | Virginica    |   0.000    0.412    0.588
Virginica    | Virginica    |   0.000    0.466    0.534
Versicolour  | Virginica    |   0.000    0.542    0.458
Versicolour  | Virginica    |   0.000    0.515    0.485
Virginica    | Virginica    |   0.000    0.460    0.540
Versicolour  | Virginica    |   0.000    0.551    0.449
```

Example 2: Nominal Attribute Usage

This example trains a Naive Bayes classifier using 24 training patterns with four nominal input attributes.

The first nominal attribute has three classifications and the others have two. The target classifications are contact lense prescriptions: hard, soft or neither recommended. These data are benchmark data from the Knowledge Discovery Databases archive maintained at the University of California, Irvine:

```
import com.imsl.datamining.*;

public class NaiveBayesClassifierEx2 {

    public static void main(String[] args) throws Exception {
        int[][] contactLensData = {
            {1, 1, 1, 1}, {1, 1, 1, 2}, {1, 1, 2, 1}, {1, 1, 2, 2},
            {1, 2, 1, 1}, {1, 2, 1, 2}, {1, 2, 2, 1}, {1, 2, 2, 2},
            {2, 1, 1, 1}, {2, 1, 1, 2}, {2, 1, 2, 1}, {2, 1, 2, 2},
            {2, 2, 1, 1}, {2, 2, 1, 2}, {2, 2, 2, 1}, {2, 2, 2, 2},
            {3, 1, 1, 1}, {3, 1, 1, 2}, {3, 1, 2, 1}, {3, 1, 2, 2},
            {3, 2, 1, 1}, {3, 2, 1, 2}, {3, 2, 2, 1}, {3, 2, 2, 2}
        };

        int[] classificationData = {
            3, 2, 3, 1,
            3, 2, 3, 1,
            3, 2, 3, 1,
            3, 2, 3, 3,
```

```
    3, 3, 3, 1,
    3, 2, 3, 3
};
/* classification values must start at 0 */
for (int i = 0; i < classificationData.length; i++) {
    classificationData[i] -= 1;
    for (int j = 0; j < contactLensData[0].length; j++) {
        contactLensData[i][j] -= 1;
    }
}
NaiveBayesClassifier nbTrainer = new NaiveBayesClassifier(0, 4, 3);

int nNominal = 4;
int categories[] = {3, 2, 2, 2};
for (int i = 0; i < nNominal; i++) {
    nbTrainer.createNominalAttribute(categories[i]);
}
nbTrainer.train(contactLensData, classificationData);

int[][] classErrors = nbTrainer.getTrainingErrors();

System.out.println("\n    Contact Lens Error Rates");
System.out.println("------------------------------------------------");
System.out.println("  Hard        Soft       Neither    |   Total");
System.out.println("  " + classErrors[0][0] + "/" + classErrors[0][1]
        + "         " + classErrors[1][0] + "/" + classErrors[1][1]
        + "        " + classErrors[2][0] + "/" + classErrors[2][1]
        + "       |   " + classErrors[3][0] + "/" + classErrors[3][1]);
System.out.println(
        "------------------------------------------------\n\n\n");

/*  Classify all patterns with the trained classifier */
int[] nominalInput = new int[contactLensData[0].length];
double[] classifiedProbabilities = new double[3];

System.out.println("Probabilities for Incorrect Classifications");
System.out.println(" Predicted   ");
System.out.println("   Class     |  Class       |   " + ""
        + "P(0)     P(1)     P(2)   |  classification error");
System.out.println("----------------------------------------"
        + "---------------------------------------");
for (int i = 0; i < contactLensData.length; i++) {
    System.arraycopy(contactLensData[i], 0,
            nominalInput, 0, contactLensData[0].length);

    classifiedProbabilities
            = nbTrainer.probabilities(null, nominalInput);
    int classification = nbTrainer.predictClass(null, nominalInput);
    double error
            = nbTrainer.classError(
                    null, nominalInput, classificationData[i]);
    if (classification == 0) {
        System.out.print(" Hard        |");
    } else if (classification == 1) {
        System.out.print(" Soft        |");
    } else if (classification == 2) {
```

```
            System.out.print(" Neither     |");
        } else {
            System.out.print(" Missing     |");
        }
        if (classificationData[i] == 0) {
            System.out.print(" Hard         |");
        } else if (classificationData[i] == 1) {
            System.out.print(" Soft         |");
        } else if (classificationData[i] == 2) {
            System.out.print(" Neither      |");
        } else {
            System.out.print(" Missing      |");
        }

        for (int j = 0; j < 3; j++) {
            Object[] pArgs = {new Double(classifiedProbabilities[j])};
            System.out.printf("   %2.3f ", pArgs);
        }
        System.out.println(" |  " + error);
    }
  }
}
```

Output

```
    Contact Lens Error Rates
------------------------------------------------
  Hard        Soft       Neither     |   Total
  0/4         0/5        1/15        |   1/24
------------------------------------------------



Probabilities for Incorrect Classifications
 Predicted
   Class     |  Class       |   P(0)     P(1)     P(2)   |  classification error
--------------------------------------------------------------------------------
 Neither     | Neither      |   0.044    0.130    0.827  |  0.17328273537224337
 Soft        | Soft         |   0.174    0.622    0.203  |  0.3777037515962849
 Neither     | Neither      |   0.186    0.018    0.795  |  0.20481484453118481
 Hard        | Hard         |   0.724    0.086    0.190  |  0.2762213881704395
 Neither     | Neither      |   0.019    0.154    0.827  |  0.1731192809428289
 Soft        | Soft         |   0.076    0.724    0.200  |  0.27580075885359856
 Neither     | Neither      |   0.092    0.024    0.884  |  0.11636647828065916
 Hard        | Hard         |   0.524    0.166    0.310  |  0.4759671271915248
 Neither     | Neither      |   0.025    0.113    0.862  |  0.1379488917335967
 Soft        | Soft         |   0.118    0.633    0.248  |  0.36667756449941913
 Neither     | Neither      |   0.113    0.017    0.870  |  0.1300941614979002
 Hard        | Hard         |   0.606    0.108    0.286  |  0.3943953202159177
 Neither     | Neither      |   0.011    0.133    0.856  |  0.1438071275344872
 Soft        | Soft         |   0.050    0.714    0.236  |  0.28621890950268947
 Neither     | Neither      |   0.054    0.021    0.925  |  0.07477065629881141
 Neither     | Neither      |   0.394    0.187    0.419  |  0.5812071082121901
 Neither     | Neither      |   0.023    0.068    0.909  |  0.09075297709414065
```

```
Soft        | Neither      |  0.142    0.509    0.349  |  0.6509258682358112
Neither     | Neither      |  0.099    0.010    0.891  |  0.1092518501445181
Hard        | Hard         |  0.599    0.071    0.330  |  0.40138301592296255
Neither     | Neither      |  0.010    0.081    0.909  |  0.09065883447725098
Soft        | Soft         |  0.062    0.594    0.344  |  0.40625618912343875
Neither     | Neither      |  0.047    0.012    0.941  |  0.059009463869502565
Neither     | Neither      |  0.391    0.124    0.485  |  0.514955474332508
```

Example 3: Naive Bayes Classifier Using User Supplied Probability Function

This example is the same as Example 1, using Fisher's (1936) Iris data to train a Naive Bayes classifier using 140 of the 150 continuous patterns, then classifies ten unknown plants using their sepal and petal measurements.

Instead of using the `NormalDistribution` class from the `com.imsl.stat` package, a user supplied normal (Gaussian) distribution is used. Rather than calculating the means and standard deviations from the data, as is done by the `NormalDistribution`'s `eval(double[])` method, the user supplied class requires the means and standard deviations in the class constructor. The output is the same as in Example 1, since the means and standard deviations in this example are simply rounded means and standard deviations of the actual data subset by target classifications.

```
import com.imsl.datamining.*;
import com.imsl.stat.ProbabilityDistribution;

public class NaiveBayesClassifierEx3 {

    private static double[][] irisData = {
        {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}
    };

    public static void main(String[] args) throws Exception {

        /* Data corrections described in the KDD data mining archive */
```

```
irisData[34][4] = 0.1;
irisData[37][2] = 3.1;
irisData[37][3] = 1.5;

/*  Train first 140 patterns of the iris Fisher Data */
int[] irisClassificationData = new int[irisData.length - 10];
double[][] irisContinuousData
        = new double[irisData.length - 10][irisData[0].length - 1];

for (int i = 0; i < irisData.length - 10; i++) {
    irisClassificationData[i] = (int) irisData[i][0] - 1;
    System.arraycopy(irisData[i], 1,
            irisContinuousData[i], 0, irisData[0].length - 1);
}

int nNominal = 0;   /* no nominal input attributes      */

int nContinuous = 4;   /* four continuous input attributes */

int nClasses = 3;   /* three classification categories  */

NaiveBayesClassifier nbTrainer
        = new NaiveBayesClassifier(nContinuous, nNominal, nClasses);

double[][] means = {
    {5.06, 5.94, 6.58}, {3.42, 2.8, 2.97},
    {1.5, 4.3, 5.6}, {0.25, 1.33, 2.1}
};
double[][] stdev = {
    {0.35, 0.52, 0.64}, {0.38, 0.3, 0.32},
    {0.17, 0.47, 0.55}, {0.12, 0.198, 0.275}
};

for (int i = 0; i < nContinuous; i++) {
    ProbabilityDistribution[] pdf
            = new ProbabilityDistribution[nClasses];
    for (int j = 0; j < nClasses; j++) {
        pdf[j] = new TestGaussFcn1(means[i][j], stdev[i][j]);
    }
    nbTrainer.createContinuousAttribute(pdf);
}
nbTrainer.train(irisContinuousData, irisClassificationData);

int[][] classErrors = nbTrainer.getTrainingErrors();

System.out.println("     Iris Classification Error Rates");
System.out.println("------------------------------------------------");
System.out.println("  Setosa   Versicolour   Virginica    |   Total");
System.out.println("  " + classErrors[0][0] + "/" + classErrors[0][1]
        + "         " + classErrors[1][0] + "/" + classErrors[1][1]
        + "        " + classErrors[2][0] + "/" + classErrors[2][1]
        + "       |   " + classErrors[3][0] + "/" + classErrors[3][1]);
System.out.println(
        "------------------------------------------------\n\n\n");
```

```
        /*  Classify last  10 iris data patterns with the trained classifier */
        double[] continuousInput = new double[(irisData[0].length - 1)];
        double[] classifiedProbabilities = new double[nClasses];

        System.out.println("Probabilities for Incorrect Classifications");
        System.out.println(" Predicted   ");
        System.out.println(
                "   Class     |  Class       |   P(0)     P(1)     P(2) ");
        System.out.println(
                "-------------------------------------------------------");
        for (int i = 0; i < 10; i++) {
            int targetClassification
                    = (int) irisData[(irisData.length - 10) + i][0] - 1;
            System.arraycopy(irisData[(irisData.length - 10) + i], 1,
                    continuousInput, 0, (irisData[0].length - 1));

            classifiedProbabilities
                    = nbTrainer.probabilities(continuousInput, null);
            int classification = nbTrainer.predictClass(continuousInput, null);
            if (classification == 0) {
                System.out.print("Setosa       |");
            } else if (classification == 1) {
                System.out.print("Versicolour  |");
            } else if (classification == 2) {
                System.out.print("Virginica    |");
            } else {
                System.out.print("Missing      |");
            }
            if (targetClassification == 0) {
                System.out.print(" Setosa       |");
            } else if (targetClassification == 1) {
                System.out.print(" Versicolour  |");
            } else if (targetClassification == 2) {
                System.out.print(" Virginica    |");
            } else {
                System.out.print(" Missing      |");
            }
            for (int j = 0; j < nClasses; j++) {
                Object[] pArgs = {new Double(classifiedProbabilities[j])};
                System.out.printf("   %2.3f ", pArgs);
            }
            System.out.println("");
        }
    }

    static public class TestGaussFcn1 implements ProbabilityDistribution {

        private double mean;
        private double stdev;

        public TestGaussFcn1(double mean, double stdev) {
            this.mean = mean;
            this.stdev = stdev;
        }

        public double[] eval(double[] xData) {
```

```
            double[] pdf = new double[xData.length];
            for (int i = 0; i < xData.length; i++) {
                pdf[i] = eval(xData[i], null);
            }
            return pdf;
        }

        public double[] eval(double[] xData, Object[] Params) {
            double[] pdf = new double[xData.length];
            for (int i = 0; i < xData.length; i++) {
                pdf[i] = eval(xData[i], Params);
            }
            return pdf;
        }

        public double eval(double xData, Object[] Params) {
            return l_gaussian_pdf(xData, mean, stdev);
        }

        public Object[] getParameters() {
            Double[] parms = new Double[2];
            parms[0] = this.mean;
            parms[1] = this.stdev;
            return parms;
        }

        private double l_gaussian_pdf(double x, double mean, double stdev) {
            double e, phi2, z, s;
            double sqrt_pi2 = 2.506628274631; /* sqrt(2*pi) */

            if (Double.isNaN(x)) {
                return Double.NaN;
            }
            if (Double.isNaN(mean) || Double.isNaN(stdev)) {
                return Double.NaN;
            } else {
                z = x;
                z -= mean;
                s = stdev;
                phi2 = sqrt_pi2 * s;
                e = -0.5 * (z * z) / (s * s);
                return Math.exp(e) / phi2;
            }
        }
    }
}
```

Output

The Naive Bayes classifier incorrectly classifies 6 of the 150 training patterns.

```
     Iris Classification Error Rates
------------------------------------------------
  Setosa   Versicolour   Virginica    |   Total
```

```
  0/50         2/50        4/40       |   6/140
------------------------------------------------


Probabilities for Incorrect Classifications
 Predicted
   Class     |  Class       |   P(0)     P(1)     P(2)
-------------------------------------------------------
Virginica    | Virginica    |   0.000    0.000    1.000
Virginica    | Virginica    |   0.000    0.000    1.000
Virginica    | Virginica    |   0.000    0.051    0.949
Virginica    | Virginica    |   0.000    0.000    1.000
Virginica    | Virginica    |   0.000    0.000    1.000
Virginica    | Virginica    |   0.000    0.000    1.000
Virginica    | Virginica    |   0.000    0.048    0.952
Virginica    | Virginica    |   0.000    0.001    0.999
Virginica    | Virginica    |   0.000    0.000    1.000
Virginica    | Virginica    |   0.000    0.126    0.874
```

Itemsets class

`public class com.imsl.datamining.Itemsets implements Serializable, Cloneable`

Object containing a set of frequent items and the number of transactions examined to obtain the frequent item set.

Methods

getItemset

`public int[] getItemset(int i)`

Description

Returns a particular itemset.

Parameter

`i` – an `int` indicating which itemset to retrieve.

Returns

an `int` array that contains the ith itemset.

getItemsetsMatrix

`public int[][] getItemsetsMatrix()`

Description

Returns the set of `Itemsets` as an integer matrix.

Returns

an `int` matrix. The number of rows is consistent with the number of item sets that meet the specified support criterion. The rows are jagged. The length of each row is determined by the number of elements (frequently simultaneous items) in the item set and an element for the corresponding support. Therfore an item set with two elements would produce a row of length 3. The first two elements of the row would be the items of the set, with the support at position length-1; that is, {item1, item2, support value}. More generally: {item1, ..., itemN, support value}.

getNumberOfItemsets

```
public int getNumberOfItemsets()
```

Description

Returns the number of itemsets in this `Itemsets`.

Returns

an `int` specifying the number of itemsets in this `Itemsets`.

getNumberOfTransactions

```
public int getNumberOfTransactions()
```

Description

Returns an `int` indicating the number of transactions used to construct the itemsets.

Returns

an `int` indicating the number of transactions used to construct the itemsets.

getSupport

```
public double getSupport(int i)
```

Description

Returns the support determined for a particular itemset.

Parameter

`i` – an `int` indicating which itemset to retrieve the support for.

Returns

a `double` that specifies the support determined for the ith itemset.

print

```
public void print()
```

Description

Prints a standard representation of the members of this object.

AssociationRule class

```
public class com.imsl.datamining.AssociationRule implements Serializable,
Cloneable
```

Contains association rules discovered by the Apriori algorithm.

An association rule has the form, $X \implies Y$, where X and Y are two disjoint sets of items (or products) that are represented in a set of occurrences (transactions). The `AssociationRule` object contains members:X: $\{X_1,\ldots,X_{n_X}\}$
Y: $\{Y_1,\ldots,Y_{n_Y}\}$
Support of $Z = X \cap Y$
Support of X
Support of Y
Confidence measure for $X \implies Y$
Lift measure for $X \implies Y$

Methods

getConfidence

```
public double getConfidence()
```

Description

The confidence measure of the association rule.

Returns

a `double` specifying the confidence measure of the association rule.

getLift

```
public double getLift()
```

Description

The lift measure of the association rule.

Returns

a `double` specifying the lift measure of the association rule.

getSupport

```
public int[] getSupport()
```

Description

Support for the Z, X, and Y components of the association rule.

Returns

an `int` array containing the Z, X, and Y components of the association rule.

getX

```
public int[] getX()
```

Description

The X components of the association rule.

Returns

an `int` array containing the X components of the association rule.

getY

```
public int[] getY()
```

Description

The Y components of the association rule.

Returns

an `int` array containing the Y components of the association rule.

print

```
public void print()
```

Description

Print the member data in this object.

The following is an example of this format:

X = {0} $\Longrightarrow$ Y = {2}
supp(X)=27, supp(Y)=33, supp(X and Y)=22
conf= 0.81, lift=1.23

print

```
static public void print(AssociationRule[] ar)
```

Description

Print out the association rules in `ar`.

Parameter

`ar` – an `AssociationRule` array containing the association rules generated from a set of itemsets.

Apriori class

```
public class com.imsl.datamining.Apriori implements Serializable, Cloneable
```

Performs the Apriori algorithm for association rule discovery.

Association rules are statements of the form, "if X, then Y", given with some measure of confidence. The main application for association rule discovery is market basket analysis, where X and Y are products or groups of products, and the occurrences are individual transactions, or "market baskets". The results help sellers learn relationships between the different products they sell, supporting better marketing decisions. There are other applications for association rule discovery, such as the problem areas of text mining and bioinformatics. The Apriori algorithm (Agrawal and Srikant, 1994) is one of the most popular algorithms for association rule discovery in transactional datasets.

In the first and most critical stage, the Apriori algorithm mines the transactions for frequent itemsets. An itemset is frequent if it appears in more than a minimum number of transactions. The number of transactions containing an itemset is known as its "support", and the minimum support (as a percentage of transactions) is a control parameter in the algorithm. The algorithm begins by finding the frequent single items. Then the algorithm generates all two-item sets from the frequent single items and determines which among them are frequent. From the collection of frequent pairs, Apriori forms candidate three-item subsets and determines which are frequent, and so on. The algorithm stops when either a maximum itemset size is reached, or when none of the candidate itemsets are frequent. In this way, the Apriori algorithm exploits the apriori-property: for an itemset to be frequent, all of its proper subsets must also be frequent. At each step the problem is reduced to only the frequent subsets.

In the second stage, the algorithm generates association rules. These are of the form, $X \implies Y$ (read, "if X, then Y"), where Y and X are disjoint frequent itemsets. The confidence measure associated with the rule is defined as the proportion of transactions containing X that also contain Y. Denote the support of X (the number of transactions containing X) as S_X and S_Z is the support of $Z = X \cup Y$. The confidence of the rule, $X \implies Y$ is the ratio, S_Z/S_X. Note that the confidence ratio is the conditional probability

$$P[Y|X] = \frac{P[XY]}{P[X]}$$

where $P[XY]$ denotes the probability of both X and Y. The probability of an itemset X is estimated by S_X/N, where N is the total number of transactions.

Another measure of the strength of the association is known as the lift, which is the ratio $(S_Z N)/(S_X S_Y)$. Lift values close to 1.0 suggest the sets are independent, and that they occur together by chance. Large lift values indicate a strong association. A minimum confidence threshold and a lift threshold can be specified.

If the transaction dataset is too large to fit in memory, Apriori can be applied to separate blocks of the transactions. The union of the frequent itemsets from each of the blocks becomes the set of candidate frequent itemsets. The frequencies of the itemsets are then accumulated over each block of data, and those with minimum support are the frequent itemsets for the entire dataset. The method is due to Savasere, Omiecinski, and Navathe (1995) and is also summarized and compared with other approaches in Rajaran and Ullman (2010).

Methods

countFrequency

```
static public int[] countFrequency(Itemsets candItemsets, int[][] x)
```

Description

Returns the frequency of each itemset in the transaction data set, `x`.

Parameters

`candItemsets` – an `Itemsets` object containing the candidate itemsets and the corresponding number of transactions.

`x` – an `int[][]` containing the transaction data. The first column of `x` contains the transaction IDs, and the second column contains the item IDs. There will be a row for each transaction ID/ item ID combination, and all records for a single transaction ID must be in adjacent rows. Furthermore, the algorithm assumes that an individual transaction is complete within a single dataset. That is, there is no matching of transaction IDs between different data sets.

Returns

an `int` array of length equal to the number of sets in `candItemsets` containing the frequencies of each itemset in `x`.

countFrequency

```
static public int[] countFrequency(Itemsets candItemsets, int[][] x, int[]
prevFrequencies)
```

Description

Returns the frequency of each itemset in the transaction data set, `x`, added to the previous frequencies.

Parameters

`candItemsets` – an `Itemsets` object containing the candidate itemsets and the corresponding number of transactions.

`x` – an `int[][]` containing the transaction data. The first column of `x` contains the transaction IDs, and the second column contains the item IDs. There will be a row for each transaction ID/ item ID combination, and all records for a single transaction ID must be in adjacent rows. Furthermore, the algorithm assumes that an individual transaction is complete within a single dataset. That is, there is no matching of transaction IDs between different data sets.

`prevFrequencies` – an `int` array containing the itemset frequencies counted so far in other blocks of transaction data.

Returns

an `int` array of length equal to the number of sets in `candItemsets` containing the frequencies of each itemset in `x` added to `prevFrequencies`.

getAssociationRules

```
static public AssociationRule[] getAssociationRules(Itemsets itemsets, double
confidence, double lift)
```

Description

Returns strong association rules among the itemsets in `itemsets`.

`confidence` and `lift` are the two criterion used to determine a strong association. If either is exceeded, the association rule will be considered "strong".

Parameters

`itemsets` – an `Itemsets` object that contains the itemsets.

`confidence` – a `double` scalar containing the minimum confidence used to determine the strong association rules. `confidence` must be in [0,1].
`lift` is the other criterion which determines whether an association is "strong". If either criterion, `confidence` or `lift` is exceeded, the association rule will be considered "strong".

`lift` – a `double` scalar containing the minimum lift used to determine the strong association rules. `lift` must be non-negative.
If either criterion, `lift` or `confidence` is exceeded, the association rule will be considered "strong".

Returns

an `AssociationRule` object containing association rules.

getFrequentItemsets

```
static public Itemsets getFrequentItemsets(int[][] x, int maxNumProducts, int
maxSetSize, double minPctSupport)
```

Description

Computes the frequent itemsets in the transaction set `x`.

Parameters

`x` – an `int[][]` containing the transaction data. The first column of `x` contains the transaction IDs, and the second column contains the item IDs. There will be a row for each transaction ID/ item ID combination, and all records for a single transaction ID must be in adjacent rows. Furthermore, the algorithm assumes that an individual transaction is complete within a single dataset. That is, there is no matching of transaction IDs between different data sets.

`maxNumProducts` – an `int` scalar containing the maximum number of items or products that may be present in the transactions. `maxNumProducts` must be greater than or equal to the number of items in `x`.

`maxSetSize` – an `int` scalar containing the maximum size of an itemset.
Only frequent itemsets with `maxSetSize` or fewer items are considered in the analysis.

`minPctSupport` – a `double` scalar containing the minimum percentage of transactions in which an item or itemset must be present to be considered frequent. `minPctSupport` must be in the interval [0,1].

Returns

an `Itemsets` containing the frequent itemsets.

getUnion

```
static public Itemsets getUnion(Itemsets[] itemsets)
```

Description

Return the union of a sequence of sets of itemsets.

The attributes of each `Itemsets` object must agree. If `maxNumProducts`, `maxSetSize` or `minPctSupport` differ between the input arguments, an exception will be thrown.

Parameter

`itemsets` – a sequence of `Itemsets` objects containing the frequent itemsets for the union.

Returns

an `Itemsets` object containing the union of the itemsets.

sum

```
static public int[] sum(int[][] freq)
```

Description

Sums up the itemset frequencies.

Parameter

`freq` – a sequence of `int` arrays containing the itemset frequencies counted over one or more blocks of transaction data.

Note: the length of the arrays in the sequence must be equal.

Returns

an `int` array containing the sum of the frequencies.

updateFrequentItemsets

```
static public Itemsets updateFrequentItemsets(Itemsets candItemsets, int[]
freq)
```

Description

Updates the set of frequent items in `candItemsets`. The minimum support is determined by `minSupportPercentage` times the total number of transactions.

Parameters

`candItemsets` – an `Itemsets` object containing the candidate itemsets and the corresponding number of transactions.

`freq` – an `int` array containing the frequencies for each itemset in `candItemsets`.

Returns

an `Itemsets` object containing the updated itemsets.

Example 1: Apriori

The data are 50 transactions involving five different product ID's. This example applies Apriori to find the frequent itemsets and strong association rules. The minimum support percentage is set to 0.30, giving a minimum required support of 15 transactions.

```
import com.imsl.datamining.*;

public class AprioriEx1 {

    public static void main(String[] args) {
        int nprods = 5;
        int[][] x = {
            {1, 3}, {1, 2}, {1, 1}, {2, 1}, {2, 2}, {2, 4}, {2, 5},
            {3, 3}, {4, 4}, {4, 3}, {4, 5}, {4, 1}, {5, 5}, {6, 1}, {6, 2},
            {6, 3}, {7, 5}, {7, 3}, {7, 2}, {8, 3}, {8, 4}, {8, 1}, {8, 5},
            {8, 2}, {9, 4}, {10, 5}, {10, 3}, {11, 2}, {11, 3}, {12, 4},
            {13, 4}, {14, 2}, {14, 3}, {14, 1}, {15, 3}, {15, 5}, {15, 1},
            {16, 2}, {17, 3}, {17, 5}, {17, 1}, {18, 5}, {18, 1}, {18, 2},
            {18, 3}, {19, 2}, {20, 4}, {21, 1}, {21, 4}, {21, 2}, {21, 5},
            {22, 5}, {22, 4}, {23, 2}, {23, 5}, {23, 3}, {23, 1}, {23, 4},
            {24, 3}, {24, 1}, {24, 5}, {25, 3}, {25, 5}, {26, 1}, {26, 4},
            {26, 2}, {26, 3}, {27, 2}, {27, 3}, {27, 1}, {27, 5}, {28, 5},
            {28, 3}, {28, 4}, {28, 1}, {28, 2}, {29, 4}, {29, 5}, {29, 2},
            {30, 2}, {30, 4}, {30, 3}, {31, 2}, {32, 5}, {32, 1}, {32, 4},
            {33, 4}, {33, 1}, {33, 5}, {33, 3}, {33, 2}, {34, 3}, {35, 5},
            {35, 3}, {36, 3}, {36, 5}, {36, 4}, {36, 1}, {36, 2}, {37, 1},
            {37, 3}, {37, 2}, {38, 4}, {38, 2}, {38, 3}, {39, 3}, {39, 2},
            {39, 1}, {40, 2}, {40, 1}, {41, 3}, {41, 5}, {41, 1}, {41, 4},
            {41, 2}, {42, 5}, {42, 1}, {42, 4}, {43, 3}, {43, 2}, {43, 4},
            {44, 4}, {44, 5}, {44, 2}, {44, 3}, {44, 1}, {45, 4}, {45, 5},
            {45, 3}, {45, 2}, {45, 1}, {46, 2}, {46, 4}, {46, 5}, {46, 3},
            {46, 1}, {47, 4}, {47, 5}, {48, 2}, {49, 1}, {49, 4}, {49, 3},
            {50, 3}, {50, 4}
        };

        // Run the algorithm on transactions X.
        Itemsets itemsets = Apriori.getFrequentItemsets(x, nprods, 10, 0.30);
        itemsets.print();

        // Get and print the strong association rules.
        AssociationRule.print(Apriori.getAssociationRules(itemsets, 0.80, 2.0));
    }
}
```

Output

```
Frequent Itemsets (Out of 50 Transactions):
Size   Support  Itemset
1      27       {0}
1      30       {1}
1      33       {2}
1      27       {3}
1      27       {4}
2      20       {0 1}
2      22       {0 2}
2      16       {0 3}
2      19       {0 4}
2      22       {1 2}
2      16       {1 3}
2      15       {1 4}
```

```
2       16       {2 3}
2       19       {2 4}
2       17       {3 4}
3       17       {0 1 2}
3       15       {0 2 4}

Association Rules (itemset X implies itemset Y):
X = {0} ==> Y = {2}
  supp(X)=27, supp(Y)=33, supp(X and Y)=22
  conf= 0.81, lift=1.23
X = {0 1} ==> Y = {2}
  supp(X)=20, supp(Y)=33, supp(X and Y)=17
  conf= 0.85, lift=1.29
```

Example 2: Apriori

The data are two separate blocks of 50 transactions involving five different product ID's. This example shows how to apply Apriori to separate blocks of data and combine results. The minimum support percentage is set to 0.30, giving a minimum required support of 30 transactions overall.

```
import com.imsl.datamining.*;

public class AprioriEx2 {

    public static void main(String[] args) {
        int numProducts = 5;
        int[][] x1 = {
            {1, 3}, {1, 2}, {1, 1}, {2, 1}, {2, 2}, {2, 4}, {2, 5},
            {3, 3}, {4, 4}, {4, 3}, {4, 5}, {4, 1}, {5, 5}, {6, 1}, {6, 2},
            {6, 3}, {7, 5}, {7, 3}, {7, 2}, {8, 3}, {8, 4}, {8, 1}, {8, 5},
            {8, 2}, {9, 4}, {10, 5}, {10, 3}, {11, 2}, {11, 3}, {12, 4},
            {13, 4}, {14, 2}, {14, 3}, {14, 1}, {15, 3}, {15, 5}, {15, 1},
            {16, 2}, {17, 3}, {17, 5}, {17, 1}, {18, 5}, {18, 1}, {18, 2},
            {18, 3}, {19, 2}, {20, 4}, {21, 1}, {21, 4}, {21, 2}, {21, 5},
            {22, 5}, {22, 4}, {23, 2}, {23, 5}, {23, 3}, {23, 1}, {23, 4},
            {24, 3}, {24, 1}, {24, 5}, {25, 3}, {25, 5}, {26, 1}, {26, 4},
            {26, 2}, {26, 3}, {27, 2}, {27, 3}, {27, 1}, {27, 5}, {28, 5},
            {28, 3}, {28, 4}, {28, 1}, {28, 2}, {29, 4}, {29, 5}, {29, 2},
            {30, 2}, {30, 4}, {30, 3}, {31, 2}, {32, 5}, {32, 1}, {32, 4},
            {33, 4}, {33, 1}, {33, 5}, {33, 3}, {33, 2}, {34, 3}, {35, 5},
            {35, 3}, {36, 3}, {36, 5}, {36, 4}, {36, 1}, {36, 2}, {37, 1},
            {37, 3}, {37, 2}, {38, 4}, {38, 2}, {38, 3}, {39, 3}, {39, 2},
            {39, 1}, {40, 2}, {40, 1}, {41, 3}, {41, 5}, {41, 1}, {41, 4},
            {41, 2}, {42, 5}, {42, 1}, {42, 4}, {43, 3}, {43, 2}, {43, 4},
            {44, 4}, {44, 5}, {44, 2}, {44, 3}, {44, 1}, {45, 4}, {45, 5},
            {45, 3}, {45, 2}, {45, 1}, {46, 2}, {46, 4}, {46, 5}, {46, 3},
            {46, 1}, {47, 4}, {47, 5}, {48, 2}, {49, 1}, {49, 4}, {49, 3},
            {50, 3}, {50, 4}
        };

        int[][] x2 = {
            {1, 2}, {1, 1}, {1, 4}, {1, 3}, {2, 2}, {2, 5}, {2, 3}, {2, 1},
            {2, 4}, {3, 5}, {3, 4}, {4, 2}, {5, 4}, {5, 2}, {5, 3}, {5, 5},
            {6, 3}, {6, 5}, {7, 2}, {7, 5}, {7, 4}, {7, 1}, {7, 3}, {8, 2},
```

```
            {9, 2}, {9, 4}, {10, 4}, {10, 2}, {11, 4}, {11, 1}, {12, 3},
            {12, 1}, {12, 5}, {12, 2}, {13, 2}, {14, 3}, {14, 4}, {14, 2},
            {15, 2}, {16, 5}, {16, 2}, {16, 4}, {17, 1}, {18, 2}, {18, 3},
            {18, 4}, {19, 3}, {19, 1}, {19, 2}, {19, 4}, {20, 5}, {20, 1},
            {21, 5}, {21, 4}, {21, 1}, {21, 3}, {22, 4}, {22, 1}, {22, 5},
            {23, 1}, {23, 2}, {24, 4}, {25, 4}, {25, 3}, {26, 5}, {26, 2},
            {26, 3}, {26, 4}, {26, 1}, {27, 2}, {27, 1}, {27, 5}, {27, 3},
            {28, 1}, {28, 2}, {28, 3}, {28, 4}, {29, 5}, {29, 2}, {29, 1},
            {30, 5}, {30, 3}, {30, 2}, {30, 4}, {31, 4}, {31, 1}, {32, 1},
            {32, 2}, {32, 3}, {32, 4}, {32, 5}, {33, 3}, {33, 2}, {33, 4},
            {33, 5}, {33, 1}, {34, 3}, {34, 4}, {34, 5}, {34, 2}, {35, 2},
            {35, 3}, {36, 3}, {36, 5}, {36, 4}, {37, 1}, {37, 4}, {37, 2},
            {37, 3}, {37, 5}, {38, 5}, {38, 3}, {38, 1}, {38, 2}, {39, 2},
            {39, 5}, {40, 4}, {40, 2}, {41, 4}, {42, 4}, {43, 5}, {43, 4},
            {44, 5}, {44, 4}, {44, 3}, {44, 2}, {44, 1}, {45, 1}, {45, 2},
            {45, 3}, {45, 5}, {45, 4}, {46, 3}, {46, 4}, {47, 4}, {47, 5},
            {47, 2}, {47, 3}, {48, 5}, {48, 3}, {48, 2}, {48, 1}, {48, 4},
            {49, 4}, {49, 5}, {50, 4}, {50, 1}
        };

        // Find frequent itemsets in x1 and x2.
        Itemsets fis1
                = Apriori.getFrequentItemsets(x1, numProducts, 4, 0.30);
        Itemsets fis2
                = Apriori.getFrequentItemsets(x2, numProducts, 4, 0.30);

        // Get the union of fis1 and fis2.
        Itemsets cis = Apriori.getUnion(fis1, fis2);

        // Count the frequencies of the itemsets in the union in
        // each of the data sets.
        int[] freq1 = Apriori.countFrequency(cis, x1);
        int[] freq2 = Apriori.countFrequency(cis, x2);
        int[] freq = Apriori.sum(freq1, freq2);

        // Get the frequent itemset of the union.
        Itemsets itemsets = Apriori.updateFrequentItemsets(cis, freq);
        itemsets.print();

        // Generate and print the association rules.
        AssociationRule.print(Apriori.getAssociationRules(itemsets, 0.80, 2.0));
    }
}
```

Output

```
Frequent Itemsets (Out of 100 Transactions):
Size   Support  Itemset
1      51       {0}
1      63       {1}
1      60       {2}
1      63       {3}
1      54       {4}
2      37       {0 1}
2      38       {0 2}
```

```
2      33      {0 3}
2      35      {0 4}
2      44      {1 2}
2      38      {1 3}
2      34      {1 4}
2      38      {2 3}
2      38      {2 4}
2      37      {3 4}
3      32      {0 1 2}
3      31      {1 2 3}

Association Rules (itemset X implies itemset Y):
X = {0 1} ==> Y = {2}
  supp(X)=37, supp(Y)=60, supp(X and Y)=32
  conf= 0.86, lift=1.44
X = {0 2} ==> Y = {1}
  supp(X)=38, supp(Y)=63, supp(X and Y)=32
  conf= 0.84, lift=1.34
X = {1 3} ==> Y = {2}
  supp(X)=38, supp(Y)=60, supp(X and Y)=31
  conf= 0.82, lift=1.36
X = {2 3} ==> Y = {1}
  supp(X)=38, supp(Y)=63, supp(X and Y)=31
  conf= 0.82, lift=1.29
```

KohonenSOM class

`public class com.imsl.datamining.KohonenSOM implements Serializable, Cloneable`

A Kohonen self organizing map.

A self-organizing map (SOM), also known as a Kohonen map or Kohonen SOM, is a technique for gathering high-dimensional data into clusters that are constrained to lie in low dimensional space, usually two dimensions. A Kohonen map is a widely used technique for the purpose of feature extraction and visualization for very high dimensional data in situations where classifications are not known beforehand. The Kohonen SOM is equivalent to an artificial neural network having inputs linked to every node in the network. Self-organizing maps use a neighborhood function to preserve the topological properties of the input space.

In a Kohonen map, nodes are arranged in a rectangular or hexagonal grid or lattice. The input is connected to each node, and the output of the Kohonen map is the zero-based (i, j) index of the node that is closest to the input. A Kohonen map involves two steps: training and forecasting. Training builds the map using input examples (vectors), and forecasting classifies a new input.

During training, an input vector is fed to the network. The input's Euclidean distance from all the nodes is calculated. The node with the shortest distance is identified and is called the Best Matching Unit, or BMU. After identifying the BMU, the weights of the BMU and the nodes closest to it in the SOM lattice are updated towards the input vector. The magnitude of the update decreases with time and with distance

(within the lattice) from the BMU. The weights of the nodes surrounding the BMU are updated according to:

$$W_{t+1} = W_t + \alpha(t) * h(d,t) * (D_t - W_t)$$

where W_t represents the node weights, $\alpha(t)$ is the monotonically decreasing learning coefficient function, $h(d,t)$ is the neighborhood function, d is the lattice distance between the node and the BMU, and D_t is the input vector.

The monotonically decreasing learning coefficient function $\alpha(t)$ is a scalar factor that defines the size of the update correction. The value of $\alpha(t)$ decreases with the step index t.

The neighborhood function $h(d,t)$ depends on the lattice distance d between the node and the BMU, and represents the strength of the coupling between the node and BMU. In the simplest form, the value of $h(d,t)$ is 1 for all nodes closest to the BMU and 0 for others, but a Gaussian function is also commonly used. Regardless of the functional form, the neighborhood function shrinks with time (Hollmén, 15.2.1996). Early on, when the neighborhood is broad, the self-organizing takes place on the global scale. When the neighborhood has shrunk to just a couple of nodes, the weights converge to local estimates.

Note that in a rectangular grid, the BMU has four closest nodes for the Von Neumann neighborhood type, or eight closest nodes for the Moore neighborhood type. In a hexagonal grid, the BMU has six closest nodes.

During training, this process is repeated for a number of iterations on all input vectors.

During forecasting, the node with the shortest Euclidean distance is the winning node, and its (i, j) index is the output.

Fields

GRID_HEXAGONAL

```
static final public int GRID_HEXAGONAL
```

Indicates a hexagonal grid.

GRID_RECTANGULAR

```
static final public int GRID_RECTANGULAR
```

Indicates a rectangular grid.

TYPE_MOORE

```
static final public int TYPE_MOORE
```

Indicates a Moore neighborhood type.

TYPE_VON_NEUMANN

```
static final public int TYPE_VON_NEUMANN
```

Indicates a Von Neumann neighborhood type.

Constructor

KohonenSOM

`public KohonenSOM(int dim, int nrow, int ncol)`

Description

Constructor for a `KohonenSOM` object.

Parameters

`dim` – An `int` scalar containing the number of weights for each node in the node grid. `dim` must be greater than zero.

`nrow` – An `int` scalar containing the number of rows in the node grid. `nrow` must be greater than zero.

`ncol` – An `int` scalar containing the number of columns in the node grid. `ncol` must be greater than zero.

Methods

forecast

`public int[] forecast(double[] input)`

Description

Returns a forecast computed using the `KohonenSOM` object.

Parameter

`input` – A `double` array containing the input data. `input.length` must be equal to `dim`.

Returns

An `int` array of length 2 containing the (i, j) index of the output node.

forecast

`public int[][] forecast(double[][] input)`

Description

Returns forecasts computed using the `KohonenSOM` object.

Parameter

`input` – A `double` matrix containing `input.length` observations of data. `input[i].length` must be equal to `dim`.

Returns

An `int` matrix containing the output indices of the nodes. The i-th row contains the (i, j) index of the output node for `input[i]`.

getDimension

`public int getDimension()`

Description

Returns the number of weights for each node.

Returns

An `int` scalar containing the number of weights for each node.

getGridType

`public int getGridType()`

Description

Returns the grid type.

Returns

An `int` scalar containing the grid type. The return value is either `KohonenSOM.GRID_RECTANGULAR` or `KohonenSOM.GRID_HEXAGONAL`

getNeighborhoodType

`public int getNeighborhoodType()`

Description

Returns the neighborhood type for the rectangular grid.

Returns

An `int` scalar containing the neighborhood type. The return value is either `KohonenSOM.TYPE_VON_NEUMANN` or `KohonenSOM.TYPE_MOORE`

getNumberOfColumns

`public int getNumberOfColumns()`

Description

Returns the number of columns of the node grid.

Returns

An `int` scalar containing the number of columns of the node grid.

getNumberOfRows

`public int getNumberOfRows()`

Description

Returns the number of rows of the node grid.

Returns

An `int` scalar containing the number of rows of the node grid.

getWeights

`public double[][][] getWeights()`

Description

Returns the weights of the nodes.

Returns

An `nrow` by `ncol` matrix of double arrays containing the weights of the nodes.

getWeights

`public double[] getWeights(int i, int j)`

Description

Returns the weights of the node at (i, j) in the node grid.

Parameters

`i` – An `int` scalar containing the row index of the node in the node grid, where $0 \le \mathrm{i} \le \mathrm{nrow} - 1$.

`j` – An `int` scalar containing the column index of the node in the node grid, where $0 \le \mathrm{j} \le \mathrm{ncol} - 1$.

Returns

A `double` array containing the weights of the node at (i, j) in the node grid.

isWrapAround

`public boolean isWrapAround()`

Description

Returns whether the opposite edges are connected or not.

Returns

A `boolean` indicating whether or not the opposite edges are connected. It is true if the opposite edges are connected. Otherwise, it is false.

setGridType

`public void setGridType(int type)`

Description

Sets the grid type.

Parameter

`type` – An `int` scalar containing the grid type, rectangular (`KohonenSOM.GRID_RECTANGULAR`) or hexagonal (`KohonenSOM.GRID_HEXAGONAL`).

Default: `type` = `GRID_RECTANGULAR`.

`type`	*Description*
`GRID_RECTANGULAR`	Use a rectangular grid (`type` = 0).
`GRID_HEXAGONAL`	Use a hexagonal grid (`type` = 1).

setNeighborhoodType

```
public void setNeighborhoodType(int type)
```

Description

Sets the neighborhood type.

Parameter

type – An int scalar containing the neighborhood type, Von Neumann (KohonenSOM.TYPE_VON_NEUMANN) or Moore (KohonenSOM.TYPE_MOORE). This method is ignored for a hexagonal grid.

Default: type = TYPE_VON_NEUMANN.

type	*Description*
TYPE_VON_NEUMANN	Use the Von Neumann (type = 0) neighborhood type.
TYPE_MOORE	Use the Moore (type = 1) neighborhood type.

setWeights

```
public void setWeights()
```

Description

Sets the weights of the nodes using random numbers. The weights are in [0.0, 1.0].

setWeights

```
public void setWeights(Random random)
```

Description

Sets the weights of the nodes using a Random object. The weights are generated using the Random.nextDouble method.

Parameter

random – A Random object used to generate random numbers for the nodes.

setWeights

```
public void setWeights(double[][][] weights)
```

Description

Sets the weights of the nodes.

Parameter

weights – An nrow by ncol matrix of double arrays containing the weights of the nodes. weights[i][j].length must be equal to dim.

setWeights

```
public void setWeights(int i, int j, double[] weights)
```

Description

Sets the weights of the node at (i, j) in the node grid.

Parameters

`i` – An `int` scalar containing the row index of the node in the node grid, where $0 \le i \le nrow - 1$.

`j` – An `int` scalar containing the column index of the node in the node grid, where $0 \le j \le ncol - 1$.

`weights` – A `double` array containing the weights. `weights.length` must be equal to `dim`.

---

wrapAround

```
public void wrapAround()
```

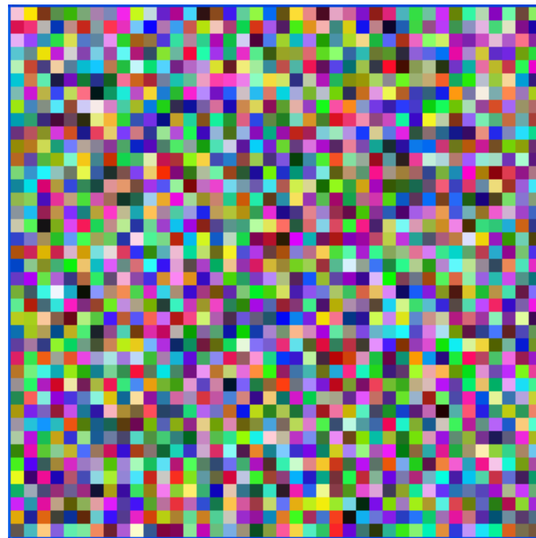
Description

Sets a flag to indicate the map should wrap around or connect opposite edges. A hexagonal grid must have an even number of rows to wrap around. By default, opposite edges are not connected.

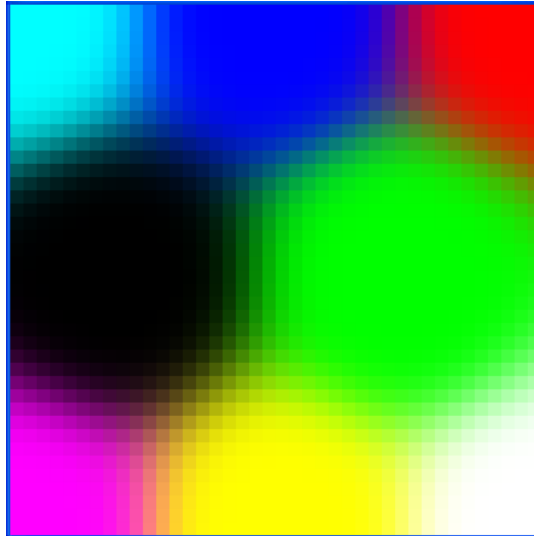
Example: Kohonen

This example creates a Kohonen network with 40 x 40 nodes. Each node has 3 weights, representing the RGB values of a color. This network is trained with 8 colors using 500 iterations. Then, the example prints out a forecast result.

Initially, the image of the nodes is:



After the training, the image is:



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

public class KohonenSOMEx1 extends KohonenSOMTrainer {

    private static int totalIter = 500, nrow = 40, ncol = 40;
    private double initialLearning = 0.07;

    public double getNeighborhoodValue(int t, double d) {
        double factor, c;

        // A Gaussian function.
        factor = Math.max(nrow, ncol) / 4.0;
        c = (double) (totalIter - t) / ((double) totalIter / factor);

        return Math.exp(-(d * d) / (2.0 * c * c));
    }

    public double getLearningCoefficient(int t) {
        return initialLearning * Math.exp(-(double) t / (double) totalIter);
    }

    public static void main(String args[]) {
        double[][] data = {
            {1.0, 0.0, 0.0}, {0.0, 1.0, 0.0},
            {0.0, 0.0, 1.0}, {1.0, 1.0, 0.0},
            {1.0, 0.0, 1.0}, {0.0, 1.0, 1.0},
            {0.0, 0.0, 0.0}, {1.0, 1.0, 1.0}
        };

        // Use a Random object to set the weights.
        Random rand = new Random(123457);
        rand.setMultiplier(16807);
```

```
        // Train the Kohonen network.
        KohonenSOM kohonen = new KohonenSOM(3, nrow, ncol);
        kohonen.setWeights(rand);

        KohonenSOMEx1 trainer = new KohonenSOMEx1();
        trainer.setIterations(totalIter);
        trainer.train(kohonen, data);

        // Get a forecast after training.
        double[] fdata = {0.25, 0.50, 0.75};
        int[] indices = kohonen.forecast(fdata);

        System.out.printf("The input (%.2f, %.2f, %.2f) has forecasted "
                + "output (%d, %d) \nwhich corresponds to the pink area on "
                + "the estimated map.\n", fdata[0], fdata[1], fdata[2],
                indices[0], indices[1]);
    }
}
```

Output

```
The input (0.25, 0.50, 0.75) has forecasted output (7, 9)
which corresponds to the pink area on the estimated map.
```

KohonenSOMTrainer class

`abstract public class com.imsl.datamining.KohonenSOMTrainer implements Serializable`

Trains a Kohonen network.

`KohonenSOMTrainer` is an abstract class with 2 abstract methods, `getNeighborhoodValue` and `getLearningCoefficient`. Therefore, a subclass of `KohonenSOMTrainer` needs to implement the `getNeighborhoodValue` and `getLearningCoefficient` methods.

Constructor

KohonenSOMTrainer

`public KohonenSOMTrainer()`

Methods

getIterations

```
public int getIterations()
```

Description

Returns the number of iterations used for training.

Returns

An `int` scalar containing the number of iterations used for training.

getLearningCoefficient

```
abstract public double getLearningCoefficient(int t)
```

Description

Returns the learning coefficient. The monotonically decreasing learning coefficient function $\alpha(t)$ is a scalar factor that defines the size of the update correction. The value of $\alpha(t)$ decreases with the step index `t`. Typical forms are linear, power, and inverse time/step. For example:

power:

$$\alpha(t) = \alpha_0 \left(\frac{\alpha_T}{\alpha_0} \right)^{t/T}$$

where t=`t`, T=the number of iterations used for training, α_0 = initial learning coefficient, α_T = final learning coefficient

inverse time:

$$\alpha(t) = \frac{A}{t+B}$$

where A and B are user determined constants

Parameter

`t` – An `int` scalar containing the current iteration of the training.

Returns

A `double` scalar containing the computed learning coefficient.

getNeighborhoodValue

```
abstract public double getNeighborhoodValue(int t, double d)
```

Description

Returns the neighborhood function value. In the simplest form, the neighborhood function $h(d,t)$ is 1 for all nodes closest to the BMU and 0 for others, but a Gaussian function is also commonly used. For example:

$h(d,t) = \exp(-d^2/2r^2)$ where r represents the neighborhood radius at index t

Parameters

`t` – An `int` scalar containing the current iteration of the training.

`d` – A `double` scalar containing the lattice distance between the best matching node and the current node.

Returns

A `double` scalar containing the computed neighborhood function value.

getNumberOfThreads

`public int getNumberOfThreads()`

Description

Returns the number of `java.lang.Thread` instances used for parallel processing.

Returns

an `int` containing the number of `java.lang.Thread` instances used for parallel processing.

setIterations

`public void setIterations(int iterations)`

Description

Sets the number of iterations to be used for training.

Parameter

`iterations` – An `int` scalar containing the number of iterations to be used for training. `iterations` must be greater than zero.
Default: `iterations` = 100.

setNumberOfThreads

`public void setNumberOfThreads(int numberOfThreads)`

Description

Sets the number of `java.lang.Thread` instances to be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the number of `java.lang.Thread` instances to be used for parallel processing.
Default: `numberOfThreads` = 1.

train

`final public void train(KohonenSOM kohonen, double[][] data)`

Description

Trains a Kohonen network.

Parameters

`kohonen` – A `KohonenSOM` object to be trained.

`data` – A `double` matrix containing the data to be used for training the Kohonen network. `data[i].length` must be equal to `dim`, the number of weights for each node, in `kohonen`.

PredictiveModel class

```
abstract public class com.imsl.datamining.PredictiveModel implements
Serializable, Cloneable
```

Specifies a predictive model. This class defines the members and methods common to predictive models in univariate prediction or classification problems.

Constructors

PredictiveModel

```
protected PredictiveModel(PredictiveModel pm)
```

Description

Constructs a `PredictiveModel` from an existing instance.

Parameter

`pm` – an instance of a `PredictiveModel`

PredictiveModel

```
protected PredictiveModel(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a `PredictiveModel` object for a single response variable and multiple predictor variables.

This constructor should be called by all classes extending `PredictiveModel`.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index in `xy` of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array of length equal to `xy[0].length` containing the type of each variable

Methods

fitModel

```
public void fitModel() throws PredictiveModel.PredictiveModelException
```

Description

Fits the predictive model to the training data (estimates the model using the training data and current configuration settings).

Subclasses of `PredictiveModel`, such as `DecisionTrees`, override this method with specific model fitting algorithms.

Exception

`PredictiveModelException` is thrown when an exception occurs in the common `PredictiveModel` methods. Implementing or overriding methods from this class may require throwing exceptions. Exceptions thrown from these methods will necessarily extend the `PredictiveModelException`.

getClassCounts

`public double[] getClassCounts()`

Description

Returns the counts of each class (level) of the categorical response variable.

If the response variable is not `com.imsl.datamining.PredictiveModel.VariableType.CATEGORICAL` (p. 1961) nor `com.imsl.datamining.PredictiveModel.VariableType.ORDERED_DISCRETE` (p. 1961) , `null` is returned.

Returns

a `double` array containing the summation of the case weights for each occurrence of a particular class found in the categorical response data.

getClassErrors

`public int[][] getClassErrors(double[] knownValues, double[] predictedValues)`

Description

Returns classification error information.

Parameters

`knownValues` – a `double` array containing the known target classifications

`predictedValues` – a `double` array containing the predicted classifications

Arrays `knownValues` and `predictedValues` must be the same length.

Returns

An `int` matrix of size `(nClasses+1)` by 2 containing the number of classification errors and the number of non-missing classifications for each target classification, plus the overall totals for these errors.

For $i <$ `nClasses`, the i-th row contains the number of classification errors for the i-th class and the number of patterns with non-missing classifications for that class. The last row contains the number of classification errors totaled over all target classifications, and the total number of patterns with non-missing target classifications.

getClassLabels

`public String[] getClassLabels()`

Description

Returns the current class labels for a categorical response variable.

Note: The labels will be null unless they have been set using the method `setClassLabels`.

Returns

a `string` array containing the labels for each class level

getClassProbabilities

```
public double[][] getClassProbabilities()
```

Description

Returns a matrix containing the predicted class probabilities for each observation in the training data

Returns

a `double` matrix containing the class probabilities

getCostMatrix

```
public double[][] getCostMatrix()
```

Description

Returns the cost matrix for a categorical response variable.

The cost matrix has elements $C(i, j)$ = cost of misclassifying a response in class j as in class i. The diagonal elements of the cost matrix must be 0. In the case that `nClasses` has not been determined (usually because `com.imsl.datamining.PredictiveModel.fitModel` (p. 1945) has not been called), an array of length zero is returned.

Returns

a square `double` matrix of dimension `nClasses` by `nClasses` containing the cost matrix for a categorical response variable, where `nClasses` is the number of classes the response variable may assume.

getMaxNumberOfCategories

```
public int getMaxNumberOfCategories()
```

Description

Returns the maximum number of categorical variables allowed.

Returns

an `int` indicating the maximum number of categorical variables allowed.

getMaxNumberOfIterations

```
public int getMaxNumberOfIterations()
```

Description

Returns the maximum number of iterations allowed for the fitting procedure or training algorithm.

Returns

an `int`, the maximum number of iterations

getNumberOfClasses

```
public int getNumberOfClasses()
```

Description

Returns the number of unique classes found in the categorical response data.

Returns

an `int` indicating the number of unique classes found in the categorical response data

getNumberOfColumns

```
public int getNumberOfColumns()
```

Description

Returns the number of columns in the training data `xy`.

Returns

an `int`, the number of columns in `xy`. If `xy` is `null`, `nCols=0`.

getNumberOfMissing

```
public int getNumberOfMissing()
```

Description

Returns the number of missing values of the response variable found in the data `xy`.

Returns

an `int`, the number of missing values

getNumberOfPredictors

```
public int getNumberOfPredictors()
```

Description

Returns the number of predictors.

Returns

an `int`, the number of predictors

getNumberOfRows

```
public int getNumberOfRows()
```

Description

Returns the number of rows in `xy` (observations).

Returns

an `int`, the number of rows in `xy` (observations)

getNumberOfUniquePredictorValues

```
public int[] getNumberOfUniquePredictorValues()
```

Description

Returns an array containing the number of distinct values of each predictor found in the input data.

For continuous predictor variables, the value is set to 0 and is not meaningful.

Returns

an `int` array containing the number of distinct values for each predictor

getPredictorIndexes

```
public int[] getPredictorIndexes()
```

Description

Returns the column indices of `xy` in which the predictor variables reside.

Returns

an `int` array containing the column indices

getPredictorTypes

```
public PredictiveModel.VariableType[] getPredictorTypes()
```

Description

Returns an array of `VariableType` objects that correspond to the predictor data types in `xy`.

Returns

a `VariableType` array that corresponds to the predictor data types in `xy`

getPrintLevel

```
public int getPrintLevel()
```

Description

Returns the current print level.

Returns

an `int`, the current print level

printLevel	Action
0	No printing.
1	Prints final results only.
2	Prints intermediate and final results.

Default: `printLevel` = 0.

getPriorProbabilities

```
public double[] getPriorProbabilities()
```

Description

Returns an array containing the prior probabilities.

Returns

a `double` array containing the prior probabilities

getRandomObject

```
public Random getRandomObject()
```

Description

Returns the random object being used in the permutation of the observations.

Returns

a `Random` object being used for permutations

getResponseColumnIndex

```
public int getResponseColumnIndex()
```

Description

Returns the column index in `xy` containing the response variable.

Returns

an `int`, the column index for the response variable

getResponseVariableAverage

```
public double getResponseVariableAverage()
```

Description

Returns the weighted average value of the response variable.

Returns

a `double`, the weighted average value of the response variable

getResponseVariableMostFrequentClass

```
public int getResponseVariableMostFrequentClass()
```

Description

Returns the most frequent value of the response variable. Only meaningful for `VariableType.CATEGORICAL` or `VariableType.ORDERED_DISCRETE`.

Returns

an `int`, the level of the most frequent class

getResponseVariableType

```
public PredictiveModel.VariableType getResponseVariableType()
```

Description

Returns the variable type of the response variable.

Returns

the `VariableType` of the response variable

---

getTotalWeight

```
public double getTotalWeight()
```

Description

Returns the sum of the active case weights.

Returns

a `double`, the sum of the active case weights

---

getVariableType

```
public PredictiveModel.VariableType[] getVariableType()
```

Description

Returns an array containing the variable types in `xy`.

Returns

a `VariableType` array containing the variable types in `xy`

---

getWeights

```
public double[] getWeights()
```

Description

Returns an array containing the case weights.

Returns

a `double` array containing the case weights

---

getXY

```
public double[][] getXY()
```

Description

Returns a copy of the `xy` data.

Returns

a `double` matrix containing the training data

---

isMustFitModel

```
public boolean isMustFitModel()
```

Description

Returns the current value of the `mustFitModel` flag.

When `true`, the `com.imsl.datamining.PredictiveModel.fitModel` (p. 1945) method should be called before doing any predictions or other analysis.

Returns

a `boolean` indicating the state of the flag

isUserFixedNClasses

```
public boolean isUserFixedNClasses()
```

Description

Returns `true` if the number of classes was fixed by the user.

Returns

a `boolean` indicating the state of the flag

predict

```
abstract public double[] predict() throws
PredictiveModel.PredictiveModelException
```

Description

Predicts the response variable using the most recent fit.

Each `PredictiveModel` subclass must override this method.

Returns

a `double` array containing the predicted values.

Exception

`PredictiveModelException` is thrown when an exception occurs in the common `PredictiveModel` methods. Implementing or overriding methods from this class may require throwing exceptions. Exceptions thrown from these methods will necessarily extend the `PredictiveModelException`.

predict

```
abstract public double[] predict(double[][] testData) throws
PredictiveModel.PredictiveModelException
```

Description

Predicts the response values using the most recent fit and the provided test data.

Each `PredictiveModel` subclass must override this method.

Parameter

`testData` – a `double` matrix containing data to be predicted. `testData` must have the same number of columns in the same arrangement as `xy` (the observations).

Returns

a `double` array containing the predicted values.

Exception

`PredictiveModelException` is thrown when an exception occurs in the common `PredictiveModel` methods. Implementing or overriding methods from this class may require throwing exceptions. Exceptions thrown from these methods will necessarily extend the `PredictiveModelException`.

predict

```
abstract public double[] predict(double[][] testData, double[] testDataWeights)
throws PredictiveModel.PredictiveModelException
```

Description

Predicts the response values using the most recent fit, the provided test data, and the test data case weights.

Each `PredictiveModel` subclass must override this method.

Parameters

`testData` – `double` matrix containing data to be predicted. `testData` must have the same number of columns in the same arrangement as `xy` (the observations).

`testDataWeights` – a `double` array containing weights for each row of `testData`.

Returns

a `double` array containing the predicted values.

Exception

`PredictiveModelException` is thrown when an exception occurs in the common `PredictiveModel` methods. Implementing or overriding methods from this class may require throwing exceptions. Exceptions thrown from these methods will necessarily extend the `PredictiveModelException`.

setClassCounts

```
public void setClassCounts(double[] classCounts)
```

Description

Sets the counts of each class of the response variable.

Use this method to set the class counts, when one or more classes do not occur in the training data due to sampling, but are otherwise valid, or when the data is distributed and the global counts are available. Only applies when the response variable is of type `com.imsl.datamining.PredictiveModel.VariableType.CATEGORICAL` (p. 1961) or `com.imsl.datamining.PredictiveModel.VariableType.ORDERED_DISCRETE` (p. 1961) .

Parameter

`classCounts` – a `double` array containing the class counts of the response variable
The default is to use the class counts discovered in the input matrix, `xy`, weighted by the values in `weights`.

setClassLabels

```
public void setClassLabels(String[] classLabels)
```

Description

Sets the class names or labels for a categorical response variable.

Parameter

`classLabels` – a `string` array containing class names or labels. The array `classLabels` must have length = `nClasses`.
Default: `classLabels` = {"1", "2", ...,"K"}, where K = `nClasses`

---

setClassProbabilities

```
public void setClassProbabilities(double[][] probs) throws
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Sets the class probabilities.

Parameter

`probs` – a `double` matrix specifying class probabilities for each pattern or observation in a data set The probabilities must range between 0.0 and 1.0 inclusive, and sum to 1.0. The number of columns in `probs` should agree with the number of classes found in the data. Otherwise an exception is thrown. Calling this method overwrites any existing values.
Default: `probs=null` unless estimated by an overriding method or set by the user.

Exception

`SumOfProbabilitiesNotOneException` is thrown when class probabilities do not sum to 1.0.

---

setConfiguration

```
abstract protected void setConfiguration(PredictiveModel pm) throws
PredictiveModel.PredictiveModelException
```

Description

Sets the configuration of `PredictiveModel` to that of the input model.

Each instance of a `PredictiveModel` must override this method. The implementation should use specific class methods to set the parameter settings to that of the input `PredictiveModel` instance, essentially creating a copy of the input model. This method is used for model parameter tuning such as is done in `CrossValidation`, where several variations of the same model are evaluated and in ensemble methods, such as `com.imsl.datamining.BootstrapAggregation` (p. 1962) , where several identical instances are fit to random samples.

Each `PredictiveModel` subclass must override this method.

Parameter

`pm` – a `PredictiveModel` object

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when exceptions occur in the enclosing class that extends `PredictiveModel`.

---

setCostMatrix

```
public void setCostMatrix(double[][] costMatrix)
```

Description

Specifies the cost matrix for a categorical response variable.

Parameter

`costMatrix` – a square `double` matrix of dimension `nClasses` by `nClasses` containing elements $C(i, j)$, the cost of misclassifying a response in class j as in class i. The diagonal elements of the cost matrix must be 0.

Both dimensions of `costMatrix` should agree with the number of classes found in the data. Otherwise an exception will be thrown.

Default: `costMatrix[i][j]`=1.0 where $i \neq j$ and `costMatrix[i][i]`=0.0.

setMaxNumberOfCategories

```
public void setMaxNumberOfCategories(int maxCategories)
```

Description

Sets the maximum number of categories allowed within categorical predictor variables.

Parameter

`maxCategories` – an `int` specifying the maximum number of categories a predictor variable can have.

Default: `maxCategories`=10

setMaxNumberOfIterations

```
public void setMaxNumberOfIterations(int maxIterations)
```

Description

Sets the maximum number of iterations allowed for the fitting procedure or training algorithm.

Most predictive models use iterative procedures to fit or train the model. Adjusting the maximum number of iterations up or down can assist in diagnosing problems.

Parameter

`maxIterations` – an `int` specifying the maximum number of iterations

Default: `maxIterations`=1000

setMustFitModel

```
public void setMustFitModel(boolean mustFitModel)
```

Description

Sets the flag of whether or not the model needs to be fit or re-estimated because of a change in the data or configuration.

Parameter

`mustFitModel` – a `boolean` giving the value of the flag

Default: `mustFitModel=true`.

setNumberOfClasses

```
public void setNumberOfClasses(int nClasses)
```

Description

Sets the number of distinct classes of the response variable. Only applies when the response variable is of type com.imsl.datamining.PredictiveModel.VariableType.CATEGORICAL (p. 1961) or com.imsl.datamining.PredictiveModel.VariableType.ORDERED_DISCRETE (p. 1961) .

Parameter

nClasses – an int representing the number of distinct classes or categories of the response variable

An error is generated if more than nClasses categories are discovered in the data.

Default: nClasses is 0.

setPredictorIndex

```
public void setPredictorIndex(int[] predIdx)
```

Description

Sets the column indices of xy in which the predictor variables reside.

This may be used to subset the full set of predictor variables (com.imsl.datamining.PredictiveModel.getPredictorTypes (p. 1949)).

Parameter

predIdx – an int array containing the column index for each predictor variable Default: All columns other than the column containing the response variable are indicated.

setPredictorTypes

```
public void setPredictorTypes(PredictiveModel.VariableType[] predVarType)
```

Description

Sets the VariableType objects that correspond to the predictor data types in xy.

Parameter

predVarType – a VariableType array of length equal to the number of predictors specifying the data type of each predictor

setPrintLevel

```
public void setPrintLevel(int printLevel)
```

Description

Sets the print level for a PredictiveModel.

Parameter

printLevel – an int specifying the level of printing to perform

printLevel	Action
0	No printing.
1	Prints final results only.
2	Prints intermediate and final results.

Default: printLevel = 0.

setPriorProbabilities

```
public void setPriorProbabilities(double[] priors) throws
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Sets the prior probabilities for class membership.

Parameter

`priors` – a `double` array specifying the prior probabilities

The prior probabilities must range between 0.0 and 1.0 inclusive, and sum to 1.0. The length of `priors` should agree with the number of classes found in the data. Otherwise an exception is thrown. Calling this method overwrites any existing values.

Default: Determined from the data.

Exception

`SumOfProbabilitiesNotOneException` is thrown when prior probabilities do not sum to 1.0.

setRandomObject

```
public void setRandomObject(Random r)
```

Description

Sets the random object to be used in the permutation of observation data.

Parameter

`r` – a `Random` object to be used in the random permutation of observation data

Specifying a seed for the `com.imsl.stat.Random` (p. 1324) object can produce repeatable/deterministic output.

setResponseColumnIndex

```
public void setResponseColumnIndex(int index)
```

Description

Sets the column index in `xy` containing the response variable.

Parameter

`index` – an `int`, the column index for the response variable

setTrainingData

```
public void setTrainingData(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Sets up the training data for the predictive model.

By calling this method, the problem is either initialized or reset to use the data in the arguments.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index in `xy` of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array of length equal to `xy[0].length` containing the type of each variable

setVariableType

```
public void setVariableType(PredictiveModel.VariableType[] varType)
```

Description

Sets the variable types for the data.

Parameter

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array of length equal to `xy[0].length` containing the type of each variable

setWeights

```
public void setWeights(double[] weights)
```

Description

Specifies the case weights.

Parameter

`weights` – a `double` array specifying case weights
Default: `weights[i]` = 1.0 for all *i*.

PredictiveModel.PredictiveModelException class

```
static public class
com.imsl.datamining.PredictiveModel.PredictiveModelException extends
com.imsl.IMSLException
```

An exception class intended to be the parent of all nested Exception classes where the enclosing class extends `PredictiveModel`. The `com.imsl.datamining.CrossValidation` (p. 1969) and `com.imsl.datamining.BootstrapAggregation` (p. 1962) classes operate on `PredictiveModel` objects. These classes, in order to maintain generality, throw this parent Exception.

Constructors

PredictiveModel.PredictiveModelException

```
public PredictiveModel.PredictiveModelException(String message)
```

Description

Constructs a `PredictiveModelException` and issues the specified message.

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

PredictiveModel.PredictiveModelException

```
public PredictiveModel.PredictiveModelException(String packageName, String key,
Object[] arguments)
```

PredictiveModel.StateChangeException class

```
static public class com.imsl.datamining.PredictiveModel.StateChangeException
extends com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Exception thrown when an input parameter has changed that might affect the model estimates or predictions.

Constructors

PredictiveModel.StateChangeException

```
public PredictiveModel.StateChangeException(String message)
```

Description

Constructs a `StateChangeException` and issues the specified message.

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

PredictiveModel.StateChangeException

```
public PredictiveModel.StateChangeException(String key, Object[] arguments)
```

Description

Constructs a `StateChangeException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameters

`key` – a `String` that contains the key of an error message in the resource bundle

`arguments` – an `Object` array containing arguments used within the error message specified by the `key`

PredictiveModel.SumOfProbabilitiesNotOneException class

`static public class com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException extends com.imsl.datamining.PredictiveModel.PredictiveModelException`

Exception thrown when the sum of probabilities is not approximately one.

Constructors

PredictiveModel.SumOfProbabilitiesNotOneException

`public PredictiveModel.SumOfProbabilitiesNotOneException(String message)`

Description

Constructs a `SumOfProbabilitiesNotOneException` and issues the specified message

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

PredictiveModel.SumOfProbabilitiesNotOneException

`public PredictiveModel.SumOfProbabilitiesNotOneException(String key, Object[] arguments)`

Description

Constructs a `SumOfProbabilitiesNotOneException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameters

`key` – a `String` that contains the key of an error message in the resource bundle

`arguments` – an `Object` array containing arguments used within the error message specified by the `key`

PredictiveModel.VariableType class

`static public final class com.imsl.datamining.PredictiveModel.VariableType extends java.lang.Enum`

Enumerates different variable types.

Type
Categorical
Ordered Discrete (Low, Med., High)
Quantitative or Continuous
Ignore this variable

Fields

CATEGORICAL

```
static final public PredictiveModel.VariableType CATEGORICAL
```

The associated variable can assume one of a limited number of values (categories).

IGNORE

```
static final public PredictiveModel.VariableType IGNORE
```

The associated variable should be ignored.

ONE_CLASS

```
static final public PredictiveModel.VariableType ONE_CLASS
```

The associated variable takes a single value.

ORDERED_DISCRETE

```
static final public PredictiveModel.VariableType ORDERED_DISCRETE
```

The associated variable can can assume a limited number of discrete, ordered values.

QUANTITATIVE_CONTINUOUS

```
static final public PredictiveModel.VariableType QUANTITATIVE_CONTINUOUS
```

The associated variable can assume any real value within a range of values.

Methods

valueOf

```
static public PredictiveModel.VariableType valueOf(String name)
```

values

```
static public PredictiveModel.VariableType[] values()
```

BootstrapAggregation class

```
public class com.imsl.datamining.BootstrapAggregation implements Serializable,
Cloneable
```

Performs bootstrap aggregation to generate predictions using predictive models.

Bootstrap aggregation, also known as *bagging*, generates predictions using predictive models. In the procedure, M bootstrap samples of size N are drawn with replacement from an original training set of size N. *Sampling with replacement* means that when an example is randomly selected, it is replaced back into the training set before the next draw. Thus a bootstrap sample can have repeated examples or observations. Using each bootstrap sample as a separate training data set, the procedure fits a predictive model and then generates predictions. For a regression problem (continuous response variable), the M predictions are combined into a single predicted value by averaging. For classification (categorical response variable), majority vote is used.

Originally proposed for decision trees, bagging leads to "improvements for unstable procedures," such as neural networks, classification and regression trees, and subset selection in linear regression. On the other hand, it can mildly degrade the performance of stable methods such as K-nearest neighbors (Breiman, 1996).

Constructor

BootstrapAggregation

```
public BootstrapAggregation(PredictiveModel pm)
```

Description

Constructs a `BootstrapAggregation` class in order to generate predictions of a `com.imsl.datamining.PredictiveModel` (p. 1945) using bootstrap aggregation.

Parameter

`pm` – a `PredictiveModel` for which the predictions are to be generated

Methods

aggregate

```
public void aggregate() throws PredictiveModel.PredictiveModelException,
NoSuchMethodException, InstantiationException, IllegalAccessException,
InvocationTargetException
```

Description

Performs the bootstrap aggregation.

Exceptions

`NoSuchMethodException` is thrown when the PredictiveModel subclass is missing a constructor with the expected signature (see `PredictiveModel (double[][], int, com.imsl.datamining.PredictiveModel. VariableType[])`).

`java.lang.InstantiationException` is thrown when an application tries to create an instance of a class using the `newInstance` method in class `Class`, but the specified class object cannot be instantiated.

`java.lang.IllegalAccessException` is thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor.

`java.lang.reflect.InvocationTargetException` is thrown when a wrapped exception is thrown by an invoked method or constructor.

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception has occurred in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

getNumberOfSamples

```
public void getNumberOfSamples(int nSamples)
```

Description

Returns the number of bootstrap samples.

Returns

an `int`, the number of bootstrap samples

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the maximum number of `java.lang.Thread` instances that may be used for parallel processing.

Returns

an `int` containing the maximum number of `java.lang.Thread` instances that may be used for parallel processing

The actual number of threads used in parallel processing will be the lesser of `numberOfThreads` and `nSamples`, the number of bootstrap samples set for bootstrap aggregation. This assessment is made to optimize use of resources.

getOutOfBagPredictionError

```
public double getOutOfBagPredictionError()
```

Description

Returns the out-of-bag mean squared prediction error for regression problems, or the out-of-bag classification percentage error for classification problems.

Returns

a `double`, the out-of-bag prediction error

Note: An out-of-bag prediction for a particular example (observation or row) is generated from only those bootstrap training sets which exclude the example. The out-of-bag predictions are done on the training data.

getOutOfBagPredictions

```
public double[] getOutOfBagPredictions()
```

Description

Returns the out-of-bag predicted values.

Returns

a `double` array containing the out-of-bag predicted values of the response variable for the examples in the training data

getPredictionError

```
public double getPredictionError()
```

Description

Returns the mean squared prediction error for regression problems, or the classification percentage error for classification problems.

Returns

a `double`, the prediction error

Note: The error is the in-sample fitted error unless the user specifies the test data using `setTestData()`.

getPredictions

```
public double[] getPredictions()
```

Description

Returns the predicted values.

Returns

a `double` array of predicted values of the response variable for the examples in the test data

To generate the predicted values, use the method `aggregate`. If `testData` is not specified, in-sample predictions are produced.

getPrintLevel

```
public int getPrintLevel()
```

Description

Returns the current print level.

Returns

an `int`, the current print level

printLevel	Action
0	No printing.
1	Prints final results only.
2	Prints intermediate and final results.

getVariableImportance

```
public double[] getVariableImportance()
```

Description

Returns the variable importance measure based on the out-of-bag prediction error.

Variable importance for a predictor is obtained by randomly permuting the out-of-bag values of the predictor and calculating the difference in predictive accuracy, before and after the permutation. The measure is averaged over all the bootstrap samples.

Returns

a `double` array containing variable importance for each predictor

isCalculateVariableImportance

```
public boolean isCalculateVariableImportance()
```

Description

Returns the boolean indicating whether or not to calculate variable importance during bootstrap aggregation.

Returns

a `boolean`, the flag indicating whether or not to calculate variable importance

setCalculateVariableImportance

```
public void setCalculateVariableImportance(boolean calculate)
```

Description

Sets the boolean to calculate variable importance.

When `true`, a permutation type variable importance measure is calculated during bootstrap aggregation.

Parameter

`calculate` – a `boolean` indicating whether or not to calculate variable importance
Default: `calculate` = false

setNumberOfSamples

```
public void setNumberOfSamples(int nSamples)
```

Description

Sets the number of bootstrap samples.

Parameter

nSamples – an int specifying the number of bootstrap samples

Default: nSamples = 50.

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the maximum number of java.lang.Thread instances that may be used for parallel processing.

Parameter

numberOfThreads – an int specifying the maximum number of java.lang.Thread instances that may be used for parallel processing

The actual number of threads used in parallel processing will be the lesser of numberOfThreads and nSamples, the number of bootstrap samples set for bootstrap aggregation. This assessment is made to optimize use of resources.

Default: numberOfThreads = 1.

setPrintLevel

```
public void setPrintLevel(int printLevel)
```

Description

Sets the print level for the predictive model.

Parameter

printLevel – An int specifying the level of printing to perform

printLevel	Action
0	No printing.
1	Prints final results only.
2	Prints intermediate and final results.

Default: printLevel = 0.

setRandomObject

```
public void setRandomObject(Random r)
```

Description

Sets a random object for the bootstrap random sampling scheme.

Parameter

r – a Random object

Default: r is created inside the code and the seed is set by the computer clock.

To obtain repeatable results, set the seed of the input r before calling this method. See com.imsl.stat.Random (p. 1324) for other options.

setTestData

```
public void setTestData(double[][] testData)
```

Description

Sets the test data to be predicted.

Parameter

`testData` – a `double` matrix containing test data for which predictions are to be made using bagging

`testData` must have the same number of columns in the same arrangement as `xy`. Missing response variable values should be indicated with `Double.NaN()`.

Default: If `testData` is not specified, in-sample predictions are produced (i.e., the original training set serves as the test data).

setTestData

`public void setTestData(double[][] testData, double[] testDataWeights)`

Description

Sets the test data to be predicted along with weights for each row in the test data.

Parameters

`testData` – a `double` matrix containing test data for which predictions are to be made using bagging

`testData` must have the same number of columns in the same arrangement as `xy`. Missing response variable values should be indicated with `Double.NaN()`.

`testDataWeights` – a `double` array containing observation weights for the test data

Default: If `testData` is not specified, in-sample predictions are produced (i.e., the original training set serves as the test data).

Example: Bootstrap Aggregation

This example illustrates bootstrap aggregation for a decision tree using a simulated data set.

```
import com.imsl.datamining.*;
import com.imsl.datamining.decisionTree.QUEST;
import com.imsl.stat.Random;

public class BootstrapAggregationEx1 {

    public static void main(String[] args) throws Exception {
        PredictiveModel.VariableType[] varType = {
            PredictiveModel.VariableType.CATEGORICAL,
            PredictiveModel.VariableType.QUANTITATIVE_CONTINUOUS,
            PredictiveModel.VariableType.CATEGORICAL,
            PredictiveModel.VariableType.CATEGORICAL
        };

        double[][] XY = {
            {2, 25.92869, 0, 0}, {1, 51.63245, 1, 1}, {1, 25.78432, 0, 2},
            {0, 39.37948, 0, 3}, {2, 24.65058, 0, 2}, {2, 45.20084, 0, 2},
            {2, 52.67960, 1, 3}, {1, 44.28342, 1, 3}, {2, 40.63523, 1, 3},
            {2, 51.76094, 0, 3}, {2, 26.30368, 0, 1}, {2, 20.70230, 1, 0},
```

```
            {2, 38.74273, 1, 3}, {2, 19.47333, 0, 0}, {1, 26.42211, 0, 0},
            {2, 37.05986, 1, 0}, {1, 51.67043, 1, 3}, {0, 42.40156, 0, 3},
            {2, 33.90027, 1, 2}, {1, 35.43282, 0, 0}, {1, 44.30369, 0, 1},
            {0, 46.72387, 0, 2}, {1, 46.99262, 0, 2}, {0, 36.05923, 0, 3},
            {2, 36.83197, 1, 1}, {1, 61.66257, 1, 2}, {0, 25.67714, 0, 3},
            {1, 39.08567, 1, 0}, {0, 48.84341, 1, 1}, {1, 39.34391, 0, 3},
            {2, 24.73522, 0, 2}, {1, 50.55251, 1, 3}, {0, 31.34263, 1, 3},
            {1, 27.15795, 1, 0}, {0, 31.72685, 0, 2}, {0, 25.00408, 0, 3},
            {1, 26.35457, 1, 3}, {2, 38.12343, 0, 1}, {0, 49.94030, 0, 2},
            {1, 42.45779, 1, 3}, {0, 38.80948, 1, 1}, {0, 43.22799, 1, 1},
            {0, 41.87624, 0, 3}, {2, 48.07820, 0, 2}, {0, 43.23673, 1, 0},
            {2, 39.41294, 0, 3}, {1, 23.93346, 0, 2}, {2, 42.84130, 1, 3},
            {2, 30.40669, 0, 1}, {0, 37.77389, 0, 2}
        };

        double[][] XYTest = {
            {0, 44.28342, 0, 2}, {0, 38.63523, 1, 3}, {2, 42.76094, 1, 3},
            {2, 20.30368, 0, 1}, {2, 25.70230, 1, 0}, {2, 38.74273, 1, 3},
            {2, 19.47333, 0, 1}
        };

        Random r = new Random(123457);
        r.setMultiplier(16807);
        QUEST dt = new QUEST(XY, 3, varType);

        dt.fitModel();
        BootstrapAggregation ba = new BootstrapAggregation(dt);
        ba.setTestData(XYTest);
        ba.setRandomObject(r);
        ba.aggregate();
        double[] predictions = ba.getPredictions();
        double MSPE = ba.getPredictionError();

        System.out.println("Actual value  Predicted value ");
        for (int k = 0; k < predictions.length; k++) {
            System.out.printf(" %3.2f \t\t %3.2f \n",
                    XYTest[k][3], predictions[k]);
        }
        System.out.printf("\n Mean squared prediction error: %3.2f \n",
                MSPE);
    }
}
```

Output

```
Actual value  Predicted value
 2.00   3.00
 3.00   3.00
 3.00   3.00
 1.00   2.00
 0.00   3.00
 3.00   3.00
 1.00   2.00

 Mean squared prediction error: 0.57
```

CrossValidation class

```
public class com.imsl.datamining.CrossValidation implements Serializable,
Cloneable
```

Performs V-Fold cross-validation for predictive models. In V-fold cross validation, the data set is partitioned randomly into V approximately equally sized sub-samples. The model is then trained V different times with each of the sub-samples removed in turn to serve as a test set. A loss or risk function is updated with the prediction errors on each fold. The total risk is averaged over the the folds and serves as a measure of the model's predictive performance. For categorical response variables (classification problems), the data can be stratified to include roughly the same proportion of each class level that occurs in the full training sample in each of the V sub-samples. See the method, `com.imsl.datamining.CrossValidation.setStratifiedCrossValidation` (p. 1973) . The cross-validated estimate of the risk function is given by

$$R^{CV}(d_k) = \frac{1}{N}\sum_{v=1}^{V}\sum_{(x_n,y_n)\in\eta_v} L(y_n, d_k^v(x_n))$$

$L(y, d(x))$ is the loss incurred when the prediction is $d(x)$ for the actual y. The inner summation is over the examples in the test sample held out for each fold.

If the predictive model is an instance of a decision tree, cross-validation is performed on each optimal sub-tree determined by cost-complexity pruning. Let the symbol η denote the full training data set, η_v the vth sub-sample. Then use d_k^v to indicate the set of predictions corresponding to the kth optimal sub-tree fitted on the training sample $\eta - \eta_v$. To select one sub-tree from among the configurations, two criteria are the minimum

$$k^* = \text{argmin}_k R^{CV}(d_k)$$

and the least complicated (smallest sub-tree) which satisfies:

$$k^{**} = \max(k) : R^{CV}(d_k) \le R^{CV}(d_{k^*}) + SE(R^{CV}(d_k))$$

The standard error is approximated by

$$SE(R^{CV}(d_k)) \approx \sqrt{\frac{s^2}{n}}$$

with

$$s^2 = \frac{1}{N}\sum_{v=1}^{V}\sum_{(x_n,y_n)\in\eta_v} (L(Y_n, d_k^v(x_n) - R^{CV}(d_k)))^2$$

The summation is over each fold and each learning sub-sample within each fold.

Constructor

CrossValidation

```
public CrossValidation(PredictiveModel pm) throws
PredictiveModel.PredictiveModelException
```

Description

Creates a `CrossValidation` object.

Parameter

`pm` – an object of a class that extends `PredictiveModel`

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception has occurred in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

Methods

crossValidate

```
public void crossValidate() throws PredictiveModel.PredictiveModelException,
NoSuchMethodException, InstantiationException, IllegalAccessException,
InvocationTargetException
```

Description

Performs V-Fold cross-validation.

Exceptions

`PredictiveModelException` is thrown when an exception occurs in the common `PredictiveModel` programming interface methods or an exception class that has extended the `PredictiveModelException` class.

`NoSuchMethodException` is thrown when the PredictiveModel subclass is missing a constructor with the expected signature (see com.imsl.datamining.PredictiveModel.PredictiveModel).

`InstantiationException` is thrown when an object fails to instantiate. This may occur if the PredictiveModel subclass is not concrete.

`IllegalAccessException` is thrown when the currently executing method does not have access to the definition of the specified class, field, method or constructor.

`java.lang.reflect.InvocationTargetException` is thrown when a wrapped exception is thrown by an invoked method or constructor.

getCrossValidatedError

```
public double getCrossValidatedError() throws
PredictiveModel.StateChangeException
```

Description

Returns the cross-validated error. If the response variable is categorical, the error is the misclassification rate, weighted by the misclassification costs and prior probabilities, attributes of the `PredictiveModel` object. If the response variable is quantitative/continuous, the error is the mean squared prediction error, also weighted if weights are set in the `PredictiveModel` object. If there are multiple model configurations, the minimum value is returned.

Returns

a `double`, the cross-validated prediction error

Exception

`com.imsl.datamining.PredictiveModel.StateChangeException` is thrown when an input parameter in the `PredictiveModel` has changed that might affect the model estimates or predictions.

getNumberOfSampleFolds

```
public int getNumberOfSampleFolds()
```

Description

Returns the number of folds set for the cross-validation.

Returns

an `int`, the number of folds

getNumberOfThreads

```
public int getNumberOfThreads()
```

Description

Returns the maximum number of `java.lang.Thread` instances that may be used for parallel processing.

Returns

an `int` containing the maximum number of `java.lang.Thread` instances that may be used for parallel processing.

The actual number of threads used in parallel processing will be the lesser of `numberOfThreads` and `nFolds`, the number of folds set for cross-validation. This assessment is made to optimize use of resources.

getRiskStandardErrors

```
public double[] getRiskStandardErrors()
```

Description

Returns the estimated standard errors for the risk values.

In most cases the length is 1. For `com.imsl.datamining.decisionTree.DecisionTree` (p. 2237) , `CrossValidation` returns an array of length >= 1.

Returns

a `double` array containing the estimated standard errors for the risk values.

---

getRiskValues

`public double[] getRiskValues()`

Description

Returns the vector of risk values.

In most cases the length is 1. For DecisionTree, `CrossValidation` returns an array of length >= 1.

Returns

a `double` array containing the estimated risk values.

---

isStratifiedCrossValidation

`public boolean isStratifiedCrossValidation()`

Description

Returns the flag to perform stratified cross-validation for a categorical response variable.

When `true`, the method `crossValidate` creates the samples to have roughly the same proportion of each class level as occurs in the full training set. When `false`, regular `V`-fold cross-validation is performed. If the response variable is continuous, the flag has no effect.

Returns

a `boolean`, the state of the flag

---

setNumberOfSampleFolds

`public void setNumberOfSampleFolds(int nFolds)`

Description

Sets the number of folds to use in cross validation.

Parameter

`nFolds` – an `int` specifying the number of folds

`nFolds` must be between 1 and the number of observations (`xy.length`), inclusive. If `nFolds` = 1 the full data set is used once to generate the `com.imsl.datamining.PredictiveModel` (p. 1945) . In other words, no cross-validation is performed. If 1 < `xy.length`/`nFolds` $\leq$ 3, leave-one-out cross validation is performed.

Default: `nFolds` = 10.

---

setNumberOfThreads

`public void setNumberOfThreads(int numberOfThreads)`

Description

Sets the maximum number of `java.lang.Thread` instances that may be used for parallel processing.

Parameter

numberOfThreads – an int specifying the maximum number of java.lang.Thread instances that may be used for parallel processing.

The actual number of threads used in parallel processing will be the lesser of numberOfThreads and nFolds, the number of folds set for cross-validation. This assessment is made to optimize use of resources.

Default: numberOfThreads = 1.

setRandomObject

```
public void setRandomObject(Random r)
```

Description

Sets the random object to be used in the permutation of observation data.

Parameter

r – a Random object to be used in random permutation of observation data.

Specifying a seed for the com.imsl.stat.Random (p. 1324) object can produce repeatable/deterministic output.

setStratifiedCrossValidation

```
public void setStratifiedCrossValidation(boolean stratify)
```

Description

Sets the flag to perform stratified cross-validation.

When true, the method crossValidate creates the samples to have roughly the same proportion of each class level as occurs in the full training set. When false, regular V-fold cross-validation is performed. If the response variable is continuous, the flag has no effect.

Parameter

stratify – a boolean indicating whether or not stratified cross-validation should be performed for categorical response variables

Default: stratify=false

Example: CrossValidation

This example applies the QUEST method to a simulated data set with 50 cases and three predictors of mixed-type. A maximally grown tree under the default controls and the optimally pruned sub-tree obtained from cross-validation and minimal cost complexity pruning are produced. Notice that the optimally pruned tree consists of just the root node, whereas the maximal tree has five nodes and three levels.

```
import com.imsl.datamining.*;
import com.imsl.datamining.decisionTree.*;
import com.imsl.stat.Random;

public class CrossValidationEx1 {
```

```
public static void main(String[] args) throws Exception {
    PredictiveModel.VariableType[] sim0VarType = {
        PredictiveModel.VariableType.CATEGORICAL,
        PredictiveModel.VariableType.QUANTITATIVE_CONTINUOUS,
        PredictiveModel.VariableType.CATEGORICAL,
        PredictiveModel.VariableType.CATEGORICAL
    };

    double[][] sim0XY = {
        {2, 25.92869, 0, 0}, {1, 51.63245, 1, 1}, {1, 25.78432, 0, 2},
        {0, 39.37948, 0, 3}, {2, 24.65058, 0, 2}, {2, 45.20084, 0, 2},
        {2, 52.67960, 1, 3}, {1, 44.28342, 1, 3}, {2, 40.63523, 1, 3},
        {2, 51.76094, 0, 3}, {2, 26.30368, 0, 1}, {2, 20.70230, 1, 0},
        {2, 38.74273, 1, 3}, {2, 19.47333, 0, 0}, {1, 26.42211, 0, 0},
        {2, 37.05986, 1, 0}, {1, 51.67043, 1, 3}, {0, 42.40156, 0, 3},
        {2, 33.90027, 1, 2}, {1, 35.43282, 0, 0}, {1, 44.30369, 0, 1},
        {0, 46.72387, 0, 2}, {1, 46.99262, 0, 2}, {0, 36.05923, 0, 3},
        {2, 36.83197, 1, 1}, {1, 61.66257, 1, 2}, {0, 25.67714, 0, 3},
        {1, 39.08567, 1, 0}, {0, 48.84341, 1, 1}, {1, 39.34391, 0, 3},
        {2, 24.73522, 0, 2}, {1, 50.55251, 1, 3}, {0, 31.34263, 1, 3},
        {1, 27.15795, 1, 0}, {0, 31.72685, 0, 2}, {0, 25.00408, 0, 3},
        {1, 26.35457, 1, 3}, {2, 38.12343, 0, 1}, {0, 49.94030, 0, 2},
        {1, 42.45779, 1, 3}, {0, 38.80948, 1, 1}, {0, 43.22799, 1, 1},
        {0, 41.87624, 0, 3}, {2, 48.07820, 0, 2}, {0, 43.23673, 1, 0},
        {2, 39.41294, 0, 3}, {1, 23.93346, 0, 2}, {2, 42.84130, 1, 3},
        {2, 30.40669, 0, 1}, {0, 37.77389, 0, 2}
    };

    Random r = new Random(123457);
    r.setMultiplier(16807);
    QUEST dt = new QUEST(sim0XY, 3, sim0VarType);

    dt.setAutoPruningFlag(true);
    dt.fitModel();
    /* print the maximal tree */
    dt.printDecisionTree(true);

    CrossValidation cv = new CrossValidation(dt);
    cv.setRandomObject(r);
    cv.crossValidate();
    double cvError = cv.getCrossValidatedError();
    double[] Rcv = cv.getRiskValues();
    double[] SERcv = cv.getRiskStandardErrors();

    System.out.println("\nTree \t Complexity\t CV Risk \t SE of CV Risk ");
    for (int k = 0; k < Rcv.length; k++) {
        System.out.printf("  %d \t  %3.2f   \t   %5.4f \t   %5.4f\n",
                k, dt.getCostComplexityValues()[k], Rcv[k], SERcv[k]);
    }
    /* prune the tree using the selected complexity value */
    dt.pruneTree(dt.getCostComplexityValues()[0]);

    System.out.printf("Minimum CV Risk Values: %5.4f\n", cvError);
    System.out.printf("Minimum CV Risk + Standard error: %5.4f\n",
            (cvError + SERcv[0]));
```

```
        /* print the pruned tree */
        dt.printDecisionTree(false);
    }
}
```

Output

```
Decision Tree:


Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes:  1  2
P(Y=0)= 0.180
P(Y=1)= 0.180
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y:   3

Node 1: Cost = 0.220, N= 17, Level = 1
   Rule: X1    <= 35.031
    P(Y=0)= 0.294
    P(Y=1)= 0.118
    P(Y=2)= 0.353
    P(Y=3)= 0.235
    Predicted Y:   2

Node 2: Cost = 0.360, N= 33, Level = 1, Child nodes:  3  4
   Rule: X1    > 35.031
    P(Y=0)= 0.121
    P(Y=1)= 0.212
    P(Y=2)= 0.212
    P(Y=3)= 0.455
    Predicted Y:   3

Node 3: Cost = 0.180, N= 19, Level = 2
      Rule: X1       <= 43.265
        P(Y=0)= 0.211
        P(Y=1)= 0.211
        P(Y=2)= 0.053
        P(Y=3)= 0.526
        Predicted Y:   3

Node 4: Cost = 0.160, N= 14, Level = 2
      Rule: X1       > 43.265
        P(Y=0)= 0.000
        P(Y=1)= 0.214
        P(Y=2)= 0.429
        P(Y=3)= 0.357
        Predicted Y:   2

Tree   Complexity  CV Risk   SE of CV Risk
  0    0.00       0.6829     0.0757
  1    0.02       0.7053     0.0807
  2    0.04       0.7281     0.0860
Minimum CV Risk Values: 0.6829
```

```
Minimum CV Risk + Standard error: 0.7586

Decision Tree:


Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes:  1  2
P(Y=0)= 0.180
P(Y=1)= 0.180
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y:   3

Node 1: Cost = 0.220, N= 17, Level = 1
   Rule: X1    <= 35.031
    P(Y=0)= 0.294
    P(Y=1)= 0.118
    P(Y=2)= 0.353
    P(Y=3)= 0.235
    Predicted Y:   2

Node 2: Cost = 0.360, N= 33, Level = 1
   Rule: X1    > 35.031
    P(Y=0)= 0.121
    P(Y=1)= 0.212
    P(Y=2)= 0.212
    P(Y=3)= 0.455
    Predicted Y:   3
Pruned at Node id 2.
```

GradientBoosting class

```
public class com.imsl.datamining.GradientBoosting extends
com.imsl.datamining.PredictiveModel implements Serializable, Cloneable
```

Performs stochastic gradient boosting for a single response variable and multiple predictor variables.

The idea behind boosting is to combine the outputs of relatively weak classifiers or predictive models to achieve iteratively better and better accuracy in either regression problems (the response variable is continuous) or classification problems (the response variable has two or more discrete values). This class implements the stochastic gradient tree boosting algorithm of Friedman, 1999. A sequence of decision trees is fit to random samples of the training data, iteratively re-weighted to minimize a specified loss function. In each iteration, pseudo-residuals are calculated based on a random sample from the original training set and the gradient of the loss function evaluated at values generated in the previous iteration. New base predictors are fit to the pseudo-residuals, and then a new prediction function is selected to minimize the loss-function, completing one iteration. The number of iterations is a parameter for the algorithm.

Gradient boosting is an ensemble method, but instead of using independent trees, gradient boosting

forms a sequence of trees, iteratively and judiciously re-weighted to minimize prediction errors. In particular, the decision tree at iteration $m+1$ is estimated on pseudo-residuals generated using the decision tree at step m. Hence, successive trees are dependent on previous trees. The algorithm in gradient boosting iterates for a fixed number of times and stops, rather than iterating until a convergence criteria is met. The number of iterations is therefore a parameter in the model. Using a randomly selected subset of the training data in each iteration has been shown to substantially improve efficiency and robustness. Thus, the method is called stochastic gradient boosting. For further discussion, see Hastie, et. al. (2008).

Constructors

GradientBoosting

`public GradientBoosting(PredictiveModel pm)`

Description

Constructs a `gradient boosting` object.

Parameter

`pm` – the `PredictiveModel` to serve as the base learner

Note: Currently only regression trees are supported as base learners.

GradientBoosting

```
public GradientBoosting(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a `GradientBoosting` object for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data

`responseColumnIndex` – an `int`, the column index for the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

fitModel

`public void fitModel() throws PredictiveModel.PredictiveModelException`

Description

Performs the gradient boosting on the training data.

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

getClassFittedValues

```
public double[][] getClassFittedValues()
```

Description

Returns the fitted values $f(x_i)$ for a categorical response variable with two or more levels.

The underlying loss function is the binomial or multinomial deviance.

Returns

a `double` matrix containing the fitted values on the training data

getClassProbabilities

```
public double[][] getClassProbabilities()
```

Description

Returns the predicted probabilities on the training data for a categorical response variable.

Returns

a `double` matrix containing the class probabilities fit on the training data. The *i,k*-th element of the matrix is the estimated probability that the observation at row index *i* belongs to the *k+1*-st class, where *k*=0,..., `nClasses`-1.

getFittedValues

```
public double[] getFittedValues()
```

Description

Returns the fitted values $f(x_i)$ for a continuous response variable after gradient boosting.

Returns

a `double` array containing the fitted values on the training data

getIterationsArray

```
public int[] getIterationsArray()
```

Description

Returns the array of different values for the number of iterations.

Different values for the number of iterations can be set and used in cross validation. See `com.imsl.datamining.GradientBoosting.setIterationsArray` (p. 1982) .

Returns

an `int` array, containing the values for the number of iterations parameter

getLossType

```
public GradientBoosting.LossFunctionType getLossType()
```

Description

Returns the current loss function type.

Returns

a LossFunctionType, the current setting of the loss function type

getLossValue

```
public double getLossValue()
```

Description

Returns the loss function value.

Returns

a double, the loss function value

getMissingTestYFlag

```
public boolean getMissingTestYFlag()
```

Description

Returns the flag that sets whether the test data is missing the response variable data.

Returns

a boolean, the flag setting whether the test data is missing the response variable values

getMultinomialResponse

```
public double[][] getMultinomialResponse()
```

Description

Returns the multinomial representation of the response variable.

Y^* is a matrix with the element at i,k, where i=0,...,nObservations-1 and k=0,...,nClasses-1

$$y^*_{ik} = \begin{cases} 1 & \text{if } y_i = k \\ 0 & \text{otherwise} \end{cases}$$

Note: This representation is not available if the response has only 2 classes (binomial).

Returns

a double matrix containing the response in multinomial representation

getNumberOfIterations

```
public int getNumberOfIterations()
```

Description

Returns the current setting for the number of iterations to use in the gradient boosting algorithm.

Different values for the number of iterations can be set and used in cross validation. See com.imsl.datamining.GradientBoosting.setIterationsArray (p. 1982) .

Returns

an `int`, the current setting for the number of iterations

getSampleSizeProportion

```
public double getSampleSizeProportion()
```

Description

Returns the current setting of the sample size proportion.

Returns

a `double`, the sample size proportion

getShrinkageParameter

```
public double getShrinkageParameter()
```

Description

Returns the current shrinkage parameter.

Returns

a `double`, the value of shrinkage parameter

getTestClassFittedValues

```
public double[][] getTestClassFittedValues()
```

Description

Returns the fitted values $f(x_i)$ for a categorical response variable with two or more levels on the test data. The underlying loss function is the binomial or multinomial deviance.

Returns

a `double` matrix containing the fitted values on the test data

getTestClassProbabilities

```
public double[][] getTestClassProbabilities()
```

Description

Returns the predicted probabilities on the test data for a categorical response variable.

Returns

a `double` matrix containing the class probabilities fit on the training data. The *i*,*k* element is the estimated probability that the *i*-th pattern belongs to the *k*-th target class, where *k*=0,...,`nClasses`-1.

getTestFittedValues

```
public double[] getTestFittedValues()
```

Description

Returns the fitted values $f(x_i)$ for a continuous response variable after gradient boosting on the test data.

Returns

a `double` array containing the fitted values on the test data

getTestLossValue

```
public double getTestLossValue()
```

Description

Returns the loss function value on the test data.

Returns

a `double`, the loss function value

predict

```
public double[] predict() throws PredictiveModel.PredictiveModelException
```

Description

Returns the predicted values on the training data.

Returns

a `double` array containing the predicted values on the training data, i.e., the fitted values

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

predict

```
public double[] predict(double[][] testData) throws
PredictiveModel.PredictiveModelException
```

Description

Returns the predicted values on the input test data.

Parameter

`testData` – a `double` matrix containing test data

Note: `testData` must have the same number of columns and the columns must be in the same arrangement as `xy`.

Returns

a `double` array containing the predicted values

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

predict

```
public double[] predict(double[][] testData, double[] testDataWeights) throws
PredictiveModel.PredictiveModelException
```

Description

Runs the gradient boosting on the training data and returns the predicted values on the weighted test data.

Parameters

`testData` – a `double` matrix containing test data

Note:`testData` must have the same number of columns and the columns must be in the same arrangement as `xy`.

`testDataWeights` – a `double` array containing weights for each row of `testData`

Returns

a `double` array containing the predicted values

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

setIterationsArray

```
public void setIterationsArray(int[] iterationsArray)
```

Description

Sets the array of different numbers of iterations.

The algorithm in gradient boosting iterates for a fixed number of times and stops, rather than iterating until a convergence criteria is met. The number of iterations is therefore a parameter in the model. After setting the `iterationsArray` to two or more values, cross-validation can be used to help determine the best choice among the values. By default, `iterationsArray` contains the single value {50}.

Parameter

`iterationsArray` – an `int` array containing the different numbers of iterations

Default: `iterationsArray` = {50}.

setLossFunctionType

```
public void setLossFunctionType(GradientBoosting.LossFunctionType lossType)
```

Description

Sets the loss function type for the gradient boosting algorithm.

Parameter

lossType – a LossFunctionType, the desired loss function type

Default: lossType=LossFunctionType.LEAST_SQUARES

setMissingTestYFlag

```
public void setMissingTestYFlag(boolean missingTestY)
```

Description

Sets the flag determining whether the test data is missing the response variable data.

Parameter

missingTestY – a boolean. When true, either the response variable data is all Double.NaN or will be treated as such.

Default: missingTestY=false

setNumberOfIterations

```
public void setNumberOfIterations(int numberOfIterations)
```

Description

Sets the number of iterations.

Parameter

numberOfIterations – an int, the number of iterations

Default: numberOfIterations = 50. The numberOfIterations must be positive.

Note: This method resets iterationsArray to an array of length 1 containing numberOfIterations.

setSampleSizeProportion

```
public void setSampleSizeProportion(double sampleSizeProportion)
```

Description

Sets the sample size proportion.

Parameter

sampleSizeProportion – a double in the interval $[0, 1]$ specifying the desired sampling proportion

Default: samplesSizeProportion = 0.50. If samplesSizeProportion = 1.0, no sampling is performed.

setShrinkageParameter

```
public void setShrinkageParameter(double shrinkageParameter)
```

Description

Sets the value of the shrinkage parameter.

Parameter

shrinkageParameter – a double in the interval $[0, 1]$ specifying the shrinkage parameter

Default: shrinkageParameter=1.0 (no shrinkage)

Example 1: Gradient Boosting

This example uses stochastic gradient boosting to obtain fitted values for a regression variable on a small data set with 6 predictor variables.

```
import com.imsl.datamining.GradientBoosting;
import com.imsl.stat.Random;

public class GradientBoostingEx1 {

    public static void main(String[] args) throws Exception {

        double[][] XY = {
            {4.45617685, 0.8587425048, 1.2705688183, 0.0, 0.0, 1.0, 0.836626959},
            {3.01895357, 0.8928761308, 1.3886538362, 2.0, 1.0, 2.0, 2.155131825},
            {5.16899757, 0.7385954093, 1.5773203815, 0.0, 4.0, 2.0, 0.075368922},
            {-0.23062048, 0.6227398487, 0.0228797458, 3.0, 4.0, 2.0, 0.070793233},
            {2.43144968, 0.8519553537, 1.2141886768, 2.0, 4.0, 2.0, 0.762200702},
            {2.28255119, 0.5578103897, 0.9185446175, 2.0, 4.0, 2.0, 0.085492814},
            {4.51650903, 0.4178302658, 1.3686663737, 0.0, 0.0, 0.0, 2.573941051},
            {5.42996967, 0.9829705667, 0.7817731784, 0.0, 5.0, 1.0, 0.865016054},
            {0.99551212, 0.3859238869, 0.2746516233, 3.0, 4.0, 0.0, 1.908151819},
            {1.23525017, 0.4165328839, 1.3154437956, 3.0, 4.0, 2.0, 2.752358041},
            {1.51599306, 0.2008399745, 0.9003028921, 3.0, 0.0, 2.0, 1.437127559},
            {2.72854297, 0.2072261081, 1.2282209327, 2.0, 5.0, 2.0, 0.68596562},
            {3.06956138, 0.9067490781, 0.8283077031, 2.0, 0.0, 2.0, 2.862403627},
            {1.81659279, 0.4506153886, 1.2822537781, 3.0, 4.0, 2.0, 1.710525684},
            {3.75978142, 0.2638894715, 0.4995447062, 0.0, 1.0, 1.0, 1.077172402},
            {5.72383445, 0.7682430062, 1.4758595745, 0.0, 3.0, 1.0, 2.365233736},
            {3.78155015, 0.6888140934, 0.4809393724, 0.0, 0.0, 1.0, 1.061246069},
            {3.60023233, 0.8470419827, 1.6149122352, 1.0, 1.0, 0.0, 0.01120048},
            {4.30238917, 0.9484412405, 1.6122899544, 1.0, 4.0, 2.0, 0.782038861},
            {-0.19206757, 0.7674867723, 0.01665624, 3.0, 5.0, 2.0, 2.924944949},
            {3.03246318, 0.8747456241, 1.6051767552, 2.0, 1.0, 0.0, 2.233971364},
            {1.56652306, 0.0947128241, 1.470864601, 3.0, 0.0, 1.0, 1.851705944},
            {2.77490671, 0.1347932827, 1.3693161067, 1.0, 2.0, 0.0, 0.795709459},
            {1.05042043, 0.258093959, 0.4679728113, 3.0, 5.0, 0.0, 2.897785557},
            {2.73366469, 0.152943752, 0.5244769375, 1.0, 4.0, 2.0, 2.712871963},
            {1.78996951, 0.7921472492, 0.4686144991, 2.0, 4.0, 1.0, 1.295327727},
            {1.10343272, 0.123231777, 0.563989053, 2.0, 4.0, 1.0, 0.510414582},
            {1.70883743, 0.1931027549, 1.8561577178, 3.0, 5.0, 1.0, 0.165721288},
            {2.17977731, 0.316932481, 1.3376214528, 2.0, 2.0, 0.0, 2.366607214},
            {2.46127675, 0.9601344266, 0.2090187217, 1.0, 3.0, 1.0, 0.846218965},
            {1.92249547, 0.1104206559, 1.739415036, 3.0, 0.0, 0.0, 0.652622544},
            {5.81907137, 0.7049566596, 1.6238740934, 0.0, 3.0, 0.0, 1.685337845},
            {2.04774497, 0.0480224835, 0.7510998738, 2.0, 5.0, 2.0, 1.400641323},
            {4.54023907, 0.0557708007, 1.0864350675, 0.0, 1.0, 1.0, 1.630408823},
            {3.66100874, 0.2939440177, 0.9709178614, 0.0, 1.0, 0.0, 0.06970193},
            {4.39253655, 0.0982369843, 1.2492676578, 0.0, 2.0, 2.0, 0.138188998},
            {3.23303353, 0.3775206071, 0.2937129182, 0.0, 0.0, 2.0, 1.070823081},
            {3.13800098, 0.7891691434, 1.90897633, 2.0, 3.0, 0.0, 1.240732062},
            {1.49034639, 0.2456938969, 0.9157859818, 3.0, 5.0, 0.0, 0.850803277},
            {0.09486277, 0.1240615626, 0.3891524528, 3.0, 5.0, 0.0, 2.532516038},
            {3.74460501, 0.0181218453, 1.4921644945, 1.0, 2.0, 1.0, 1.92839241},
            {3.24158796, 0.9203409508, 1.1644667462, 2.0, 3.0, 1.0, 1.956283022},
```

```
            {1.97796767, 0.5977597698, 0.5501609747, 2.0, 5.0, 2.0, 0.39384095},
            {4.15214037, 0.1433333508, 1.4292114358, 1.0, 0.0, 0.0, 1.114095218},
            {0.7799787, 0.8539819908, 0.7039108537, 3.0, 0.0, 1.0, 1.468978726},
            {2.01869009, 0.8919721926, 1.1436212659, 3.0, 4.0, 1.0, 2.09256257},
            {0.56311561, 0.0899261576, 0.7989077698, 3.0, 5.0, 0.0, 0.195650739},
            {4.74296429, 0.9625684835, 1.5732420743, 0.0, 3.0, 2.0, 2.685061853},
            {2.97981809, 0.5511086562, 1.6053283028, 2.0, 5.0, 2.0, 0.906810926},
            {2.82187135, 0.3869563073, 0.9321342241, 1.0, 5.0, 1.0, 0.756223386},
            {5.24390592, 0.3500950718, 1.7769328682, 0.0, 3.0, 2.0, 1.328165314},
            {3.17307157, 0.8798056154, 1.4647966106, 2.0, 5.0, 1.0, 0.561835038},
            {0.78246075, 0.1472158518, 0.4658273738, 2.0, 0.0, 0.0, 1.317240539},
            {1.57827027, 0.3415432149, 0.7513634153, 2.0, 2.0, 0.0, 1.502675544},
            {0.84104905, 0.1501226462, 0.9332020828, 3.0, 1.0, 2.0, 1.083374695},
            {2.63627352, 0.1707233109, 1.1676406977, 2.0, 3.0, 0.0, 2.236639737},
            {1.30863625, 0.2616807753, 0.8342161868, 3.0, 2.0, 2.0, 1.778402721},
            {2.7313073, 0.9616109401, 1.596915911, 3.0, 3.0, 1.0, 0.303127344},
            {3.56848173, 0.4072918599, 1.5345127448, 1.0, 2.0, 2.0, 1.47452504},
            {5.40152982, 0.7796053565, 1.3659530994, 0.0, 4.0, 1.0, 0.484531098},
            {3.94901823, 0.5052344366, 1.9319026601, 1.0, 2.0, 0.0, 2.504392843}};

      GradientBoosting.VariableType[] VarType = {
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS
        };

        GradientBoosting gb = new GradientBoosting(XY,0,VarType);
        gb.setShrinkageParameter(0.05);
        gb.setSampleSizeProportion(0.5);
        gb.setRandomObject(new Random(123457));
        gb.fitModel();
        double[] fittedValues = gb.predict();

        System.out.println("Fitted Values vs Actuals");
        for (int i = 0; i < fittedValues.length; i++) {
            System.out.printf("%5.3f, %5.3f\n", fittedValues[i], XY[i][0]);
        }
        System.out.printf("Loss Value: %5.5f\n", gb.getLossValue());
    }
}
```

Output

```
Fitted Values vs Actuals
4.341, 4.456
2.865, 3.019
4.485, 5.169
1.217, -0.231
2.757, 2.431
2.263, 2.283
4.212, 4.517
```

```
3.829, 5.430
1.177, 0.996
1.878, 1.235
1.473, 1.516
2.628, 2.729
2.352, 3.070
1.878, 1.817
3.511, 3.760
4.485, 5.724
3.551, 3.782
3.622, 3.600
3.622, 4.302
1.306, -0.192
2.902, 3.032
2.022, 1.567
3.349, 2.775
1.177, 1.050
2.648, 2.734
2.056, 1.790
1.927, 1.103
2.022, 1.709
2.628, 2.180
2.777, 2.461
2.022, 1.922
4.485, 5.819
2.039, 2.048
4.212, 4.540
3.929, 3.661
4.212, 4.393
3.511, 3.233
2.902, 3.138
1.473, 1.490
1.177, 0.095
3.493, 3.745
2.757, 3.242
1.968, 1.978
3.493, 4.152
1.306, 0.780
2.007, 2.019
1.366, 0.563
4.485, 4.743
2.813, 2.980
2.943, 2.822
4.356, 5.244
2.902, 3.173
1.927, 0.782
2.039, 1.578
1.473, 0.841
2.628, 2.636
1.473, 1.309
2.151, 2.731
3.493, 3.568
4.341, 5.402
3.533, 3.949
Loss Value: 0.35968
```

Example 2: Gradient Boosting

This example uses stochastic gradient boosting to obtain fitted probability estimates for a binary response variable and 4 predictor variables. The estimated probabilities are obtained for the training data and a small test data set. Probabilities less than or equal to 0.5 are associated with Y=0, while probabilities greater than 0.5 associate with Y=1 and would lead to these predictions on the test data.

```
import com.imsl.datamining.GradientBoosting;
import com.imsl.stat.Random;

public class GradientBoostingEx2 {

    public static void main(String[] args) throws Exception {

        GradientBoosting.VariableType[] VarType = {
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS
        };
        double[][] trainingData = {
            {0.0, 0.4223019897, 1.7540411302, 3.0, 0.763836258},
            {0.0, 0.0907259332, 0.8722643796, 2.0, 1.859006285},
            {0.0, 0.1384744535, 0.838324877, 1.0, 0.249729405},
            {1.0, 0.5435024537, 1.2359190206, 4.0, 0.831992314},
            {0.0, 0.8359154933, 1.8527500411, 1.0, 1.089201049},
            {1.0, 0.3577950741, 0.3652825342, 3.0, 2.204364955},
            {1.0, 0.6799094002, 0.6610595905, 3.0, 1.44730419},
            {0.0, 0.5821297709, 1.6180879478, 1.0, 2.957565282},
            {1.0, 0.8229457375, 1.0201675948, 3.0, 2.872570117},
            {0.0, 0.0633462721, 0.4140600134, 1.0, 0.63906323},
            {1.0, 0.1019134156, 0.0677204356, 3.0, 1.493447564},
            {0.0, 0.1551713238, 1.541201456, 3.0, 1.90219884},
            {1.0, 0.8273822817, 0.2114979578, 3.0, 2.855730173},
            {0.0, 0.7955570114, 1.8757067556, 2.0, 2.930132627},
            {0.0, 0.6537275917, 1.2139678737, 2.0, 1.535853243},
            {1.0, 0.1243124125, 1.5130919744, 4.0, 2.733670775},
            {0.0, 0.2163864174, 0.7051185896, 2.0, 2.755841087},
            {0.0, 0.2522670308, 1.2821007571, 2.0, 0.342119491},
            {0.0, 0.8677104027, 1.9003869346, 2.0, 2.454376481},
            {1.0, 0.8670932774, 0.7993045617, 4.0, 2.732812615},
            {0.0, 0.5384287981, 0.1856947718, 1.0, 1.838702635},
            {0.0, 0.7236269342, 0.4993310347, 1.0, 1.030699128},
            {0.0, 0.0789361731, 1.011216166, 1.0, 2.539607478},
            {1.0, 0.7631686032, 0.0536725423, 2.0, 1.401761686},
            {0.0, 0.1157020777, 0.0123261618, 1.0, 2.098372295},
            {1.0, 0.1451248352, 1.9153951635, 3.0, 0.492650534},
            {1.0, 0.8497178114, 1.80941298, 4.0, 2.653985489},
            {0.0, 0.8027864883, 1.2631045617, 3.0, 2.716214291},
            {0.0, 0.798560373, 0.6872106791, 2.0, 2.763023936},
            {1.0, 0.1816879204, 0.4323868025, 4.0, 0.098090197},
            {1.0, 0.6301239238, 0.3670980479, 3.0, 0.02313788},
            {1.0, 0.0411311248, 0.0173408454, 3.0, 1.994786958},
            {1.0, 0.0427366099, 0.8114635572, 3.0, 2.966069741},
```

```
{1.0, 0.4107826762, 0.1929467283, 4.0, 0.573832348},
{0.0, 0.9441903098, 0.0729898885, 1.0, 1.710992303},
{1.0, 0.3597549822, 0.2799857073, 2.0, 0.969428934},
{0.0, 0.3741368004, 1.6052779425, 2.0, 1.866030486},
{0.0, 0.3515911719, 0.3383029872, 1.0, 2.639469598},
{0.0, 0.9184092905, 1.7116801264, 1.0, 1.380178652},
{1.0, 0.77803064, 1.9830028405, 3.0, 1.834021992},
{0.0, 0.573786814, 0.0258851023, 1.0, 1.52130144},
{1.0, 0.3279244492, 0.6977945678, 4.0, 1.322451157},
{0.0, 0.7924819048, 0.3694838509, 1.0, 2.369654865},
{0.0, 0.9787846403, 1.1470323382, 2.0, 0.037156113},
{1.0, 0.6910662795, 0.1019420708, 2.0, 2.58588334},
{0.0, 0.1367050812, 0.6635301332, 2.0, 0.368273583},
{0.0, 0.2826360366, 1.4468787988, 1.0, 2.705811968},
{0.0, 0.4524727969, 0.7885378413, 2.0, 0.851228449},
{0.0, 0.5118664701, 1.061143666, 1.0, 0.249325278},
{0.0, 0.9965170731, 0.2068265025, 2.0, 0.9210639},
{1.0, 0.7801500652, 1.565742691, 4.0, 1.827419217},
{0.0, 0.2906187973, 1.7036567871, 2.0, 2.842997725},
{0.0, 0.1753704017, 0.7124397112, 2.0, 1.262811961},
{1.0, 0.7796778064, 0.3478030777, 3.0, 0.90719801},
{1.0, 0.3889356288, 1.1771452101, 4.0, 1.298438454},
{0.0, 0.9374473374, 1.1879778663, 1.0, 1.854424331},
{1.0, 0.1939157653, 0.093336341, 4.0, 0.166025681},
{1.0, 0.2023756928, 0.0623724433, 3.0, 0.536441906},
{0.0, 0.1691352043, 1.1587338657, 2.0, 2.15494096},
{1.0, 0.0921523357, 0.2247394961, 3.0, 2.006995301},
{0.0, 0.819186907, 0.0392292971, 1.0, 1.282159743},
{0.0, 0.9458126165, 1.5268264762, 1.0, 1.960050194},
{0.0, 0.1373939656, 1.8025095677, 2.0, 0.633624267},
{0.0, 0.0555424779, 0.5022063241, 2.0, 0.639495004},
{1.0, 0.3581428374, 1.4436954968, 3.0, 1.408938169},
{1.0, 0.1189418568, 0.8011626904, 4.0, 0.210266769},
{1.0, 0.5782070206, 1.58215921, 3.0, 2.648622607},
{0.0, 0.460689794, 0.0704823257, 1.0, 1.45671379},
{0.0, 0.6959878858, 0.2245675903, 2.0, 1.849515461},
{0.0, 0.1930288749, 0.6296302159, 2.0, 2.597390946},
{0.0, 0.4912149447, 0.0713489084, 1.0, 0.426487798},
{0.0, 0.3496920248, 1.0135462089, 1.0, 2.962295362},
{1.0, 0.7716284667, 0.5387295927, 4.0, 0.736709363},
{1.0, 0.3463061263, 0.7819578522, 4.0, 1.597238498},
{1.0, 0.6897138762, 1.2793166582, 4.0, 2.376281484},
{0.0, 0.2818824656, 1.4379718141, 3.0, 2.627468417},
{0.0, 0.5659798421, 1.6243568249, 1.0, 1.624809581},
{0.0, 0.7965560518, 0.3933029529, 2.0, 0.415849269},
{0.0, 0.9156922165, 1.0465683565, 1.0, 2.802914008},
{0.0, 0.8299879942, 1.2237155279, 1.0, 2.611676934},
{0.0, 0.0241912066, 1.9213823564, 1.0, 0.659596571},
{0.0, 0.0948590154, 0.3609640412, 1.0, 1.287687748},
{0.0, 0.230467916, 1.9421709292, 3.0, 2.290064565},
{0.0, 0.2209760561, 0.4812708795, 1.0, 1.862393057},
{0.0, 0.4704530933, 0.2644400774, 1.0, 1.960189529},
{1.0, 0.1986645423, 0.48924731, 2.0, 0.333790415},
{0.0, 0.9201823308, 1.4247304946, 1.0, 0.367654009},
{1.0, 0.8118424334, 0.1017034058, 2.0, 2.001390385},
{1.0, 0.1347265388, 0.1362061207, 3.0, 1.151431168},
```

```
    {0.0, 0.9884603191, 1.5700038988, 2.0, 0.717332943},
    {0.0, 0.1964012324, 0.4306495111, 1.0, 1.689056823},
    {1.0, 0.4031848807, 1.1251849262, 4.0, 1.977734922},
    {1.0, 0.0341882701, 0.3717348906, 4.0, 1.830587439},
    {0.0, 0.5073120815, 1.7860476542, 3.0, 0.142862822},
    {0.0, 0.6363195451, 0.6631249222, 2.0, 1.211148724},
    {1.0, 0.1642774614, 1.1963615627, 3.0, 0.843113448},
    {0.0, 0.0945515088, 1.8669327218, 1.0, 2.417198514},
    {0.0, 0.2364508687, 1.4035215094, 2.0, 2.964026097},
    {1.0, 0.7490112646, 0.1778408242, 4.0, 2.343119453},
    {1.0, 0.5193473259, 0.3090019161, 3.0, 1.300277323}};

double[][] testData = {
    {0.0, 0.0093314846, 0.0315045565, 1.0, 2.043737003},
    {0.0, 0.0663379349, 0.0822378928, 2.0, 1.202557951},
    {1.0, 0.9728333529, 0.8778284262, 4.0, 0.205940753},
    {1.0, 0.7655418115, 0.3292853828, 4.0, 2.940793653},
    {1.0, 0.1610695978, 0.3832762009, 4.0, 1.96753633},
    {0.0, 0.0849463812, 1.4988451041, 2.0, 2.307902221},
    {0.0, 0.7932621511, 1.2098399368, 1.0, 0.886761862},
    {0.0, 0.1336030525, 0.2794256401, 2.0, 2.672175208},
    {0.0, 0.4758480834, 0.0441179522, 1.0, 0.399722717},
    {1.0, 0.1137434335, 0.922533263, 3.0, 1.927635631}};

GradientBoosting gb = new GradientBoosting(trainingData, 0, VarType);
gb.setShrinkageParameter(0.05);
gb.setSampleSizeProportion(0.5);
gb.setRandomObject(new Random(123457));
gb.fitModel();

/* Run gradient boosting, generating fitted values and predicted values
 on the test data.
 */
gb.predict(testData);
/* Retrieve the fitted class probabilities on the training data.
 For binomial data, there will be only 1 column in the matrix.
 */
double[][] probabilities = gb.getClassProbabilities();

System.out.println("Training Data Probabilities vs Actuals");
for (int i = 0; i < probabilities.length; i++) {
    System.out.printf("%5.3f, %5.3f\n", probabilities[i][0],
            trainingData[i][0]);
}
System.out.printf("Training Data Loss  Function Value: %5.5f\n",
        gb.getLossValue());

/* Retrieve the predicted binomial probabilities on the test data. */
double[][] testProbabilities = gb.getTestClassProbabilities();
System.out.println("Test Data Probabilities vs Actuals");
for (int i = 0; i < testProbabilities.length; i++) {
    System.out.printf("%5.3f, %5.3f\n", testProbabilities[i][0],
            testData[i][0]);
}
System.out.printf("Test Data Loss Function Value: %5.5f\n",
        gb.getTestLossValue());
```

```
    }
}
```

Output

```
Training Data Probabilities vs Actuals
0.648, 0.000
0.173, 0.000
0.122, 0.000
0.784, 1.000
0.090, 0.000
0.766, 1.000
0.721, 1.000
0.090, 0.000
0.710, 1.000
0.129, 0.000
0.799, 1.000
0.660, 0.000
0.788, 1.000
0.136, 0.000
0.165, 0.000
0.763, 1.000
0.181, 0.000
0.149, 0.000
0.136, 0.000
0.802, 1.000
0.166, 0.000
0.122, 0.000
0.116, 0.000
0.253, 1.000
0.176, 0.000
0.648, 1.000
0.743, 1.000
0.673, 0.000
0.181, 0.000
0.812, 1.000
0.766, 1.000
0.799, 1.000
0.721, 1.000
0.853, 1.000
0.176, 0.000
0.240, 1.000
0.136, 0.000
0.150, 0.000
0.090, 0.000
0.648, 1.000
0.176, 0.000
0.802, 1.000
0.150, 0.000
0.173, 0.000
0.253, 1.000
0.181, 0.000
0.100, 0.000
0.181, 0.000
0.116, 0.000
```

```
0.240, 0.000
0.753, 1.000
0.136, 0.000
0.181, 0.000
0.766, 1.000
0.793, 1.000
0.116, 0.000
0.862, 1.000
0.799, 1.000
0.173, 0.000
0.788, 1.000
0.176, 0.000
0.095, 0.000
0.136, 0.000
0.181, 0.000
0.673, 1.000
0.802, 1.000
0.660, 1.000
0.176, 0.000
0.240, 0.000
0.181, 0.000
0.176, 0.000
0.116, 0.000
0.802, 1.000
0.802, 1.000
0.763, 1.000
0.673, 0.000
0.090, 0.000
0.199, 0.000
0.116, 0.000
0.111, 0.000
0.090, 0.000
0.150, 0.000
0.648, 0.000
0.129, 0.000
0.166, 0.000
0.190, 1.000
0.100, 0.000
0.253, 1.000
0.799, 1.000
0.142, 0.000
0.129, 0.000
0.793, 1.000
0.837, 1.000
0.648, 0.000
0.181, 0.000
0.710, 1.000
0.090, 0.000
0.149, 0.000
0.853, 1.000
0.788, 1.000
Training Data Loss  Function Value: 0.62967
Test Data Probabilities vs Actuals
0.176, 0.000
0.253, 0.000
0.793, 1.000
```

```
0.845, 1.000
0.821, 1.000
0.149, 0.000
0.111, 0.000
0.240, 0.000
0.176, 0.000
0.710, 1.000
Test Data Loss Function Value: 0.48412
```

Example 3: Gradient Boosting

This example applies cross-validation to the number of iterations parameter in stochastic gradient boosting. The number of iterations with the minimum cross-validated risk estimate is 50.

```
import com.imsl.datamining.CrossValidation;
import com.imsl.datamining.GradientBoosting;
import com.imsl.stat.Random;

public class GradientBoostingEx3 {

    public static void main(String[] args) throws Exception {
        GradientBoosting.VariableType[] VarType = {
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS
        };
        double[][] trainingData = {
            {0.0, 0.4223019897, 1.7540411302, 3.0, 0.763836258},
            {0.0, 0.0907259332, 0.8722643796, 2.0, 1.859006285},
            {0.0, 0.1384744535, 0.838324877, 1.0, 0.249729405},
            {1.0, 0.5435024537, 1.2359190206, 4.0, 0.831992314},
            {0.0, 0.8359154933, 1.8527500411, 1.0, 1.089201049},
            {1.0, 0.3577950741, 0.3652825342, 3.0, 2.204364955},
            {1.0, 0.6799094002, 0.6610595905, 3.0, 1.44730419},
            {0.0, 0.5821297709, 1.6180879478, 1.0, 2.957565282},
            {1.0, 0.8229457375, 1.0201675948, 3.0, 2.872570117},
            {0.0, 0.0633462721, 0.4140600134, 1.0, 0.63906323},
            {1.0, 0.1019134156, 0.0677204356, 3.0, 1.493447564},
            {0.0, 0.1551713238, 1.541201456, 3.0, 1.90219884},
            {1.0, 0.8273822817, 0.2114979578, 3.0, 2.855730173},
            {0.0, 0.7955570114, 1.8757067556, 2.0, 2.930132627},
            {0.0, 0.6537275917, 1.2139678737, 2.0, 1.535853243},
            {1.0, 0.1243124125, 1.5130919744, 4.0, 2.733670775},
            {0.0, 0.2163864174, 0.7051185896, 2.0, 2.755841087},
            {0.0, 0.2522670308, 1.2821007571, 2.0, 0.342119491},
            {0.0, 0.8677104027, 1.9003869346, 2.0, 2.454376481},
            {1.0, 0.8670932774, 0.7993045617, 4.0, 2.732812615},
            {0.0, 0.5384287981, 0.1856947718, 1.0, 1.838702635},
            {0.0, 0.7236269342, 0.4993310347, 1.0, 1.030699128},
            {0.0, 0.0789361731, 1.011216166, 1.0, 2.539607478},
            {1.0, 0.7631686032, 0.0536725423, 2.0, 1.401761686},
            {0.0, 0.1157020777, 0.0123261618, 1.0, 2.098372295},
```

```
{1.0, 0.1451248352, 1.9153951635, 3.0, 0.492650534},
{1.0, 0.8497178114, 1.80941298, 4.0, 2.653985489},
{0.0, 0.8027864883, 1.2631045617, 3.0, 2.716214291},
{0.0, 0.798560373, 0.6872106791, 2.0, 2.763023936},
{1.0, 0.1816879204, 0.4323868025, 4.0, 0.098090197},
{1.0, 0.6301239238, 0.3670980479, 3.0, 0.02313788},
{1.0, 0.0411311248, 0.0173408454, 3.0, 1.994786958},
{1.0, 0.0427366099, 0.8114635572, 3.0, 2.966069741},
{1.0, 0.4107826762, 0.1929467283, 4.0, 0.573832348},
{0.0, 0.9441903098, 0.0729898885, 1.0, 1.710992303},
{1.0, 0.3597549822, 0.2799857073, 2.0, 0.969428934},
{0.0, 0.3741368004, 1.6052779425, 2.0, 1.866030486},
{0.0, 0.3515911719, 0.3383029872, 1.0, 2.639469598},
{0.0, 0.9184092905, 1.7116801264, 1.0, 1.380178652},
{1.0, 0.77803064, 1.9830028405, 3.0, 1.834021992},
{0.0, 0.573786814, 0.0258851023, 1.0, 1.52130144},
{1.0, 0.3279244492, 0.6977945678, 4.0, 1.322451157},
{0.0, 0.7924819048, 0.3694838509, 1.0, 2.369654865},
{0.0, 0.9787846403, 1.1470323382, 2.0, 0.037156113},
{1.0, 0.6910662795, 0.1019420708, 2.0, 2.58588334},
{0.0, 0.1367050812, 0.6635301332, 2.0, 0.368273583},
{0.0, 0.2826360366, 1.4468787988, 1.0, 2.705811968},
{0.0, 0.4524727969, 0.7885378413, 2.0, 0.851228449},
{0.0, 0.5118664701, 1.061143666, 1.0, 0.249325278},
{0.0, 0.9965170731, 0.2068265025, 2.0, 0.9210639},
{1.0, 0.7801500652, 1.565742691, 4.0, 1.827419217},
{0.0, 0.2906187973, 1.7036567871, 2.0, 2.842997725},
{0.0, 0.1753704017, 0.7124397112, 2.0, 1.262811961},
{1.0, 0.7796778064, 0.3478030777, 3.0, 0.90719801},
{1.0, 0.3889356288, 1.1771452101, 4.0, 1.298438454},
{0.0, 0.9374473374, 1.1879778663, 1.0, 1.854424331},
{1.0, 0.1939157653, 0.093336341, 4.0, 0.166025681},
{1.0, 0.2023756928, 0.0623724433, 3.0, 0.536441906},
{0.0, 0.1691352043, 1.1587338657, 2.0, 2.15494096},
{1.0, 0.0921523357, 0.2247394961, 3.0, 2.006995301},
{0.0, 0.819186907, 0.0392292971, 1.0, 1.282159743},
{0.0, 0.9458126165, 1.5268264762, 1.0, 1.960050194},
{0.0, 0.1373939656, 1.8025095677, 2.0, 0.633624267},
{0.0, 0.0555424779, 0.5022063241, 2.0, 0.639495004},
{1.0, 0.3581428374, 1.4436954968, 3.0, 1.408938169},
{1.0, 0.1189418568, 0.8011626904, 4.0, 0.210266769},
{1.0, 0.5782070206, 1.58215921, 3.0, 2.648622607},
{0.0, 0.460689794, 0.0704823257, 1.0, 1.45671379},
{0.0, 0.6959878858, 0.2245675903, 2.0, 1.849515461},
{0.0, 0.1930288749, 0.6296302159, 2.0, 2.597390946},
{0.0, 0.4912149447, 0.0713489084, 1.0, 0.426487798},
{0.0, 0.3496920248, 1.0135462089, 1.0, 2.962295362},
{1.0, 0.7716284667, 0.5387295927, 4.0, 0.736709363},
{1.0, 0.3463061263, 0.7819578522, 4.0, 1.597238498},
{1.0, 0.6897138762, 1.2793166582, 4.0, 2.376281484},
{0.0, 0.2818824656, 1.4379718141, 3.0, 2.627468417},
{0.0, 0.5659798421, 1.6243568249, 1.0, 1.624809581},
{0.0, 0.7965560518, 0.3933029529, 2.0, 0.415849269},
{0.0, 0.9156922165, 1.0465683565, 1.0, 2.802914008},
{0.0, 0.8299879942, 1.2237155279, 1.0, 2.611676934},
{0.0, 0.0241912066, 1.9213823564, 1.0, 0.659596571},
```

```
            {0.0, 0.0948590154, 0.3609640412, 1.0, 1.287687748},
            {0.0, 0.230467916, 1.9421709292, 3.0, 2.290064565},
            {0.0, 0.2209760561, 0.4812708795, 1.0, 1.862393057},
            {0.0, 0.4704530933, 0.2644400774, 1.0, 1.960189529},
            {1.0, 0.1986645423, 0.48924731, 2.0, 0.333790415},
            {0.0, 0.9201823308, 1.4247304946, 1.0, 0.367654009},
            {1.0, 0.8118424334, 0.1017034058, 2.0, 2.001390385},
            {1.0, 0.1347265388, 0.1362061207, 3.0, 1.151431168},
            {0.0, 0.9884603191, 1.5700038988, 2.0, 0.717332943},
            {0.0, 0.1964012324, 0.4306495111, 1.0, 1.689056823},
            {1.0, 0.4031848807, 1.1251849262, 4.0, 1.977734922},
            {1.0, 0.0341882701, 0.3717348906, 4.0, 1.830587439},
            {0.0, 0.5073120815, 1.7860476542, 3.0, 0.142862822},
            {0.0, 0.6363195451, 0.6631249222, 2.0, 1.211148724},
            {1.0, 0.1642774614, 1.1963615627, 3.0, 0.843113448},
            {0.0, 0.0945515088, 1.8669327218, 1.0, 2.417198514},
            {0.0, 0.2364508687, 1.4035215094, 2.0, 2.964026097},
            {1.0, 0.7490112646, 0.1778408242, 4.0, 2.343119453},
            {1.0, 0.5193473259, 0.3090019161, 3.0, 1.300277323}};

        GradientBoosting gb = new GradientBoosting(trainingData, 4, VarType);

        int[] cvIterations = {10, 30, 50, 100, 500};
        gb.setIterationsArray(cvIterations);
        gb.setShrinkageParameter(0.05);
        gb.setSampleSizeProportion(0.5);
        gb.setRandomObject(new Random(123457));

        CrossValidation cv = new CrossValidation(gb);
        cv.setRandomObject(new Random(123457));
        cv.crossValidate();
        double cvError = cv.getCrossValidatedError();
        double[] Rcv = cv.getRiskValues();
        double[] SERcv = cv.getRiskStandardErrors();

        System.out.println("\nModel \t Number Of Iterations\t   CV Risk       "
                + "SE of CV Risk ");
        for (int k = 0; k < Rcv.length; k++) {
            System.out.printf("  %d            %3d                %6.2f          %7.2f\n",
                    k, cvIterations[k], Rcv[k], SERcv[k]);
        }

        System.out.printf("Minimum CV Risk Values: %5.4f\n", cvError);
        System.out.printf("Minimum CV Risk + Standard error: %5.4f\n",
                (cvError + SERcv[0]));
    }
}
```

Output

```
Model   Number Of Iterations    CV Risk       SE of CV Risk
  0           10                 106.89          926.69
  1           30                 102.32          850.83
  2           50                  98.43          785.00
```

```
  3           100                111.95         1016.45
  4           500                124.51         1257.52
Minimum CV Risk Values: 98.4270
Minimum CV Risk + Standard error: 1025.1126
```

Example 4: Gradient Boosting

This example illustrates using the alternative constructor for gradient boosting. This constructor accepts an instance of a regression tree to serve as the base learner and allows for more flexibility in controlling the base learner configuration.

```
import com.imsl.datamining.GradientBoosting;
import com.imsl.datamining.decisionTree.ALACART;
import com.imsl.stat.Random;

public class GradientBoostingEx4 {
     public static void main(String[] args) throws Exception {

        GradientBoosting.VariableType[] VarType = {
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS,
            GradientBoosting.VariableType.CATEGORICAL,
            GradientBoosting.VariableType.QUANTITATIVE_CONTINUOUS
        };
        double[][] trainingData = {
            {0.0, 0.4223019897, 1.7540411302, 3.0, 0.763836258},
            {0.0, 0.0907259332, 0.8722643796, 2.0, 1.859006285},
            {0.0, 0.1384744535, 0.838324877, 1.0, 0.249729405},
            {1.0, 0.5435024537, 1.2359190206, 4.0, 0.831992314},
            {0.0, 0.8359154933, 1.8527500411, 1.0, 1.089201049},
            {1.0, 0.3577950741, 0.3652825342, 3.0, 2.204364955},
            {1.0, 0.6799094002, 0.6610595905, 3.0, 1.44730419},
            {0.0, 0.5821297709, 1.6180879478, 1.0, 2.957565282},
            {1.0, 0.8229457375, 1.0201675948, 3.0, 2.872570117},
            {0.0, 0.0633462721, 0.4140600134, 1.0, 0.63906323},
            {1.0, 0.1019134156, 0.0677204356, 3.0, 1.493447564},
            {0.0, 0.1551713238, 1.541201456, 3.0, 1.90219884},
            {1.0, 0.8273822817, 0.2114979578, 3.0, 2.855730173},
            {0.0, 0.7955570114, 1.8757067556, 2.0, 2.930132627},
            {0.0, 0.6537275917, 1.2139678737, 2.0, 1.535853243},
            {1.0, 0.1243124125, 1.5130919744, 4.0, 2.733670775},
            {0.0, 0.2163864174, 0.7051185896, 2.0, 2.755841087},
            {0.0, 0.2522670308, 1.2821007571, 2.0, 0.342119491},
            {0.0, 0.8677104027, 1.9003869346, 2.0, 2.454376481},
            {1.0, 0.8670932774, 0.7993045617, 4.0, 2.732812615},
            {0.0, 0.5384287981, 0.1856947718, 1.0, 1.838702635},
            {0.0, 0.7236269342, 0.4993310347, 1.0, 1.030699128},
            {0.0, 0.0789361731, 1.011216166, 1.0, 2.539607478},
            {1.0, 0.7631686032, 0.0536725423, 2.0, 1.401761686},
            {0.0, 0.1157020777, 0.0123261618, 1.0, 2.098372295},
            {1.0, 0.1451248352, 1.9153951635, 3.0, 0.492650534},
            {1.0, 0.8497178114, 1.80941298, 4.0, 2.653985489},
            {0.0, 0.8027864883, 1.2631045617, 3.0, 2.716214291},
```

```
{0.0, 0.798560373, 0.6872106791, 2.0, 2.763023936},
{1.0, 0.1816879204, 0.4323868025, 4.0, 0.098090197},
{1.0, 0.6301239238, 0.3670980479, 3.0, 0.02313788},
{1.0, 0.0411311248, 0.0173408454, 3.0, 1.994786958},
{1.0, 0.0427366099, 0.8114635572, 3.0, 2.966069741},
{1.0, 0.4107826762, 0.1929467283, 4.0, 0.573832348},
{0.0, 0.9441903098, 0.0729898885, 1.0, 1.710992303},
{1.0, 0.3597549822, 0.2799857073, 2.0, 0.969428934},
{0.0, 0.3741368004, 1.6052779425, 2.0, 1.866030486},
{0.0, 0.3515911719, 0.3383029872, 1.0, 2.639469598},
{0.0, 0.9184092905, 1.7116801264, 1.0, 1.380178652},
{1.0, 0.77803064, 1.9830028405, 3.0, 1.834021992},
{0.0, 0.573786814, 0.0258851023, 1.0, 1.52130144},
{1.0, 0.3279244492, 0.6977945678, 4.0, 1.322451157},
{0.0, 0.7924819048, 0.3694838509, 1.0, 2.369654865},
{0.0, 0.9787846403, 1.1470323382, 2.0, 0.037156113},
{1.0, 0.6910662795, 0.1019420708, 2.0, 2.58588334},
{0.0, 0.1367050812, 0.6635301332, 2.0, 0.368273583},
{0.0, 0.2826360366, 1.4468787988, 1.0, 2.705811968},
{0.0, 0.4524727969, 0.7885378413, 2.0, 0.851228449},
{0.0, 0.5118664701, 1.061143666, 1.0, 0.249325278},
{0.0, 0.9965170731, 0.2068265025, 2.0, 0.9210639},
{1.0, 0.7801500652, 1.565742691, 4.0, 1.827419217},
{0.0, 0.2906187973, 1.7036567871, 2.0, 2.842997725},
{0.0, 0.1753704017, 0.7124397112, 2.0, 1.262811961},
{1.0, 0.7796778064, 0.3478030777, 3.0, 0.90719801},
{1.0, 0.3889356288, 1.1771452101, 4.0, 1.298438454},
{0.0, 0.9374473374, 1.1879778663, 1.0, 1.854424331},
{1.0, 0.1939157653, 0.093336341, 4.0, 0.166025681},
{1.0, 0.2023756928, 0.0623724433, 3.0, 0.536441906},
{0.0, 0.1691352043, 1.1587338657, 2.0, 2.15494096},
{1.0, 0.0921523357, 0.2247394961, 3.0, 2.006995301},
{0.0, 0.819186907, 0.0392292971, 1.0, 1.282159743},
{0.0, 0.9458126165, 1.5268264762, 1.0, 1.960050194},
{0.0, 0.1373939656, 1.8025095677, 2.0, 0.633624267},
{0.0, 0.0555424779, 0.5022063241, 2.0, 0.639495004},
{1.0, 0.3581428374, 1.4436954968, 3.0, 1.408938169},
{1.0, 0.1189418568, 0.8011626904, 4.0, 0.210266769},
{1.0, 0.5782070206, 1.58215921, 3.0, 2.648622607},
{0.0, 0.460689794, 0.0704823257, 1.0, 1.45671379},
{0.0, 0.6959878858, 0.2245675903, 2.0, 1.849515461},
{0.0, 0.1930288749, 0.6296302159, 2.0, 2.597390946},
{0.0, 0.4912149447, 0.0713489084, 1.0, 0.426487798},
{0.0, 0.3496920248, 1.0135462089, 1.0, 2.962295362},
{1.0, 0.7716284667, 0.5387295927, 4.0, 0.736709363},
{1.0, 0.3463061263, 0.7819578522, 4.0, 1.597238498},
{1.0, 0.6897138762, 1.2793166582, 4.0, 2.376281484},
{0.0, 0.2818824656, 1.4379718141, 3.0, 2.627468417},
{0.0, 0.5659798421, 1.6243568249, 1.0, 1.624809581},
{0.0, 0.7965560518, 0.3933029529, 2.0, 0.415849269},
{0.0, 0.9156922165, 1.0465683565, 1.0, 2.802914008},
{0.0, 0.8299879942, 1.2237155279, 1.0, 2.611676934},
{0.0, 0.0241912066, 1.9213823564, 1.0, 0.659596571},
{0.0, 0.0948590154, 0.3609640412, 1.0, 1.287687748},
{0.0, 0.230467916, 1.9421709292, 3.0, 2.290064565},
{0.0, 0.2209760561, 0.4812708795, 1.0, 1.862393057},
```

```
    {0.0, 0.4704530933, 0.2644400774, 1.0, 1.960189529},
    {1.0, 0.1986645423, 0.48924731, 2.0, 0.333790415},
    {0.0, 0.9201823308, 1.4247304946, 1.0, 0.367654009},
    {1.0, 0.8118424334, 0.1017034058, 2.0, 2.001390385},
    {1.0, 0.1347265388, 0.1362061207, 3.0, 1.151431168},
    {0.0, 0.9884603191, 1.5700038988, 2.0, 0.717332943},
    {0.0, 0.1964012324, 0.4306495111, 1.0, 1.689056823},
    {1.0, 0.4031848807, 1.1251849262, 4.0, 1.977734922},
    {1.0, 0.0341882701, 0.3717348906, 4.0, 1.830587439},
    {0.0, 0.5073120815, 1.7860476542, 3.0, 0.142862822},
    {0.0, 0.6363195451, 0.6631249222, 2.0, 1.211148724},
    {1.0, 0.1642774614, 1.1963615627, 3.0, 0.843113448},
    {0.0, 0.0945515088, 1.8669327218, 1.0, 2.417198514},
    {0.0, 0.2364508687, 1.4035215094, 2.0, 2.964026097},
    {1.0, 0.7490112646, 0.1778408242, 4.0, 2.343119453},
    {1.0, 0.5193473259, 0.3090019161, 3.0, 1.300277323}};

double[][] testData = {
    {0.0, 0.0093314846, 0.0315045565, 1.0, 2.043737003},
    {0.0, 0.0663379349, 0.0822378928, 2.0, 1.202557951},
    {1.0, 0.9728333529, 0.8778284262, 4.0, 0.205940753},
    {1.0, 0.7655418115, 0.3292853828, 4.0, 2.940793653},
    {1.0, 0.1610695978, 0.3832762009, 4.0, 1.96753633},
    {0.0, 0.0849463812, 1.4988451041, 2.0, 2.307902221},
    {0.0, 0.7932621511, 1.2098399368, 1.0, 0.886761862},
    {0.0, 0.1336030525, 0.2794256401, 2.0, 2.672175208},
    {0.0, 0.4758480834, 0.0441179522, 1.0, 0.399722717},
    {1.0, 0.1137434335, 0.922533263, 3.0, 1.927635631}};

ALACART dTree= new ALACART(trainingData,0,VarType);

dTree.setMaxDepth(10);
dTree.setMinObsPerNode(10);
dTree.setMaxNodes(4);

GradientBoosting gb = new GradientBoosting(dTree);
gb.setShrinkageParameter(0.05);
gb.setSampleSizeProportion(0.5);
gb.setRandomObject(new Random(123457));
gb.fitModel();

/* Run gradient boosting, generating fitted values and predicted values
 on the test data.
*/
gb.predict(testData);
/* Retrieve the fitted binomial probabilities on the training data. */
double[][] probabilities = gb.getClassProbabilities();

System.out.println("Training Data Probabilities vs Actuals");
for (int i = 0; i < probabilities.length; i++) {
    System.out.printf("%5.3f, %5.3f\n", probabilities[i][0],
            trainingData[i][0]);
}
System.out.printf("Training Data Loss  Function Value: %5.5f\n",
        gb.getLossValue());
```

```
        /* Retrieve the predicted binomial probabilities on the test data. */
        double[][] testProbabilities = gb.getTestClassProbabilities();
        System.out.println("Test Data Probabilities vs Actuals");
        for (int i = 0; i < testProbabilities.length; i++) {
            System.out.printf("%5.3f, %5.3f\n", testProbabilities[i][0],
                    testData[i][0]);
        }
        System.out.printf("Test Data Loss Function Value: %5.5f\n",
                gb.getTestLossValue());
    }
}
```

Output

```
Training Data Probabilities vs Actuals
0.648, 0.000
0.173, 0.000
0.122, 0.000
0.784, 1.000
0.090, 0.000
0.766, 1.000
0.721, 1.000
0.090, 0.000
0.710, 1.000
0.129, 0.000
0.799, 1.000
0.660, 0.000
0.788, 1.000
0.136, 0.000
0.165, 0.000
0.763, 1.000
0.181, 0.000
0.149, 0.000
0.136, 0.000
0.802, 1.000
0.166, 0.000
0.122, 0.000
0.116, 0.000
0.253, 1.000
0.176, 0.000
0.648, 1.000
0.743, 1.000
0.673, 0.000
0.181, 0.000
0.812, 1.000
0.766, 1.000
0.799, 1.000
0.721, 1.000
0.853, 1.000
0.176, 0.000
0.240, 1.000
0.136, 0.000
0.150, 0.000
0.090, 0.000
0.648, 1.000
```

```
0.176, 0.000
0.802, 1.000
0.150, 0.000
0.173, 0.000
0.253, 1.000
0.181, 0.000
0.100, 0.000
0.181, 0.000
0.116, 0.000
0.240, 0.000
0.753, 1.000
0.136, 0.000
0.181, 0.000
0.766, 1.000
0.793, 1.000
0.116, 0.000
0.862, 1.000
0.799, 1.000
0.173, 0.000
0.788, 1.000
0.176, 0.000
0.095, 0.000
0.136, 0.000
0.181, 0.000
0.673, 1.000
0.802, 1.000
0.660, 1.000
0.176, 0.000
0.240, 0.000
0.181, 0.000
0.176, 0.000
0.116, 0.000
0.802, 1.000
0.802, 1.000
0.763, 1.000
0.673, 0.000
0.090, 0.000
0.199, 0.000
0.116, 0.000
0.111, 0.000
0.090, 0.000
0.150, 0.000
0.648, 0.000
0.129, 0.000
0.166, 0.000
0.190, 1.000
0.100, 0.000
0.253, 1.000
0.799, 1.000
0.142, 0.000
0.129, 0.000
0.793, 1.000
0.837, 1.000
0.648, 0.000
0.181, 0.000
0.710, 1.000
```

```
0.090, 0.000
0.149, 0.000
0.853, 1.000
0.788, 1.000
Training Data Loss  Function Value: 0.62967
Test Data Probabilities vs Actuals
0.176, 0.000
0.253, 0.000
0.793, 1.000
0.845, 1.000
0.821, 1.000
0.149, 0.000
0.111, 0.000
0.240, 0.000
0.176, 0.000
0.710, 1.000
Test Data Loss Function Value: 0.48412
```

GradientBoosting.LossFunctionType class

```
static public final class com.imsl.datamining.GradientBoosting.LossFunctionType
extends java.lang.Enum
```

The loss function type as specified by the error measure.

Fields

ADABOOST

`static final public GradientBoosting.LossFunctionType ADABOOST`

The loss criteria is the AdaBoost.M1 criterion. The loss function
$L(y_i, f(x_i)) = \sum \exp\left(-(2y_i - 1)f(x_i)\right)$

BERNOULLI

`static final public GradientBoosting.LossFunctionType BERNOULLI`

The loss criteria is the binomial or Bernoulli negative log-likelihood, or deviance.
$L(y_i, f(x_i)) = -2\sum\left(y_i f(x_i) - \log(1 + \exp(f(x_i)))\right)$

HUBER_M

`static final public GradientBoosting.LossFunctionType HUBER_M`

The loss criteria is the Huber-M weighted squared error and absolute deviation error with parameter α. That is, the loss function

$L(y_i, f(x_i)) = \sum \Psi(y_i, f(x_i))$where

$$\Psi(y, f(x)) = \begin{cases} (y - f(x))^2 & \text{for } |y - f(x)| \leq \delta \\ 2\delta(|y - f(x)| - \delta/2) & \text{for } |y - f(x)| > \delta \end{cases}$$

And where δ is the α -empirical quantile of the residuals, $y_i - f(x_i), i = 1, ..., n$

LEAST_ABSOLUTE_DEVIATION

```
static final public GradientBoosting.LossFunctionType LEAST_ABSOLUTE_DEVIATION
```

The loss criteria is least absolute deviation error. That is, the loss function

$L(y_i, f(x_i)) = \sum |(y_i - f(x_i))|$

LEAST_SQUARES

```
static final public GradientBoosting.LossFunctionType LEAST_SQUARES
```

The loss criteria is least squared error. That is, the loss function

$L(y_i, f(x_i)) = \sum (y_i - f(x_i))^2$

MULTINOMIAL_DEVIANCE

```
static final public GradientBoosting.LossFunctionType MULTINOMIAL_DEVIANCE
```

The loss criteria is the (K-class) multinomial negative log-likelihood, or multinomial deviance. The loss function

$L(y_i, f(x_i)) = -2 \sum_i \sum_k y_{ik} \log(p_{ik})$where

$p_{ik} = \frac{\exp f_k(x_i)}{(\sum_k \exp f_k(x_i))}$

Methods

valueOf

```
static public GradientBoosting.LossFunctionType valueOf(String name)
```

values

```
static public GradientBoosting.LossFunctionType[] values()
```

Chapter 30: Support Vector Machines

Types

class SupportVectorMachine . . . 2003
class SVClassification . . . 2013
class SVOneClass . . . 2025
class SVRegression . . . 2029
class Kernel . . . 2037
class LinearKernel . . . 2039
class SigmoidKernel . . . 2040
class RadialBasisKernel . . . 2042
class PolynomialKernel . . . 2044
class DataNode . . . 2045

SupportVectorMachine class

```
abstract public class
com.imsl.datamining.supportvectormachine.SupportVectorMachine extends
com.imsl.datamining.PredictiveModel
```

Abstract class for generating a support vector machine.

Support vector machines are machine learning algorithms for classification, regression, and the so-called one class problem. This package contains `SVClassification`, `SVRegression`, and `SVOneClass` for each of these problems, respectively. Each formulation uses a kernel based optimization to find support vectors that optimally separate clouds of high dimensional data for the purpose of making predictions. `SupportVectorMachine` and all of the classes in this package are based on the `LIBSVM` package of Chang, Chih-Chung; Lin, Chih-Jen (2011).

Constructors

SupportVectorMachine

```
public SupportVectorMachine(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a support vector machine for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

SupportVectorMachine

```
public SupportVectorMachine(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType, Kernel k) throws
CloneNotSupportedException
```

Description

Constructs a support vector machine for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

`k` – a `Kernel`, the kernel function

Exception

`java.lang.CloneNotSupportedException` thrown to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface

Methods

fitModel

```
public void fitModel() throws PredictiveModel.PredictiveModelException
```

Description

Fits the model to the training data, i.e, trains the support vector machine.

Exception

`PredictiveModel.PredictiveModelException` an exception has occurred in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException, com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException, and com.imsl.datamining.supportvectormachine.SupportVectorMachine.ReflectiveOperationException.

getConvergenceTolerance

```
public double getConvergenceTolerance()
```

Description

Returns the convergence tolerance.

Returns

a `double`, the tolerance value

getKernel

```
protected Kernel getKernel()
```

Description

Returns the kernel object being used in the optimization.

Returns

a `Kernel` object

getKernelParameters

```
public double[] getKernelParameters()
```

Description

Returns the kernel parameters. See documentation for each `Kernel` class for specifications of the parameters.

Returns

a `double` array containing the kernel parameter values

getModel

```
protected SVModel getModel()
```

Description

Returns the model object.

Returns

an `SVModel` object

getNuParameter

```
public double getNuParameter()
```

Description

Returns the value of the ν parameter.

Returns

a `double`, the current value of the ν parameter

getRegularizationParameter

```
public double getRegularizationParameter()
```

Description

Returns the value of the regularization parameter, C.

Returns

a `double`, the value of the regularization parameter

getWorkingArraySize

```
public double getWorkingArraySize()
```

Description

Returns the setting for the work array size.

Returns

a `double`, the number of megabytes allocated to the work array used in the decomposition method

isNuFormulation

```
public boolean isNuFormulation()
```

Description

Returns the boolean to perform the ν-formulation of the optimization problem. When `true`, the ν-formulation of the optimization problem is solved. When `false`, the standard or default formulation is solved.

Returns

a `boolean`, the current value of the flag

isProbability

```
public boolean isProbability()
```

Description

Returns the boolean to calculate probability estimates.

Returns

a `boolean`, indicating whether or not probability estimates for each class level should be computed

isShrinking

```
public boolean isShrinking()
```

Description

Returns the boolean to perform shrinking during optimization. When `true`, the optimization performs shrinking.

Returns

a `boolean`, the value of the `shrinking` flag

optimize

```
abstract protected SVModel optimize(DataNode[][] x, double[] y, double[] w, int
len, Kernel kernel) throws NoSuchMethodException, InstantiationException,
IllegalAccessException, InvocationTargetException
```

Description

Abstract method to perform the support vector machine optimization.

Parameters

- `x` – a `DataNode` matrix containing the attribute data
- `y` – a `double` array containing the response variable
- `len` – an `int`, the total possible number of support vectors
- `w` – a `double` array containing the observation weights
- `kernel` – an `Kernel` object

Returns

an `SVModel` structure containing the fitted model

Exceptions

- `java.lang.NoSuchMethodException` thrown when a particular method cannot be found
- `java.lang.InstantiationException` thrown when an application tries to create an instance of a class using the `newInstance` method in class `Class`, but the specified class object cannot be instantiated
- `java.lang.IllegalAccessException` thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor
- `java.lang.reflect.InvocationTargetException` a checked exception that wraps an exception thrown by an invoked method or constructor

predict

```
public double[] predict() throws
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Returns the predicted values on the training data, i.e., returns the fitted values.

Returns

a `double` array containing the fitted values on the training data

Exception

`SumOfProbabilitiesNotOneException` the sum of probabilities is not approximately one

predict

```
public double[] predict(double[][] testData) throws
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Returns the predicted values on the input test data.

Parameter

`testData` – a `double` matrix containing test data

Note: `testData` must have the same number of columns as `xy` and the columns must be in the same arrangement as in `xy`.

Returns

a `double` array containing the predicted values

Exception

`SumOfProbabilitiesNotOneException` the sum of probabilities is not approximately one

predictValues

```
abstract protected double[] predictValues(SVModel model, double[][]
attributeData) throws PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Abstract method for generating the predicted values using the fitted support vector machine model.

Parameters

`model` – a fitted `SVModel` object

`attributeData` – a `double` matrix containing the attribute (or predictor) data

Returns

a `double` array containing the predictions for each row in the input attribute data

Exception

`com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException` the sum of probabilities is not approximately one

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
SupportVectorMachine.CloneNotSupportedException
```

Description

Sets the configuration of `PredictiveModel` to that of the input model.

Note that the input `PredictiveModel` object must be the same subclass of `PredictiveModel` as is this instance.

Parameter

`pm` – a `PredictiveModel` object which is to have its attributes duplicated in this instance

Exception

`CloneNotSupportedException` a `java.lang.CloneNotSupportedException` has occurred. The original exception has been added to the `SupportVectorMachine.CloneNotSupportedException` as a suppressed exception.

Default: The class uses its default configuration as described in the different methods.

setConvergenceTolerance

```
public void setConvergenceTolerance(double eps)
```

Description

Sets the convergence tolerance.

Parameter

`eps` – a `double`, the tolerance value

Default: `eps`=0.001

setKernel

```
public void setKernel(Kernel kernel)
```

Description

Sets the kernel to be used in the optimization.

See the class `com.imsl.datamining.supportvectormachine.Kernel` (p. 2037) for details.

Parameter

`kernel` – an instance of class `Kernel`

Default: kernel=`RadialBasisKernel`

setKernelParameters

```
public void setKernelParameters(double[] kParams)
```

Description

Sets the kernel parameters. See documentation for each `Kernel` class for specifications of the parameters.

Parameter

`kParams` – a `double` array containing the kernel parameter values

setNuFormulation

```
public void setNuFormulation(boolean nuFormulation)
```

Description

Sets the boolean to perform the ν-formulation of the optimization problem. When `true`, the ν-formulation of the optimization problem is solved. When `false`, the standard formulation is solved.

The ν-support vector classification (ν-SVC) algorithm presents a new parameter $\nu \in (0,1]$ which acts as an upper bound on the fraction of training errors and a lower bound on the fraction of support vectors.

The primal optimization problem for the binary variable $y \in \{1,-1\}$ is

$$\min_{w,b,\xi,\rho} \frac{1}{2} w^T w - \nu\rho + \frac{1}{l} \sum_{i=1}^{l} \xi_i$$

$$\text{subject to} \, y_i(w^T \phi(x_i) + b) \geq \rho - \xi_i,$$

$$\xi_i \geq 0, i = 1, \ldots, l, \rho \geq 0$$

Similar to ν-SVC, in ν-support vector regression (ν-SVR) the parameter $\nu \in (0,1]$ controls the number of support vectors. The ν-SVR primal problem is

$$\min_{w,b,\xi,\xi^*,\varepsilon} \frac{1}{2} w^T w + C(\nu\varepsilon + \frac{1}{l} \sum_{i=1}^{l} (\xi_i + \xi_i{}^*))$$

$$\text{subject to } (w^T \phi(x_i) + b) - z_i \leq \varepsilon + \xi_i,$$

$$z_i - (w^T \phi(x_i) + b) \leq \varepsilon + \xi_i{}^*,$$

$$\xi_i, \xi_i^* \geq 0, i = 1, \ldots, l, \varepsilon \geq 0$$

Parameter

`nuFormulation` – a `boolean`, giving the value for the flag

Default: `nuFormulation=false`

setNuParameter

```
public void setNuParameter(double nu)
```

Description

Sets the value of ν in the ν-formulation of the optimization problem.

Parameter

`nu` – a `double` in (0,1], the ν parameter

Default: `nu=0.5`

setProbability

```
public void setProbability(boolean probability)
```

Description

Sets the boolean to calculate probability estimates.

In classification problems, when the flag is `true`, class probabilities are estimated during the training procedure. For regression problems, when the flag is `true`, distributional parameters will be estimated that allow probability inferences for the response variable. When `false`, probability information is not calculated.

Parameter

`probability` – a `boolean`, indicating whether or not to calculate probability estimates

Default: `probability=false`

---

setRegularizationParameter

```
public void setRegularizationParameter(double C)
```

Description

Sets the regularization parameter, C.

Parameter

`C` – a `double`, greater than 0.0

Default: `C`=1.0

---

setShrinking

```
public void setShrinking(boolean shrinking)
```

Description

Sets the boolean to perform shrinking during optimization.

The shrinking technique tries to identify and remove some bounded elements during the application of the SMO (sequential minimal optimization) algorithm, so that a smaller optimization problem is solved.

Parameter

`shrinking` – a `boolean`, indicating whether or not shrinking should be performed during the optimization algorithm

Default: `shrinking=true`

---

setWorkArraySize

```
public void setWorkArraySize(double workSize)
```

Description

Sets the work array size.

A larger work array size can reduce the computational time of the decomposition method.

Parameter

`workSize` – a `double`, the number of megabytes allocated for the array used during the decomposition method. `workSize` must be greater than 0.

Default: `workSize` = 100.0

SupportVectorMachine.ReflectiveOperationException class

```
static public class
com.imsl.datamining.supportvectormachine.SupportVectorMachine.ReflectiveOperationException
extends com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Class that wraps exceptions thrown by reflective operations in core reflection.

Constructors

SupportVectorMachine.ReflectiveOperationException

```
public SupportVectorMachine.ReflectiveOperationException(String message)
```

Description

Constructs a `ReflectiveOperationException` and issues the specified message.

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

SupportVectorMachine.ReflectiveOperationException

```
public SupportVectorMachine.ReflectiveOperationException(String key, Object[]
arguments)
```

Description

Constructs a `ReflectiveOperationException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameters

`key` – a `String` that contains the key of an error message in the resource bundle

`arguments` – an `Object` array containing arguments used within the error message specified by the `key`

SupportVectorMachine.CloneNotSupportedException class

```
static public class
com.imsl.datamining.supportvectormachine.SupportVectorMachine.CloneNotSupportedException
```

```
extends com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Wraps the `java.lang.CloneNotSupportedException` to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface.

Constructors

SupportVectorMachine.CloneNotSupportedException

```
public SupportVectorMachine.CloneNotSupportedException(String message)
```

Description

Constructs a `CloneNotSupportedException` and issues the specified message.

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

SupportVectorMachine.CloneNotSupportedException

```
public SupportVectorMachine.CloneNotSupportedException(String key, Object[]
arguments)
```

Description

Constructs a `CloneNotSupportedException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameters

`key` – a `String` that contains the key of an error message in the resource bundle

`arguments` – an `Object` array containing arguments used within the error message specified by the `key`

SVClassification class

```
public class com.imsl.datamining.supportvectormachine.SVClassification extends
com.imsl.datamining.supportvectormachine.SupportVectorMachine
```

Specifies a support vector machine for classification (SVC).

The *C*-support vector classification (*C*-SVC) is the fundamental algorithm for the SVM optimization problem and its primal form is given as

$$\min_{w,b,\xi} \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi_i$$

$$\text{subject to } y_i(w^T \phi(x_i) + b) \geq 1 - \xi_i,$$

$$\xi_i \geq 0, i = 1, \ldots, l$$

The set $\{(x_i, y_i) : i = 1, \ldots, l\}$ is the set of instance-label pairs in the training data, with $x_i \in R^n$, $y_i \in \{1, -1\}$, and l equal to the number of training examples. The ξ_i are the slack variables in the optimization and represent an upper bound on the number of errors. The regularization parameter $C > 0$ acts as a tradeoff parameter between error and margin. This is the default algorithm for the classification problem.

Constructors

SVClassification

```
public SVClassification(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a support vector machine for classification (SVC).

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int`, the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

SVClassification

```
public SVClassification(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType, Kernel k) throws
CloneNotSupportedException
```

Description

Constructs a support vector machine for classification (SVC).

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int`, the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

`k` – a `Kernel`, the kernel function

Exception

`java.lang.CloneNotSupportedException` thrown to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface

Methods

getClassPenaltyWeights

```
public double[] getClassPenaltyWeights()
```

Description

Returns the class weights.

Returns

a `double` array containing the class weights

getClassWeightLabels

```
public int[] getClassWeightLabels()
```

Description

Returns the weight labels array.

Returns

an `int` array containing the class labels corresponding to the penalty weights

optimize

```
protected SVModel optimize(DataNode[][] x, double[] y, double[] w, int len,
Kernel kernel) throws NoSuchMethodException, InstantiationException,
IllegalAccessException, InvocationTargetException
```

Description

Performs the classification support vector machine optimization problem.

Parameters

`x` – a `DataNode` matrix containing the attribute data

`y` – a `double` array containing the response variable

`w` – a `double` array containing the observation weights

`len` – an `int`, the total possible number of support vectors

`kernel` – a `Kernel` object

Returns

an `SVModel` structure containing the fitted model

Exceptions

`java.lang.NoSuchMethodException` thrown when a particular method cannot be found

`java.lang.InstantiationException` thrown when an application tries to create an instance of a class using the `newInstance` method in class `Class`, but the specified class object cannot be instantiated

`java.lang.IllegalAccessException` thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor

`java.lang.reflect.InvocationTargetException` a checked exception that wraps an exception thrown by an invoked method or constructor

predictValues

```
protected double[] predictValues(SVModel model, double[][] attributeData)
throws PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Generates the predicted values on the attribute data using the given support vector machine model.

Parameters

`model` – a fitted `SVModel` object

`attributeData` – a `double` matrix containing the attribute (or predictor) data

Returns

a `double` array containing the predictions for each row in the input attribute data

Exception

`com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException` the sum of probabilities is not approximately one

setClassPenaltyWeights

```
public void setClassPenaltyWeights(int[] weightLabels, double[] weights)
```

Description

Sets the class penalty weights.

Parameters

`weightLabels` – an `int` array containing the class labels to which the weights should be applied

`weights` – a `double` array of length `weightLabels.length` containing the corresponding weights. The weight values must be non-negative.
Default: By default, the weight value is 1.0, leading to a penalty value of C for each class.

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
SupportVectorMachine.CloneNotSupportedException
```

Description

Sets the configuration to that of the input `PredictiveModel`.

Note that the input `PredictiveModel` object must be the same subclass of `PredictiveModel` as is this instance.

Parameter

`pm` – an `SVClassification` instance

Exception

`CloneNotSupportedException` a `java.lang.CloneNotSupportedException` has occurred. The original exception has been added to the `SupportVectorMachine.CloneNotSupportedException` as a suppressed exception.

Default: The class uses its default configuration as described in the different methods.

Example 1: SVClassification

This example selects training data from Fisher's Iris Data and fits a support vector machine using defaults. Then, predictions (fitted values) on the training data and predictions on the entire data set are produced and classification errors are shown.

```
import com.imsl.datamining.neural.*;
import com.imsl.datamining.supportvectormachine.*;

public class SupportVectorMachineEx1 {

    public static void main(String[] args) throws Exception {

        SVClassification.VariableType[] irisVarType = {
            SVClassification.VariableType.CATEGORICAL,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS
        };
        String dashes
                = "--------------------------------------------------------------";

        double[][] irisFisherData = {
            {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, .1}, {1.0, 5.0, 3.2, 1.2, .2},
{1.0, 5.5, 3.5, 1.3, .2}, {1.0, 4.9, 3.1, 1.5, .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}
};

// Create a scaled version of the Iris attribute data.
double[][] x = new double[150][4];
double[][] xx = new double[150][4];

// Get the data.
for (int i = 0; i < 150; i++) {
    for (int j = 0; j < 4; j++) {
        x[i][j] = irisFisherData[i][j + 1];
    }
}

// Scale the data.
double realMin = 0.0;
double realMax = 10.0;
double targetMin = 0.0;
double targetMax = 1.0;

ScaleFilter scaleFilter = new ScaleFilter(ScaleFilter.BOUNDED_SCALING);
scaleFilter.setBounds(realMin, realMax, targetMin, targetMax);

for (int i = 0; i < 150; i++) {
    xx[i] = scaleFilter.encode(x[i]);
}

// Build a training data set.
int nTrain = 30;
double[][] xy = new double[nTrain][5];
double[] knownClass = new double[nTrain];

int ii = 0;

// The response variable (Iris Species) is encoded starting in "1".
// Here, subtract 1 from the response because the class assumes
// 0 based categorical response variable.
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 10; j++) {
        xy[ii][0] = irisFisherData[(i * 50) + j][0] - 1;
        knownClass[ii] = xy[ii][0];
        System.arraycopy(xx[(i * 50 + j)], 0, xy[ii], 1, 4);
        ii++;
    }
}
// Construct a Support Vector Machine.
SVClassification svm = new SVClassification(xy, 0, irisVarType);

// Train the model on the training sample.
svm.fitModel();
```

```
        // Get the fitted values (classify the training data).
        double[] fittedClass = svm.predict();
        int[][] fittedClassErrors = svm.getClassErrors(knownClass, fittedClass);

        System.out.println("\n   Iris Classification Error Rates"
                + " (fitted values)");
        System.out.println("\n" + dashes);
        System.out.println(" Setosa Versicolour Virginica | TOTAL");
        System.out.println("  " + fittedClassErrors[0][0]
                + "/" + fittedClassErrors[0][1]
                + "      " + fittedClassErrors[1][0]
                + "/" + fittedClassErrors[1][1]
                + "       " + fittedClassErrors[2][0]
                + "/" + fittedClassErrors[2][1]
                + "     " + fittedClassErrors[3][0]
                + "/" + fittedClassErrors[3][1]);

        System.out.println(dashes);

        // Classify the entire data set.
        xy = new double[150][5];
        knownClass = new double[150];
        for (int i = 0; i < 150; i++) {
            xy[i][0] = irisFisherData[i][0] - 1;
            knownClass[i] = xy[i][0];
            System.arraycopy(xx[i], 0, xy[i], 1, 4);
        }

        double[] predictedClass = svm.predict(xy);
        int[][] classErrors = svm.getClassErrors(knownClass, predictedClass);

        System.out.println("\n   Iris Classification Error Rates "
                + "(predicted values)");
        System.out.println("\n" + dashes);
        System.out.println(" Setosa Versicolour Virginica | TOTAL");
        System.out.println("  " + classErrors[0][0] + "/" + classErrors[0][1]
                + "      " + classErrors[1][0] + "/" + classErrors[1][1]
                + "       " + classErrors[2][0] + "/" + classErrors[2][1]
                + "     " + classErrors[3][0] + "/" + classErrors[3][1]);

        System.out.println(dashes);
    }
}
```

Output

```
   Iris Classification Error Rates (fitted values)

--------------------------------------------------------------
 Setosa Versicolour Virginica | TOTAL
  0/10      1/10       1/10     2/30
--------------------------------------------------------------

   Iris Classification Error Rates (predicted values)
```

```
------------------------------------------------------------
 Setosa Versicolour Virginica | TOTAL
  0/50      4/50       5/50     9/150
------------------------------------------------------------
```

Example 2: SVClassification

This example uses stratified cross-validation to select parameter settings for C and γ using the minimum CV error criterion. Then, the fitted model using the "best" settings is used to classify the entire dataset. The classification errors are shown.

```
import com.imsl.datamining.*;
import com.imsl.datamining.supportvectormachine.*;
import com.imsl.datamining.neural.*;
import com.imsl.stat.*;

public class SupportVectorMachineEx2 {

    public static void main(String[] args) throws Exception {

        SVClassification.VariableType[] irisVarType = {
            SVClassification.VariableType.CATEGORICAL,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS,
            SVClassification.VariableType.QUANTITATIVE_CONTINUOUS
        };

        String dashes
                = "-------------------------------------------------------------";

        double[][] irisFisherData = {
            {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, .1}, {1.0, 5.0, 3.2, 1.2, .2},
            {1.0, 5.5, 3.5, 1.3, .2}, {1.0, 4.9, 3.1, 1.5, .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}
    };
```

```
// Create a scaled version of the Iris attribute data.
double[][] x = new double[150][4];
double[][] xx = new double[150][4];

// Get the data.
for (int i = 0; i < 150; i++) {
    for (int j = 0; j < 4; j++) {
        x[i][j] = irisFisherData[i][j + 1];
    }
}
// Scale the data.
double realMin = 0.0, realMax = 10.0, targetMin = 0.0, targetMax = 1.0;

ScaleFilter scaleFilter = new ScaleFilter(ScaleFilter.BOUNDED_SCALING);
scaleFilter.setBounds(realMin, realMax, targetMin, targetMax);

for (int i = 0; i < 150; i++) {
    xx[i] = scaleFilter.encode(x[i]);
}

// Build a training data set.
int nTrain = 30;
double[][] xy = new double[nTrain][5];
int ii = 0;
// The response variable (Iris Species) is encoded starting in "1".
// Here, subtract 1 from the response because the class assumes
// 0 based categorical response variable.
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 10; j++) {
        xy[ii][0] = irisFisherData[(i * 50) + j][0] - 1;
        System.arraycopy(xx[(i * 50 + j)], 0, xy[ii], 1, 4);
        ii++;
    }
}

// Construct a Support Vector Machine.
SVClassification svm = new SVClassification(xy, 0, irisVarType);

double[] gamma = {0.1};
double C = 2.0;
double result;
double minResult = 10000.0;
double bestGamma = 0.0;
double bestC = 0.0;

CrossValidation svmCV = new CrossValidation(svm);
svmCV.setNumberOfSampleFolds(5);
svmCV.setRandomObject(new Random(123457));
svmCV.setStratifiedCrossValidation(true);

for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 5; j++) {
        svm.setRegularizationParameter(C);
        svm.setKernelParameters(gamma);
        svmCV.crossValidate();
        result = svmCV.getCrossValidatedError();
```

```
                if (result < minResult) {
                    minResult = result;
                    bestGamma = gamma[0];
                    bestC = C;
                }
                gamma[0] = gamma[0] * 2.0;
            }
            gamma[0] = 0.1;
            C = C * 2.0;
        }
        System.out.printf("Best C: %5.0f \n", bestC);
        System.out.printf("Best gamma: %5.3f \n", bestGamma);
        System.out.printf("Minimum CV error: %5.3f \n", minResult);

        svm.setRegularizationParameter(bestC);
        gamma[0] = bestGamma;
        svm.setKernelParameters(gamma);
        // Train the model on the training sample (30 observations).
        svm.fitModel();

        // Classify the entire data set with the fitted model
        //  using the "best" C and gamma parameter values.
        xy = new double[150][5];
        double[] knownClass = new double[150];
        for (int i = 0; i < 150; i++) {
            xy[i][0] = irisFisherData[i][0] - 1;
            knownClass[i] = xy[i][0];
            System.arraycopy(xx[i], 0, xy[i], 1, 4);
        }

        double[] predictedClass = svm.predict(xy);
        int[][] classErrors = svm.getClassErrors(knownClass, predictedClass);

        System.out.println("\n   Iris Classification Error Rates");
        System.out.println("\n" + dashes);
        System.out.println(" Setosa Versicolour Virginica | TOTAL");
        System.out.println("  " + classErrors[0][0] + "/" + classErrors[0][1]
                + "      " + classErrors[1][0] + "/" + classErrors[1][1]
                + "       " + classErrors[2][0] + "/" + classErrors[2][1]
                + "     " + classErrors[3][0] + "/" + classErrors[3][1]);

        System.out.println(dashes);
    }
}
```

Output

```
Best C:    64
Best gamma: 1.600
Minimum CV error: 0.000

   Iris Classification Error Rates

-------------------------------------------------------------
 Setosa Versicolour Virginica | TOTAL
```

```
  0/50      1/50       3/50     4/150
--------------------------------------------------------------
```

SVOneClass class

`public class com.imsl.datamining.supportvectormachine.SVOneClass extends com.imsl.datamining.supportvectormachine.SupportVectorMachine`

Specifies a support vector machine for the one class problem. The one class SVM estimates the support (the range of values with positive density) of a high-dimensional distribution.

The one-class SVM algorithm estimates the support of a high-dimensional distribution without any class information. The primal problem of one-class SVM is

$$\min_{w,\xi,\rho} \frac{1}{2}w^T w - \rho + \frac{1}{\nu l}\sum_{i=1}^{l}\xi_i$$

$$\text{subject to } (w^T\phi(x_i)) \geq \rho - \xi_i,$$

$$\xi_i \geq 0, i = 1, \ldots, l$$

Constructors

SVOneClass

`public SVOneClass(double[][] xy, int responseColumnIndex, PredictiveModel.VariableType[] varType)`

Description

Constructs a one class support vector machine.

Parameters

`xy` – a `double` matrix containing the training data and associated response values by the number of variables

`responseColumnIndex` – an `int`, the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

SVOneClass

```
public SVOneClass(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType, Kernel k) throws
CloneNotSupportedException
```

Description

Constructs a one class support vector machine.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int`, the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

`k` – a `Kernel`, the kernel function

Exception

`java.lang.CloneNotSupportedException` thrown to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface

Methods

optimize

```
protected SVModel optimize(DataNode[][] x, double[] y, double[] w, int len,
Kernel kernel) throws NoSuchMethodException, InstantiationException,
IllegalAccessException, InvocationTargetException
```

Description

Performs the one class support vector machine optimization problem.

Parameters

`x` – a `DataNode` matrix containing the attribute data

`y` – a `double` array containing the response variable

`len` – an `int`, the total possible number of support vectors

`w` – a `double` array containing the observation weights

`kernel` – a `Kernel` object

Returns

an `SVModel` structure containing the fitted model

Exceptions

`java.lang.NoSuchMethodException` thrown when a particular method cannot be found

`java.lang.InstantiationException` thrown when an application tries to create an instance of a class using the `newInstance` method in class `Class`, but the specified class object cannot be instantiated.

`java.lang.IllegalAccessException` thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor

`java.lang.reflect.InvocationTargetException` a checked exception that wraps an exception thrown by an invoked method or constructor

predictValues

`protected double[] predictValues(SVModel model, double[][] attributeData) throws PredictiveModel.SumOfProbabilitiesNotOneException`

Description

Generates the predicted values on the attribute data using the given support vector machine model.

Parameters

`model` – a fitted `SVModel` object

`attributeData` – a `double` matrix containing the attribute (or predictor) data

Returns

a `double` array containing the predictions for each row in the input attribute data

Exception

`com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException` the sum of probabilities is not approximately one

Example: SVOneClass

In this example, a one class support vector machine is used to detect examples in the test data that do not belong to the target population. The target population is the uniform distribution.

```
import com.imsl.datamining.supportvectormachine.*;
import com.imsl.stat.*;

public class SupportVectorMachineEx3 {

    public static void main(String[] args) throws Exception {

        SVOneClass.VariableType[] ex3VarType = {
            SVOneClass.VariableType.ONE_CLASS,
            SVOneClass.VariableType.QUANTITATIVE_CONTINUOUS
        };

        // 1000 training patterns
        int nTrain = 1000;
```

```
        // 100 test patterns
        int nTest = 100;
        int nContaminated = 10;
        int nAttributes = 1;

        double[][] xyTrain = new double[nTrain][nAttributes + 1];
        double[][] xyTest = new double[nTest][2];

        // Create the training set from a uniform distribution.
        Random r = new Random(123457);
        r.setMultiplier(16807);
        for (int i = 0; i < nTrain; i++) {
            xyTrain[i][1] = r.nextFloat();
            xyTrain[i][0] = 1;
        }
        //Construct an SVOneClass object.
        SVOneClass svm1 = new SVOneClass(xyTrain, 0, ex3VarType);
        svm1.setNuParameter(0.001);
        // Train with the training data.
        svm1.fitModel();

        // Create a testing set from a uniform distribution.
        for (int i = 0; i < nTest; i++) {
            xyTest[i][1] = r.nextFloat();
            xyTest[i][0] = 1;
        }

        //Contaminate the testing set with deviates from a normal distribution.
        for (int i = 0; i < nContaminated; i++) {
            xyTest[i * 10][1] = r.nextNormal() * Math.sqrt(.2) + .1;
        }
        double[] predictedClass = svm1.predict(xyTest);
        System.out.println("\n\n\n                   Classification Results\n");
        for (int i = 0; i < nTest; i++) {
            if (predictedClass[i] != 1.0) {
                System.out.println("\n The " + i + "-th observation may not "
                        + "belong to the target distribution.");
            }
        }
    }
}
```

Output

```



                   Classification Results


 The 0-th observation may not belong to the target distribution.

 The 20-th observation may not belong to the target distribution.

 The 30-th observation may not belong to the target distribution.
```

```
The 40-th observation may not belong to the target distribution.

The 60-th observation may not belong to the target distribution.

The 70-th observation may not belong to the target distribution.
```

SVRegression class

`public class com.imsl.datamining.supportvectormachine.SVRegression extends com.imsl.datamining.supportvectormachine.SupportVectorMachine`

Specifies a support vector machine for regression (SVR).

The standard form SVR is the so-called *epsilon*-support vector regression, or ε-SVR. If z_i is the target output, then given the parameters $C > 0, \varepsilon > 0$, the standard form SVR is

$$\min_{w,b,\xi,\xi^*} \frac{1}{2} w^T w + C \sum_{i=1}^{l} \xi_i + C \sum_{i=1}^{l} \xi_i{}^*$$

$$\text{subject to } w^T \phi(x_i) + b - z_i \leq \varepsilon + \xi_i,$$

$$z_i - w^T \phi(x_i) - b \leq \varepsilon + \xi_i{}^*,$$

$$\xi_i, \xi_i{}^* \geq 0, i = 1, \ldots, l$$

The variables ξ_i and $\xi_i{}^*$ are two slack variables, one for exceeding the target value by more than ε and the other for being more than ε below the target. ε is controlled through the use of the set method `setInsensitivityBand`.

Constructors

SVRegression

`public SVRegression(double[][] xy, int responseColumnIndex, PredictiveModel.VariableType[] varType)`

Description

Constructs a support vector machine for regression (SVR).

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

SVRegression

```
public SVRegression(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType, Kernel k) throws
CloneNotSupportedException
```

Description

Constructs a support vector machine for regression (SVR).

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `PredictiveModel.VariableType` array of length equal to `xy[0].length` containing the type of each variable

`k` – a `Kernel`, the kernel function

Exception

`java.lang.CloneNotSupportedException` thrown to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface

Methods

getInsensitivityBand

```
public double getInsensitivityBand()
```

Description

Returns the insensitivity band parameter, ε, in the standard formulation of the SVM regression problem.

Returns

a `double`, the value of the insensitivity band parameter

optimize

```
protected SVModel optimize(DataNode[][] x, double[] y, double[] w, int len,
Kernel kernel) throws NoSuchMethodException, InstantiationException,
IllegalAccessException, InvocationTargetException
```

Description

Performs the regression support vector machine optimization problem.

Parameters

`x` – a `DataNode` matrix containing the attribute data

`y` – a `double` array containing the response variable

`len` – an `int`, the total possible number of support vectors

`w` – a `double` array containing the observation weights

`kernel` – a `Kernel` object

Returns

an `SVModel` structure containing the fitted model

Exceptions

`java.lang.NoSuchMethodException` thrown when a particular method cannot be found

`java.lang.InstantiationException` thrown when an application tries to create an instance of a class using the `newInstance` method in class `Class`, but the specified class object cannot be instantiated

`java.lang.IllegalAccessException` thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method or constructor

`java.lang.reflect.InvocationTargetException` a checked exception that wraps an exception thrown by an invoked method or constructor

predictValues

```
protected double[] predictValues(SVModel model, double[][] attributeData)
throws PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Generates the predicted values on the attribute data using the given support vector machine model.

Parameters

`model` – a fitted `SVModel` object

`attributeData` – a `double` matrix containing the attribute (or predictor) data

Returns

a `double` array containing the predictions for each row in the input attribute data

Exception

`com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException` the sum of probabilities is not approximately one

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
SupportVectorMachine.CloneNotSupportedException
```

Description

Sets the configuration to that of the input `PredictiveModel`.

Note that the input `PredictiveModel` object must be the same subclass of `PredictiveModel` as is this instance.

Parameter

`pm` – a `SVRegression` instance

Exception

`CloneNotSupportedException` a `java.lang.CloneNotSupportedException` has occurred. The original exception has been added to the `SupportVectorMachine.CloneNotSupportedException` as a suppressed exception.

Default: The class uses its default configuration as described in the different methods.

setInsensitivityBand

```
public void setInsensitivityBand(double epsilon)
```

Description

Sets the insensitivity band parameter, ε, in the standard formulation of the SVM regression problem.

Parameter

`epsilon` – a `double` > 0.0, the value of the infeasibility band parameter

Default: `epsilon`=0.1

Example 1: SVRegression

In this example, support vector regression (ν-SVR) is applied to a categorical response variable. The fitted values and prediction values are in the right tendency. Compare these to the results from support vector classification (ν-SVC), which is more appropriate for a categorical response variable.

```
import com.imsl.datamining.supportvectormachine.*;

public class SupportVectorMachineEx4 {

    public static void main(String[] args) throws Exception {

        SVRegression.VariableType[] ex4DataType = {
            SVRegression.VariableType.CATEGORICAL,
            SVRegression.VariableType.QUANTITATIVE_CONTINUOUS,
            SVRegression.VariableType.QUANTITATIVE_CONTINUOUS};

        String dashes
                = "-------------------------------------------------------------";

        double C = 50., nu = .01;
        double[][] xyTrain = {
            {1, 0.19, 0.61}, {1, 0.156, 0.564}, {1, 0.224, 0.528},
            {1, 0.178, 0.51}, {1, 0.234, 0.578}, {2, 0.394, 0.296},
            {2, 0.478, 0.254}, {2, 0.454, 0.294}, {2, 0.48, 0.358},
```

```
    {2, 0.398, 0.336}
};
double[][] xyTest = {
    {1, 0.316, 0.556}, {1, 0.278, 0.622},
    {2, 0.562, 0.336}, {2, 0.522, 0.412}
};

/* Construct an SVMRegression. */
SVRegression svm1 = new SVRegression(xyTrain, 0, ex4DataType);
svm1.setNuFormulation(true);
svm1.setNuParameter(nu);
svm1.setRegularizationParameter(C);
svm1.fitModel();
double[] fittedValues = svm1.predict();

System.out.println("\n" + dashes);
System.out.println(" NU SVR: Training data predicted (fitted) values");
System.out.println(" Actual  Fitted value | Difference");
for (int i = 0; i < fittedValues.length; i++) {
    System.out.printf("  %2.1f      %5.4f       %5.4f\n", xyTrain[i][0],
            fittedValues[i], (fittedValues[i] - xyTrain[i][0]));
}
System.out.println("\n" + dashes);
double[] testPredictedValues = svm1.predict(xyTest);
System.out.println("\n NU SVR:   Test data predictions");
System.out.println("  Actual  Prediction  | Difference");
for (int i = 0; i < testPredictedValues.length; i++) {
    System.out.printf("  %2.1f      %5.4f      %5.4f\n", xyTest[i][0],
            testPredictedValues[i],
            (testPredictedValues[i] - xyTest[i][0]));
}

/* Now use the categorical version and compare results. */
SVClassification svm2 = new SVClassification(xyTrain, 0, ex4DataType);
svm2.setNuFormulation(true);
svm2.setNuParameter(nu);
svm2.setRegularizationParameter(C);
svm2.fitModel();
fittedValues = svm2.predict();

System.out.println("\n" + dashes);
System.out.println(" NU SVC:  Training data predicted (fitted) values");
System.out.println(" Actual  Fitted value | Difference");
for (int i = 0; i < fittedValues.length; i++) {
    System.out.printf("  %2.1f      %5.4f       %5.4f\n", xyTrain[i][0],
            fittedValues[i], (fittedValues[i] - xyTrain[i][0]));
}
System.out.println("\n" + dashes);
testPredictedValues = svm2.predict(xyTest);
System.out.println("\n NU SVC:  Test data predictions");
System.out.println(" Actual  Prediction  | Difference");
for (int i = 0; i < testPredictedValues.length; i++) {
    System.out.printf("  %2.1f      %5.4f      %5.4f\n", xyTest[i][0],
            testPredictedValues[i],
            (testPredictedValues[i] - xyTest[i][0]));
}
```

```
    }
}
```

Output

```
--------------------------------------------------------------
 NU SVR: Training data predicted (fitted) values
 Actual  Fitted value | Difference
  1.0      1.3662       0.3662
  1.0      1.3730       0.3730
  1.0      1.4175       0.4175
  1.0      1.4066       0.4066
  1.0      1.3987       0.3987
  2.0      1.5993       -0.4007
  2.0      1.6522       -0.3478
  2.0      1.6249       -0.3751
  2.0      1.6063       -0.3937
  2.0      1.5825       -0.4175

--------------------------------------------------------------

 NU SVR:   Test data predictions
  Actual  Prediction  | Difference
  1.0      1.4436      0.4436
  1.0      1.3972      0.3972
  2.0      1.6485      -0.3515
  2.0      1.5983      -0.4017

--------------------------------------------------------------
 NU SVC:  Training data predicted (fitted) values
 Actual  Fitted value | Difference
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000

--------------------------------------------------------------

 NU SVC:  Test data predictions
 Actual  Prediction  | Difference
  1.0      1.0000      0.0000
  1.0      1.0000      0.0000
  2.0      2.0000      0.0000
  2.0      2.0000      0.0000
```

Example 2: SVRegression

This example is identical to Example 4 except that it also shows how to apply case weights to the training data. Compare the fitted values between the two examples (the predicted values do not change).

```
import com.imsl.datamining.supportvectormachine.*;

public class SupportVectorMachineEx5 {

    public static void main(String[] args) throws Exception {

        SVRegression.VariableType[] ex4DataType = {
            SVRegression.VariableType.CATEGORICAL,
            SVRegression.VariableType.QUANTITATIVE_CONTINUOUS,
            SVRegression.VariableType.QUANTITATIVE_CONTINUOUS};

        String dashes
                = "--------------------------------------------------------------";

        double C = 50., nu = .01;
        double[][] xyTrain = {
            {1, 0.19, 0.61}, {1, 0.156, 0.564}, {1, 0.224, 0.528},
            {1, 0.178, 0.51}, {1, 0.234, 0.578}, {2, 0.394, 0.296},
            {2, 0.478, 0.254}, {2, 0.454, 0.294}, {2, 0.48, 0.358},
            {2, 0.398, 0.336}
        };
        double[][] xyTest = {
            {1, 0.316, 0.556}, {1, 0.278, 0.622}, {2, 0.562, 0.336},
            {2, 0.522, 0.412}
        };

        double[] trainingWts = {10, 1, 1, 1, 1, 1, 1, 1, 1, 1};

        /* Construct a Support Vector Machine. */
        SVRegression svm1 = new SVRegression(xyTrain, 0, ex4DataType);
        svm1.setNuFormulation(true);
        svm1.setNuParameter(nu);
        svm1.setRegularizationParameter(C);
        svm1.setWeights(trainingWts);
        svm1.fitModel();

        double[] fittedValues = svm1.predict();

        System.out.println("\n" + dashes);
        System.out.println(" NU SVR: Training data predicted (fitted) values");
        System.out.println(" Actual  Fitted value | Difference");
        for (int i = 0; i < fittedValues.length; i++) {
            System.out.printf("  %2.1f      %5.4f       %5.4f\n", xyTrain[i][0],
                    fittedValues[i], (fittedValues[i] - xyTrain[i][0]));
        }
        System.out.println("\n" + dashes);

        double[] testPredictedValues = svm1.predict(xyTest);

        System.out.println("\n NU SVR:   Test data predictions");
```

```
        System.out.println("  Actual  Prediction  | Difference");
        for (int i = 0; i < testPredictedValues.length; i++) {
            System.out.printf("  %2.1f      %5.4f      %5.4f\n", xyTest[i][0],
                    testPredictedValues[i],
                    (testPredictedValues[i] - xyTest[i][0]));
        }
        /* Now use the categorical version and compare results. */
        SVClassification svm2 = new SVClassification(xyTrain, 0, ex4DataType);
        svm2.setNuFormulation(true);
        svm2.setNuParameter(nu);
        svm2.setRegularizationParameter(C);
        svm2.setWeights(trainingWts);
        svm2.fitModel();

        fittedValues = svm2.predict();

        System.out.println("\n" + dashes);
        System.out.println(" NU SVC:  Training data predicted (fitted) values");
        System.out.println(" Actual  Fitted value | Difference");
        for (int i = 0; i < fittedValues.length; i++) {
            System.out.printf("  %2.1f      %5.4f       %5.4f\n", xyTrain[i][0],
                    fittedValues[i], (fittedValues[i] - xyTrain[i][0]));
        }
        System.out.println("\n" + dashes);

        testPredictedValues = svm2.predict(xyTest);

        System.out.println("\n NU SVC:  Test data predictions");
        System.out.println(" Actual  Prediction  | Difference");
        for (int i = 0; i < testPredictedValues.length; i++) {
            System.out.printf("  %2.1f      %5.4f      %5.4f\n", xyTest[i][0],
                    testPredictedValues[i],
                    (testPredictedValues[i] - xyTest[i][0]));
        }
    }
}
```

Output

```
--------------------------------------------------------------
 NU SVR: Training data predicted (fitted) values
 Actual  Fitted value | Difference
  1.0      1.2458      0.2458
  1.0      1.2586      0.2586
  1.0      1.3432      0.3432
  1.0      1.3226      0.3226
  1.0      1.3075      0.3075
  2.0      1.6886      -0.3114
  2.0      1.7892      -0.2108
  2.0      1.7373      -0.2627
  2.0      1.7020      -0.2980
  2.0      1.6568      -0.3432

--------------------------------------------------------------
```

```
 NU SVR:   Test data predictions
  Actual  Prediction  | Difference
  1.0      1.3928      0.3928
  1.0      1.3048      0.3048
  2.0      1.7822      -0.2178
  2.0      1.6868      -0.3132

--------------------------------------------------------------
 NU SVC:  Training data predicted (fitted) values
 Actual  Fitted value | Difference
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  1.0      1.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000
  2.0      2.0000       0.0000

--------------------------------------------------------------

 NU SVC:  Test data predictions
 Actual  Prediction  | Difference
  1.0      1.0000      0.0000
  1.0      1.0000      0.0000
  2.0      2.0000      0.0000
  2.0      2.0000      0.0000
```

Kernel class

`abstract public class com.imsl.datamining.supportvectormachine.Kernel implements Cloneable`

Abstract class to specify a kernel function for support vector machines.

Constructor

Kernel

`protected Kernel(int numberOfParameters)`

Description

Creates a `Kernel` and specifies the number of kernel parameters for a specific Kernel.

Parameter

`numberOfParameters` – an `int` indicating the number of parameters this `Kernel` must have

Methods

clone

`public Kernel clone() throws CloneNotSupportedException`

Description

Returns a clone of this object.

Returns

a `Kernel` which is a clone of this object

Exception

`CloneNotSupportedException` Thrown to indicate that the `clone` method in class `Object` has been called to clone an object, but that the object's class does not implement the `Cloneable` interface.

dot

`protected double dot(DataNode[] x, DataNode[] y)`

Description

Calculates the dot product between two `DataNode` arrays. The array lengths may be different.

Parameters

`x` – a `DataNode` array

`y` – a `DataNode` array

Returns

a `double`, the dot product between `x` and `y`

getParameters

`public double[] getParameters()`

Description

Returns the kernel parameters.

Returns

a `double` array containing the kernel parameters

kernelFunction

`abstract public double kernelFunction(DataNode[] x, DataNode[] y)`

Description

Abstract method to calculate the kernel function between two `DataNode` arrays.

Parameters

x – a DataNode array

y – a DataNode array

Returns

a double, the kernel function evaluated at x and y

kernelFunction

```
abstract public double kernelFunction(DataNode[][] x, int i, int j)
```

Description

Abstract method to calculate the kernel function between two DataNode arrays.

Parameters

x – a DataNode matrix

i – an int, the index of the first DataNode, x[i]

j – an int, the index of the second DataNode, x[j]

Returns

a double, the kernel function evaluated at x[i] and x[j]

setParameters

```
public void setParameters(double[] kParams)
```

Description

Sets the kernel parameters.

Specific instances can override this method to test for proper values of the parameters.

Parameter

kParams – a double array containing the parameters

LinearKernel class

```
public class com.imsl.datamining.supportvectormachine.LinearKernel extends
com.imsl.datamining.supportvectormachine.Kernel
```

Specifies the linear kernel for support vector machines.

The kernel function at two data nodes, x_i and x_j, is given by $K(x_i,x_j) = x_i^T x_j$.

Constructor

LinearKernel

`public LinearKernel()`

Description

Constructs a `LinearKernel` with default parameters.

Methods

kernelFunction

`public double kernelFunction(DataNode[] x, DataNode[] y)`

Description

Calculates the kernel function between two `DataNode`s.

Parameters

- `x` – a `DataNode` array
- `y` – a `DataNode` array

Returns

a `double`, the kernel function evaluated at `x` and `y`

kernelFunction

`public double kernelFunction(DataNode[][] x, int i, int j)`

Description

Calculates the kernel function between two `DataNode`s.

Parameters

- `x` – a `DataNode` matrix
- `i` – an `int`, index to the first `DataNode`, `x[i]`
- `j` – an `int`, index to the second `DataNode`, `x[j]`

Returns

a `double`, the kernel function evaluated at `x[i]` and `x[j]`

SigmoidKernel class

```
public class com.imsl.datamining.supportvectormachine.SigmoidKernel extends
com.imsl.datamining.supportvectormachine.Kernel
```

Specifies the sigmoid kernel for support vector machines.

The kernel function at two data nodes, x_i and x_j, is given by $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$ where γ and r are configurable parameters.

Constructors

SigmoidKernel

```
public SigmoidKernel()
```

Description

Constructor for the sigmoid kernel.

The default values for the sigmoid kernel are γ=1.0 and `r`=0.0.

SigmoidKernel

```
public SigmoidKernel(double gamma, double r)
```

Description

Constructs a sigmoid kernel.

Parameters

`gamma` – a `double`, the sigmoid kernel slope parameter

`r` – a `double`, the function intercept parameter

Methods

kernelFunction

```
public double kernelFunction(DataNode[] x, DataNode[] y)
```

Description

Calculates the kernel function between two `DataNode`s.

Parameters

`x` – a `DataNode` array

`y` – a `DataNode` array

Returns

a `double`, the kernel function evaluated at `x` and `y`

kernelFunction

```
public double kernelFunction(DataNode[][] x, int i, int j)
```

Description

Calculates the kernel function between two `DataNodes`.

Parameters

`x` – a `DataNode` matrix

`i` – an `int`, index to the first `DataNode`, `x[i]`

`j` – an `int`, index to the second `DataNode`, `x[j]`

Returns

a `double`, the kernel function evaluated at `x[i]` and `x[j]`

setParameters

```
public void setParameters(double[] kParams)
```

Description

Sets the parameters for the sigmoid kernel.

Parameter

`kParams` – a `double` array of length 2 containing parameter values: `kParams[0]`=γ and `kParams[1]`=r

Default: `kParams[0]`=γ=1.0 and `kParams[1]`=r=0

RadialBasisKernel class

```
public class com.imsl.datamining.supportvectormachine.RadialBasisKernel extends
com.imsl.datamining.supportvectormachine.Kernel
```

Specifies the radial basis kernel for support vector machines.

The kernel function at two data nodes, x_i and x_j, is given by $K(x_i, x_j) = \exp(-\gamma\| x_i - x_j \|^2)$, where $\gamma > 0$ is a configurable parameter.

Constructors

RadialBasisKernel

```
public RadialBasisKernel()
```

Description

Constructs a radial basis kernel with a γ value of 1.0.

RadialBasisKernel

```
public RadialBasisKernel(double gamma)
```

Description

Constructs a radial basis kernel.

Parameter

gamma – a double, the radial basis kernel free parameter

Methods

kernelFunction

```
public double kernelFunction(DataNode[] x, DataNode[] y)
```

Description

Calculates the kernel function between two DataNodes.

Parameters

x – a DataNode array

y – a DataNode array

Returns

a double, the kernel function evaluated at x and y

kernelFunction

```
public double kernelFunction(DataNode[][] x, int i, int j)
```

Description

Calculates the kernel function between two DataNodes.

Parameters

x – a DataNode matrix

i – an int, index to the first DataNode, x[i]

j – an int, index to the second DataNode, x[j]

Returns

a double, the kernel function evaluated at x[i] and x[j]

setParameters

```
public void setParameters(double[] kParams)
```

Description

Sets the parameters for the radial basis kernel.

Parameter

kParams – a double array of length 1 containing the parameter value: kParams[0]=γ
Default: kParams[0]=γ=1.0

PolynomialKernel class

`public class com.imsl.datamining.supportvectormachine.PolynomialKernel extends com.imsl.datamining.supportvectormachine.Kernel`

Specifies the polynomial kernel for support vector machines.

The kernel function at two data nodes, x_i and x_j, is given by $K(x_i,x_j) = (\gamma x_i^T x_j + r)^d$, where $\gamma > 0$, r, and $d > 0$ are configurable parameters.

Constructors

PolynomialKernel

`public PolynomialKernel()`

Description

Constructor for the polynomial kernel.

The default values for the polynomial kernel are γ=1.0, `r`=0.0, and `degree`=1.

PolynomialKernel

`public PolynomialKernel(double gamma, double r, int degree)`

Description

Constructs a polynomial kernel.

Parameters

`gamma` – a `double`, the polynomial kernel free parameter

`r` – a `double`, the soft margin cost function parameter

`degree` – a `double`, the degree of the polynomial kernel

Methods

kernelFunction

`public double kernelFunction(DataNode[] x, DataNode[] y)`

Description

Calculates the kernel function between two `DataNode`s.

Parameters

`x` – a `DataNode` array

`y` – a `DataNode` array

Returns

a `double`, the kernel function evaluated at `x` and `y`

kernelFunction

```
public double kernelFunction(DataNode[][] x, int i, int j)
```

Description

Calculates the kernel function between two `DataNode`s.

Parameters

`x` – a `DataNode` matrix

`i` – an `int`, the index of the first `DataNode`, `x[i]`

`j` – an `int`, the index of the second `DataNode`, `x[j]`

Returns

a `double`, the kernel function evaluated at `x[i]` and `x[j]`

setParameters

```
public void setParameters(double[] kParams)
```

Description

Sets the parameters for the polynomial kernel.

Parameter

`kParams` – a `double` array of length 3 containing parameter values: `kParams[0]`=γ, `kParams[1]`=r, and `kParams[2]`=d.

Default: `kParams[0]`=γ=1.0, `kParams[1]`=r=0, and `kParams[2]`=d=1.

DataNode class

```
public class com.imsl.datamining.supportvectormachine.DataNode
```

Specifies a data node for a support vector machine.

A node has two elements, an `index` and a `value`.

Constructor

DataNode

```
public DataNode()
```

Methods

getIndex

`public int getIndex()`

Description

Returns the index of the node.

Returns

an `int`, the index of the node

getValue

`public double getValue()`

Description

Returns the value of the node.

Returns

a `double`, the value of the node

setIndex

`public void setIndex(int idx)`

Description

Sets the index of the node.

Parameter

`idx` – an `int`, the index of the node

setValue

`public void setValue(double val)`

Description

Sets the value of the node.

Parameter

`val` – a `double`, the value of the node

Chapter 31: Neural Nets

Types

class Network 2090
class FeedForwardNetwork 2099
class Layer 2113
class InputLayer 2114
class HiddenLayer 2115
class OutputLayer 2116
class Node 2118
class InputNode 2118
class Perceptron 2119
class OutputPerceptron 2120
interface Activation 2121
class Link 2123
interface Trainer 2124
class QuasiNewtonTrainer 2125
class LeastSquaresTrainer 2134
class EpochTrainer 2138
class BinaryClassification 2144
class MultiClassification 2187
class ScaleFilter 2202
class UnsupervisedNominalFilter 2211
class UnsupervisedOrdinalFilter 2215
class TimeSeriesFilter 2220
class TimeSeriesClassFilter 2222

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.

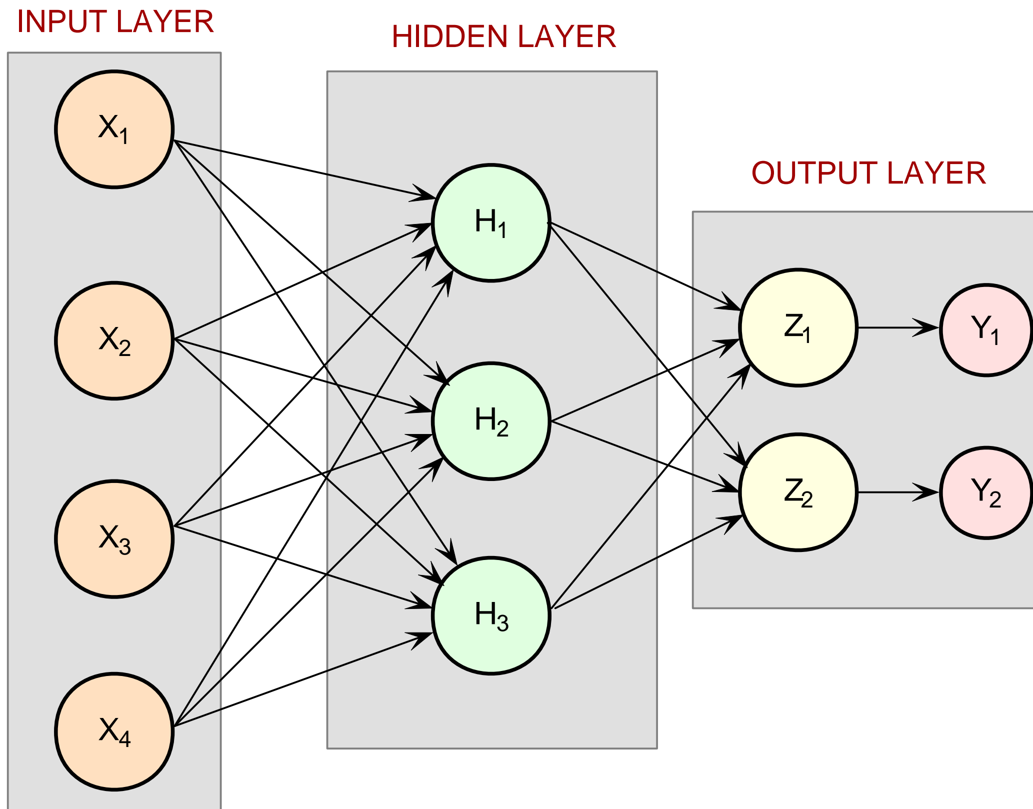


Figure 1. A 2-layer, Feed-Forward Network with 4 Inputs and 2 Outputs

Other networks, such as recurrent neural networks, allow data and information to flow in both directions, see Mandic and Chambers (2001).

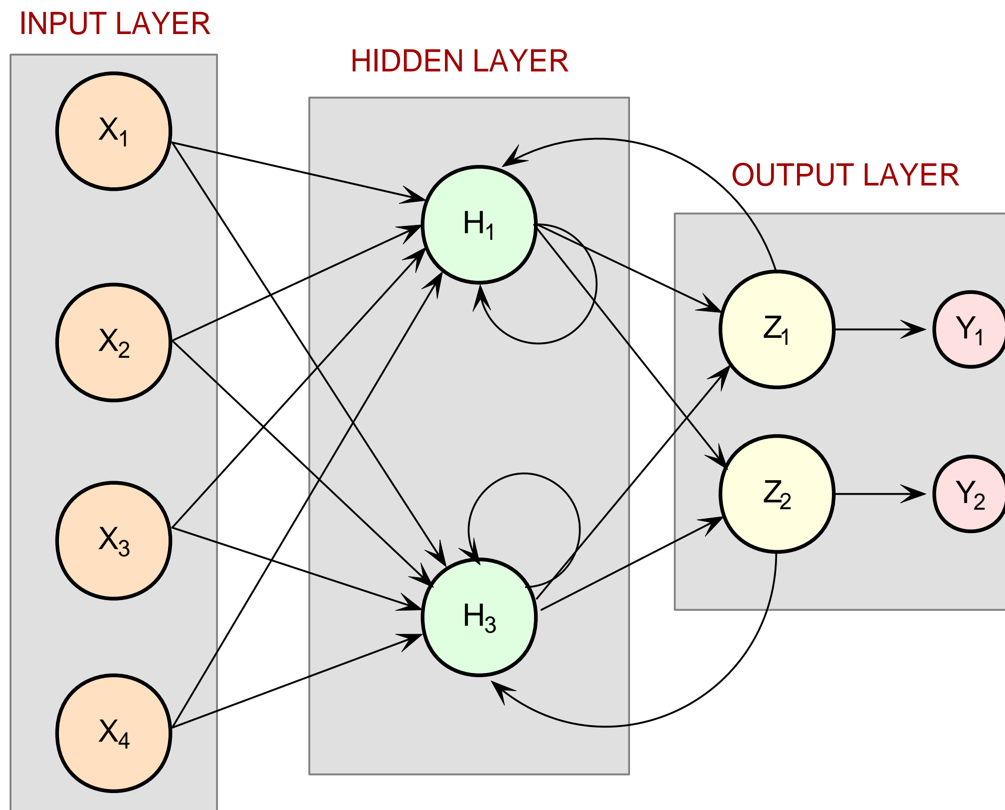


Figure 2. A Recurrent Neural Network with 4 Inputs and 2 Outputs

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.

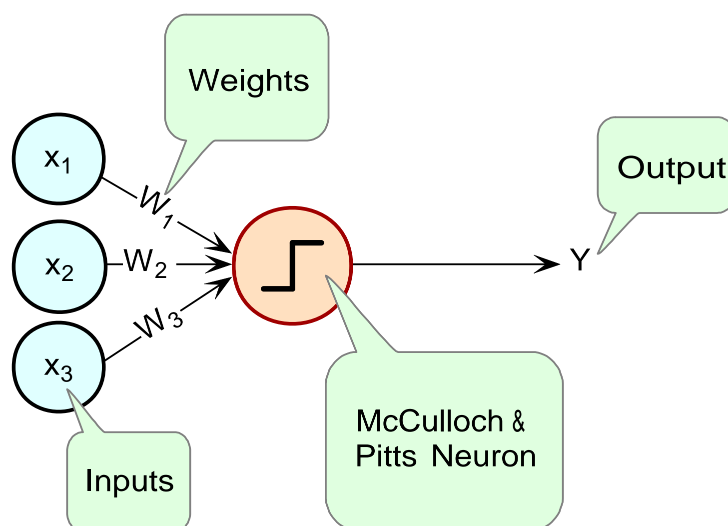


Figure 3. The McCulloch and Pitts Threshold Neuron

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 μ 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.

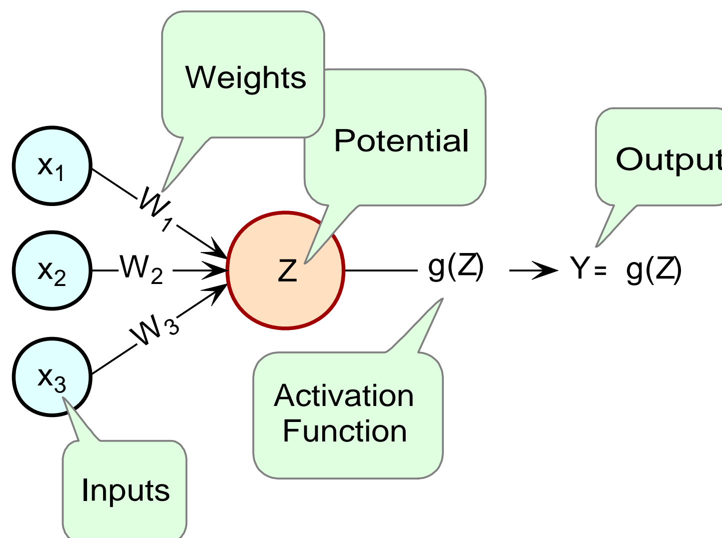


Figure 4. The Perceptron

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 μ is a bias correction.
Activation Function	$g()$
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.

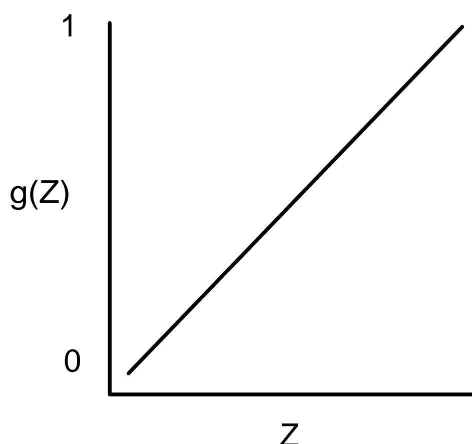


Figure 5. An Identity (Linear) Activation Function

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.

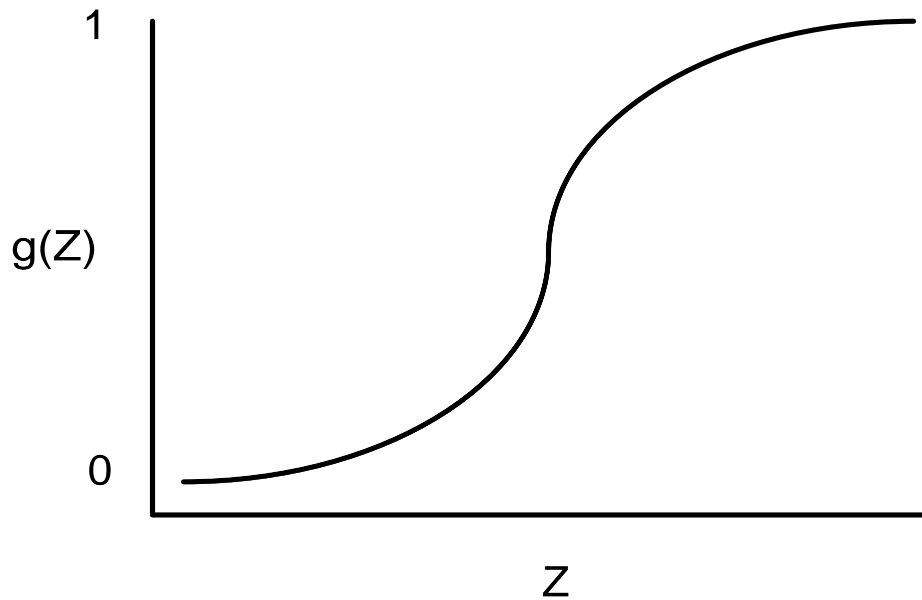


Figure 6. A Sigmoid Activation Function

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 Schütze (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 a 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 one output perceptron to each class in the classification problem. Inputs to the network are associated with the class */ with the highest probability for that input pattern. However, this approach requires the output probabilities sum to one, which is a requirement for any valid multivariate probability distribution.

To ensure these probabilities sum to one, the softmax activation function, see Bridle (1990), is applied to the network outputs ensuring 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 Z_i is the weighted sum of products between its weights and

inputs for the i-th output, i.e., $Z_i = \sum_j w_{ji} y_{ji}$, then

$$\text{softmax}_i = \frac{e^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}}$$

The softmax activation function ensures that the outputs all conform to the requirements for multivariate probabilities. That is,

$$0 < \text{softmax}_i < 1, \quad \text{for all } i = 1, 2, \ldots, C$$

and

$$\sum_{i=1}^{C} \text{softmax}_i = 1$$

A pattern is assigned to the i-th classification when 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.

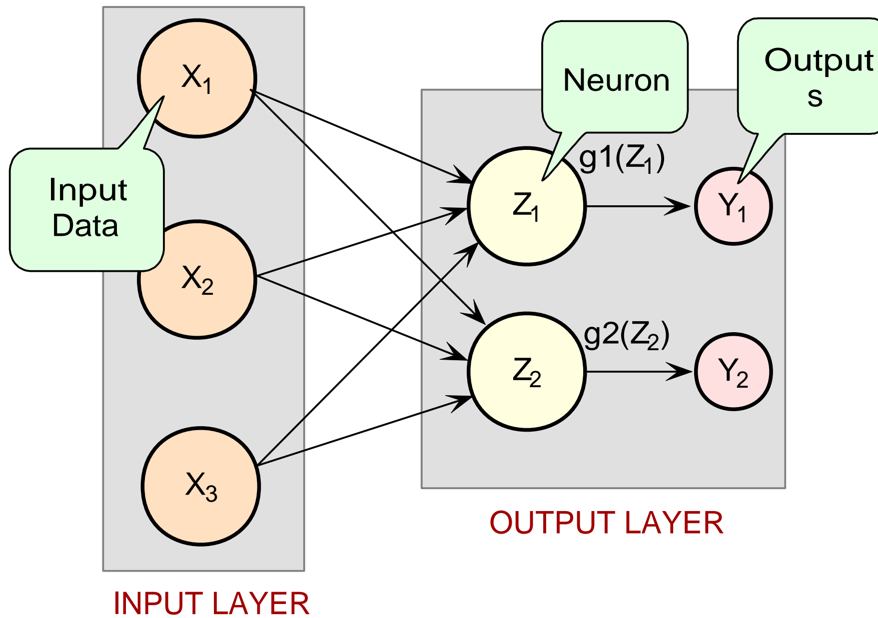


Figure 7. A Single-Layer Feed-Forward Neural Net

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:

$$E = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{C} (t_{ij} - \hat{t}_{ij})^2$$

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:

$$E^{RMS} = \frac{\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij}-\hat{t}_{ij}\right)^2}{\sum_{i=1}^{N}\sum_{j=1}^{C}\left(t_{ij}-\bar{t}\right)^2}$$

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:

$$E^{RMS} = \frac{2}{\bar{t}} \cdot E$$

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 network 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):

$$E^C = -\sum_{i=1}^{N} \{t_i \ln(\hat{t}_i) + (1 - t_i)\ln(1 - \hat{t}_i)\}.$$

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:

$$E^C_{\min} = -\sum_{i=1}^{N} \{t_i \ln(t_i) + (1 - t_i)\ln(1 - t_i)\}$$

Subtracting this from E^C gives an error term bounded below by zero, i.e., $E^{CE} \geq 0$ where:

$$E^{CE} = E^C - E^C_{\min} = -\sum_{i=1}^{N} \left\{ t_i \ln\left[\frac{\hat{t}_i}{t_i}\right] + (1 - t_i)\ln\left[\frac{1 - \hat{t}_i}{1 - t_i}\right] \right\}$$

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.

Cross-Entropy Error for Multi-Classification

Using a multilayer feedforward neural network for binary classification is relatively straightforward. A network for binary classification only has a single output that estimates the probability that an input pattern belongs to the *yes* class, i.e., $t_i = 1$. In classification problems with more than two mutually exclusive classes, the calculations and network configurations are not as simple.

One approach is to use multiple network outputs, one for each of the C classes. Using this approach, the j-th output for the i-th training pattern, $t_{ij} = 1$, is the estimated probability that the i-th pattern is the network's j-th class, denoted by $\hat{t}_{ij}$. An easy way to estimate these probabilities is to use logistic activation for each output. This ensures that each output satisfies the univariate probability requirements, i.e., $0 \leq \hat{t}_{ij} \leq 1$.

However, since the classification categories are mutually exclusive, each pattern can only be assigned to one of the C classes, which means that the sum of these individual probabilities should always equal 1. However, if each output is the estimated probability for that class, it is very unlikely that $\sum_{j=1}^{C} \hat{t}_{ij} = 1$. In fact, the sum of the individual probability estimates can easily exceed 1 if logistic activation is applied to every output.

Support Vector Machine (SVM) neural networks use this approach with one modification. An SVM network classifies a pattern as belonging to the i-th category if the activation calculation for that category

exceeds a threshold and the other calculations do not exceed this value. That is, the i-th pattern is assigned to the j-th category if and only if $\hat{t}_{ij} > \delta$ and $\hat{t}_{ik} \leq \delta$ for all $k \neq j$, where δ is the threshold. If this does not occur, then the pattern is marked as *unclassified*.

Another approach to multi-class classification problems is to use the softmax activation function developed by Bridle (1990) on the network outputs. This approach produces outputs that conform to the requirements of a multinomial distribution. That is

$$\sum_{j=1}^{C} \hat{t}_{ij} = 1$$

for all $i = 1, 2, \cdots, N$ and $0 \leq \hat{t}_{ij} \leq 1$ for all $i = 1, 2, \cdots, N$ and $j = 1, 2, \cdots, C$ The softmax activation function estimates classification probabilities using the following softmax activation function:

$$\hat{t}_{ij} = \frac{e^{Z_{ij}}}{\sum_{j=1}^{C} e^{Z_{ij}}}$$

where Z_{ij} is the potential for the j-th output perceptron, or category, using the i-th pattern.

For this activation function, it is clear that:

1. $0 \leq \hat{t}_{ij} \leq 1$ for all $i = 1, 2, \cdots, N$ and
2. $\sum_{j=1}^{C} \hat{t}_{ij} = 1$ for all $i = 1, 2, \cdots, N$

Modeling the C network outputs as multinomial observations leads to the cross-entropy error function described by Hopfield (1987) and Bishop (1995):

$$E^C = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \ln(\hat{t}_{ij})$$

where N is the number of training patterns, t_{ij} is the target value for the j-th class of i-th pattern (either 1 or 0), and $\hat{t}_{ij}$ is the neural network's estimate of the j-th output for the i-th pattern. $\hat{t}_{ij}$ is equal to the neural network's estimate of the probability that the i-th pattern should be classified into the j-th category.

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:

$$E^C_{\min} = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \ln(t_{ij})$$

Subtracting this from E^C gives an error term bounded below by zero, i.e., $E^{CE} \geq 0$ where:

$$E^{CE} = E^C - E^C_{\min} = -\sum_{i=1}^{N} \sum_{j=1}^{C} t_{ij} \ln\left[\frac{\hat{t}_{ij}}{t_{ij}}\right]$$

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. That is, when 1-in-C encoding of the target variable is used,

$$t_{ij} = \begin{cases} 1 & \text{if the } i-\text{th pattern belongs to the } j-\text{th category} \\ 0 & \text{if the } i-\text{th pattern does not belong to the } j-\text{th category} \end{cases}$$

Small values, values near zero, 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:

$$E = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{C} (t_{ij} - \hat{t}_{ij})^2$$

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 δ_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	$g(a) = \tanh(a)$	$g'(a) = \text{sech}^2(a) = 1 - \tanh^2(a)$
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:

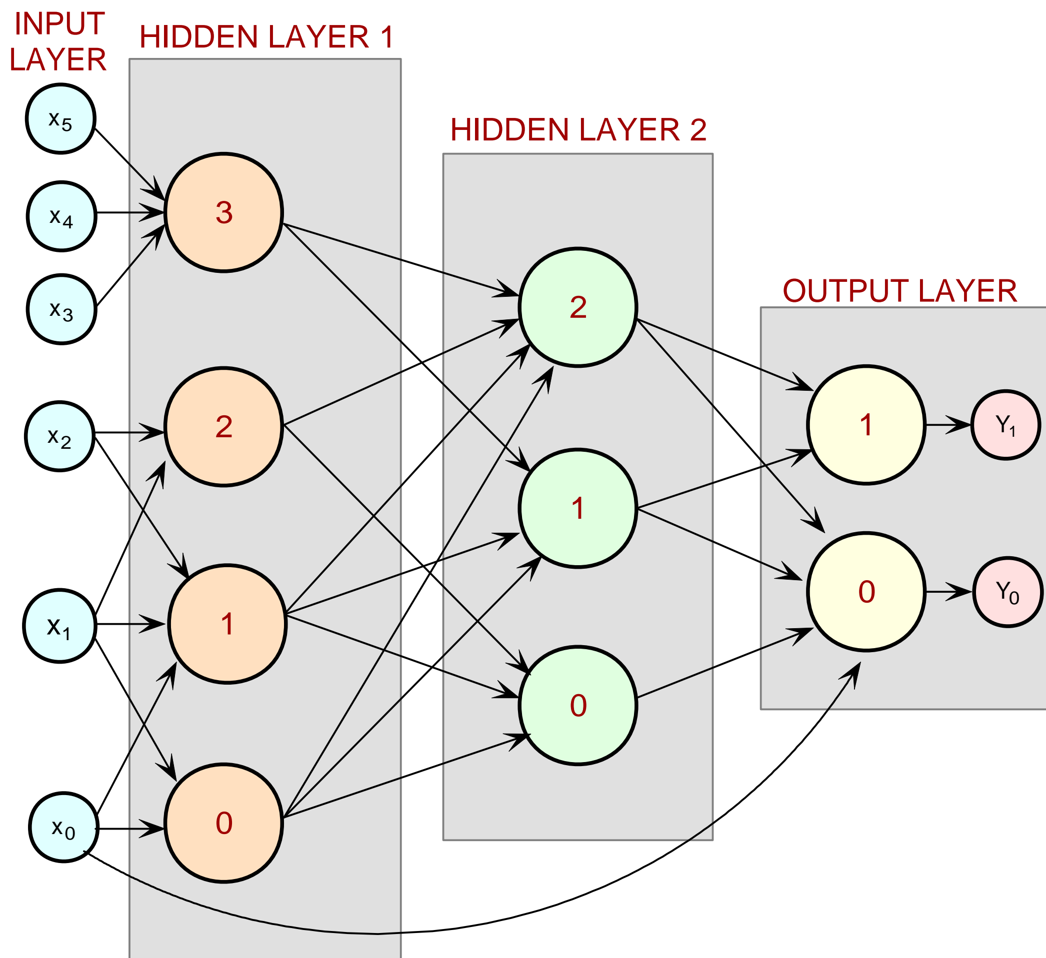


Figure 8. A Three-Layer Feed-Forward Neural Net

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. For this reason, the quasi-Newton trainer is used to solve classification problems.

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.encode(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 Forecasting 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:

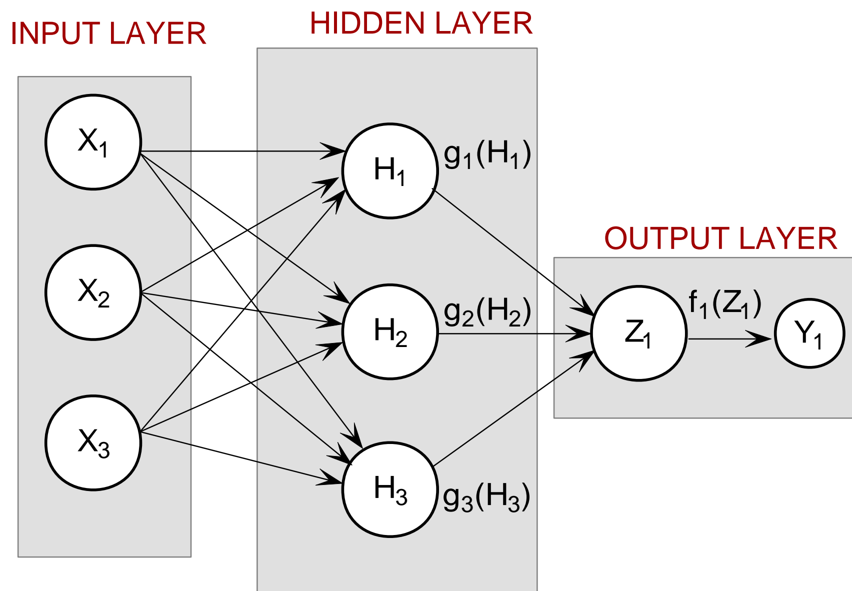


Figure 9. An example 2-layer Feed Forward Neural Network

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 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 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;

// ******************************************************************
// 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\ncurrent error value.  The current solution may "
        + "be an approximate\nsolution and no more accuracy is "
        + "possible, or the step tolerance\nmay be too large.";
private static String errorMsg2
        = "--> Relative function convergence;  both both the actual "
        + "and \npredicted relative reductions in the error function "
        + "are less than\nor equal to the relative fu nction "
        + "convergence tolerance.";
private static String errorMsg3
        = "--> Scaled step tolerance satisfied;  the current solution "
        + "may be\nan approximate local solution, or the algorithm is "
        + "making very slow\nprogress 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 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, 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};
    // *****************************************************************
    // 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];
    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.

Link to Java source.

Links to Input Data Files Used in this Example and the Training Log:

continuous.txt - Input attributes file.
output.txt - Network targets file.
NeuralNetworkEx1.log - Training Log.

Network class

```
abstract public class com.imsl.datamining.neural.Network implements
Serializable
```

Neural network base class.

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

```
abstract public HiddenLayer createHiddenLayer()
```

Description

Creates the next `HiddenLayer` in the `Network`.

Returns

The new `HiddenLayer`.

forecast

```
abstract public double[] forecast(double[] x)
```

Description

Returns a forecast for each of the `Network`'s outputs computed from the trained `Network`.

Parameter

`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

```
abstract public double[][] getForecastGradient(double[] x)
```

Description

Returns the derivatives of the outputs with respect to the *weight*s.

Parameter

`x` – 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 value of `gradient[i][j]` is dy_i/dw_j, where y_i is the i-th output and w_j is the j-th weight.

getInputLayer

```
abstract public InputLayer getInputLayer()
```

Description

Returns the `InputLayer` object.

Returns

The `Network` `InputLayer`.

getLinks

```
abstract public Link[] getLinks()
```

Description

Returns an array containing the `Link` objects in the `Network`.

Returns

An array of `Links` associated with this `Network`.

getNumberOfInputs

```
abstract public int getNumberOfInputs()
```

Description

Returns the number of `Network` inputs.

Returns

An `int` which contains the number of inputs.

getNumberOfLinks

```
abstract public 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

```
abstract public int getNumberOfOutputs()
```

Description

Returns the number of `Network` output `Perceptrons`.

Returns

An `int` which contains the number of outputs.

getNumberOfWeights

```
abstract public 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

```
abstract public OutputLayer getOutputLayer()
```

Description

Returns the `OutputLayer`.

Returns

The `Network` `OutputLayer`.

getPerceptrons

```
abstract public Perceptron[] getPerceptrons()
```

Description

Returns an array containing the `Perceptrons` in the `Network`.

Returns

An array of `Perceptrons` associated with this `Network`.

getWeights

```
abstract public double[] getWeights()
```

Description

Returns the *weights*.

Returns

A `double` array containing the *weights* associated with `Network Links`.

setWeights

```
abstract public void setWeights(double[] weights)
```

Description

Sets the *weights*.

Parameter

`weights` – A `double` array which specifies the *weights* 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:

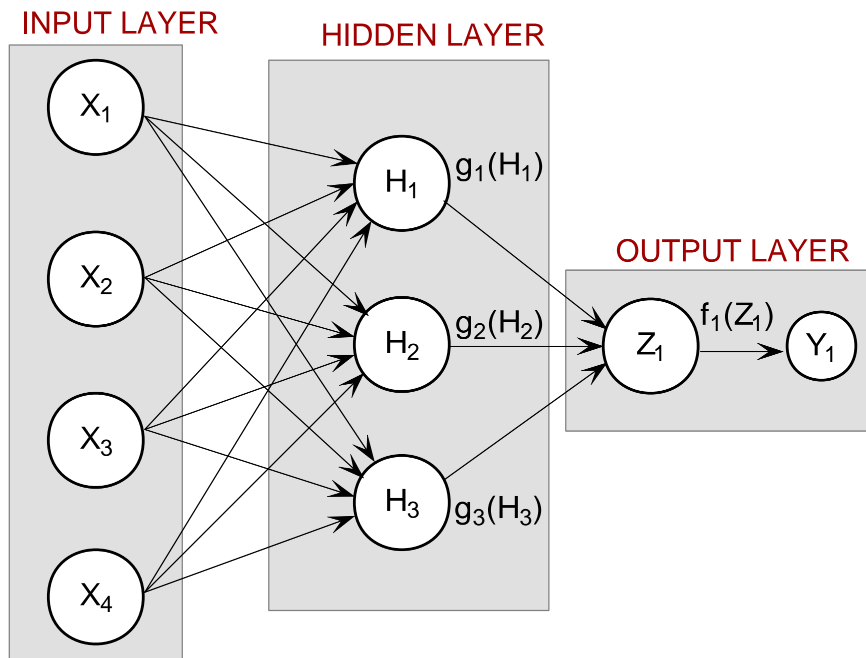


Figure 10. An example 2-layer Feed Forward Neural Network with 4 Inputs

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
```

FeedForwardNetwork class

`public class com.imsl.datamining.neural.FeedForwardNetwork extends com.imsl.datamining.neural.Network`

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` (p. 2090) 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.

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 `Node`s.

Parameters

`from` – The origination `Node`.

`to` – The destination `Node`.

Returns

A `Link` between the two `Node`s, or `null` if no such `Link` exists.

findLinks

`public Link[] findLinks(Node to)`

Description

Returns all of the `Link`s to a given `Node`.

Parameter

`to` – A `Node` who's `Links` are to be determined.

Returns

An array of `Link`s containing all of the `Link`s to the given `Node`.

forecast

`public double[] forecast(double[] x)`

Description

Computes a forecast using the `Network`.

Parameter

`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 derivatives of the outputs with respect to the *weight*s.

Parameter

`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 value of `gradient[i][j]` is dy_i/dw_j, where y_i is the i-th output and w_j is the j-th weight.

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 in 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.

Parameter

`link` – The `Link` deleted from the network.

setEqualWeights

```
public void setEqualWeights(double[][] xData)
```

Description

Initializes network *weights* using equal weighting.

The equal weights approach starts by assigning equal values to the inputs of each `Perceptron`. If a `Perceptron` has 4 inputs, then this method starts by assigning the value 1/4 to each of the `perceptron`'s input *weights*. The *bias* weight is initially assigned a value of zero.

The *weights* for the first `Layer` of `Perceptrons`, either the first `HiddenLayer` if the number of `Layers` is greater than 1 or the `OutputLayer`, are scaled using the training patterns. Scaling is accomplished by dividing the initial *weights* for the first `Layer` by the standard deviation, *s*, of the potential for that `Perceptron`. The *bias* weight is set to *-avg/s*, where *avg* is the average potential for that `Perceptron`. This makes the average potential for the `Perceptrons` in this first `Layer` approximately 0 and its standard deviation equal to 1.

This reduces the possibility of saturation during network training resulting from very large or small values for the `Perceptron`'s potential. During training random noise is added to these intial values at each training stage. If the `EpochTrainer` is used, noise is added to these initial values at the start of each epoch.

Parameter

`xData` – An input `double` matrix containing training patterns. The number of columns in `xData` must equal the number of `Perceptrons` in the `InputLayer`.

setRandomWeights

```
public void setRandomWeights(double[][] xData, Random random)
```

Description

Initializes network *weights* using random weights.

The `RandomWeights` algorithm assigns equal *weights* to all `Perceptrons`, except those in the first `Layer` connected to the `InputLayer`. Like the `EqualWeights` algorithm, `Perceptrons` not in the first `Layer` are assigned *weights* 1/*k*, where *k* is the number of inputs connected to that `Perceptron`.

For the first `Layer Perceptron` *weights*, they are initially assigned values from the uniform random distribution on the interval [-0.5, +0.5]. These are then scaled using the training patterns. The random *weights* for a `Perceptron` are divided by *s*, the standard deviation of the potential for that `Perceptron` calculated using the intial random values. Its *bias* value is set to *-avg/s*, where *avg* is the average potential for that `Perceptron potential for the Perceptrons in this first Layer approximately 0 and its standard deviation equal to 1.`

This reduces the possibility of saturation during network training resulting from very large or small values for the `Perceptron`'s potential. During training random noise is added to these intial values at each training stage. If the `EpochTrainer` is used, noise is added to these initial values at the start of each epoch.

Parameters

`xData` – An input `double` matrix containing training patterns. The number of columns in `xData` must equal the number of `Perceptrons` in the `InputLayer`.

`random` – A `Random` object.

---

setWeights

```
public void setWeights(double[] weights)
```

Description

Sets the *weight*s for the `Links` in this `Network`.

Parameter

`weights` – A `double` array containing the *weight*s in the same order as `com.imsl.datamining.neural.FeedForwardNetwork.getWeights` (p. 2103) .

---

validateLink

```
protected void validateLink(Node from, Node to) throws IllegalArgumentException
```

Description

Checks that a `Link` between two `Nodes` is valid.

In a `FeedForwardNetwork` 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`.

Exception

`IllegalArgumentException` is thrown if the `Link` is not valid

Example: FeedForwardNetwork

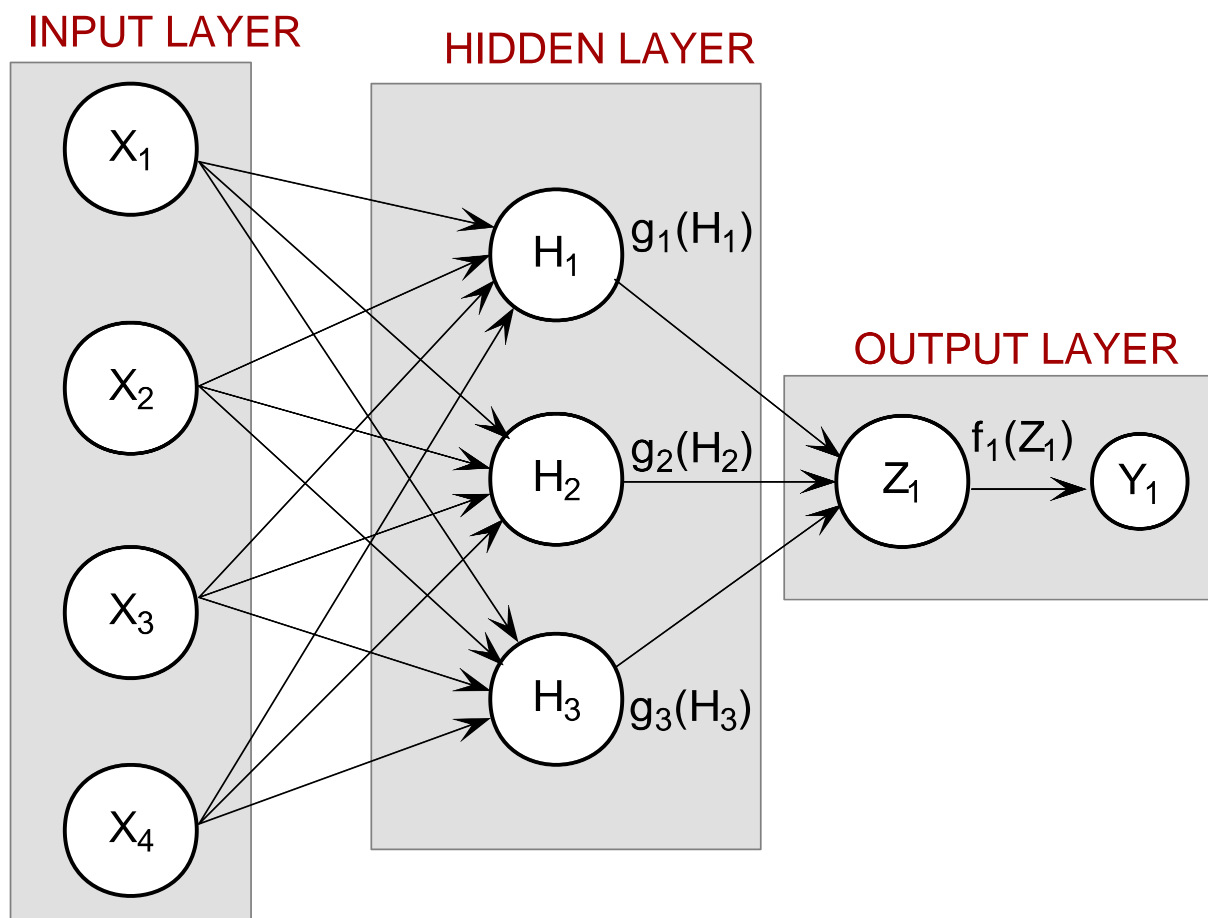
This example trains a 2-layer network using 100 training patterns from one nominal and one continuous input attribute. The nominal attribute has three classifications which are encoded using binary encoding. This results in three binary network input columns. The continuous input attribute is scaled 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 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 \nmay be an approximate local solution, or the"
            + " algorithm is making\nslow progress and is not near a"
            + " solution, or the Step Tolerance\nis too big";
    private static String errorMsg2
            = "--> Scaled actual and predicted reductions in the function"
            + " are\nless than or equal to the relative function"
            + " convergence\ntolerance 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 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];
        }
        // Scale continuous input
        xData[i][nCategorical] = contAtt[i] / 10.0;

        // outputs are unscaled
        yData[i][0] = outs[i];
    }
    // ***************************************************************
```

```
// 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:                       1.013444E-15
--> RMS:                       2.007463E-19
--> Laplacian Error:           3.0058033E-7
--> Scaled Laplacian Error:    3.5352335E-10
--> Largest Absolute Residual: 2.784276E-8
***********************************************

--> Getting Network Weights and Gradients

--> Network Weights and Gradients:
***********************************************
w[0]=-1.4917853 g[0]=-2.6110781E-8
w[1]=-1.4917853 g[1]=-2.6110781E-8
w[2]=-1.4917853 g[2]=-2.6110781E-8
w[3]=1.6169184 g[3]=6.182036E-8
w[4]=1.6169184 g[4]=6.182036E-8
w[5]=1.6169184 g[5]=6.182036E-8
w[6]=4.725622 g[6]=-5.2738493E-8
w[7]=4.725622 g[7]=-5.2738493E-8
w[8]=4.725622 g[8]=-5.2738493E-8
w[9]=6.217407 g[9]=-8.732707E-10
w[10]=6.217407 g[10]=-8.732707E-10
```

```
w[11]=6.217407 g[11]=-8.732707E-10
w[12]=1.0722584 g[12]=-1.6909704E-7
w[13]=1.0722584 g[13]=-1.6909704E-7
w[14]=1.0722584 g[14]=-1.6909704E-7
w[15]=3.8507552 g[15]=-1.7028917E-8
w[16]=3.8507552 g[16]=-1.7028917E-8
w[17]=3.8507552 g[17]=-1.7028917E-8
w[18]=2.4117248 g[18]=-1.5881357E-8
**********************************************

--> Saving Trained Network into FeedForwardNetworkEx1.ser
--> Saving xData into FeedForwardxDataEx1.ser
--> Saving yData into FeedForwardyDataEx1.ser
--> Saving Network Trainer into FeedForwardTrainerEx1.ser
```

Layer class

`abstract public class com.imsl.datamining.neural.Layer implements Serializable`

The base class for `Layers` in a neural network.

Constructor

Layer

`protected Layer(FeedForwardNetwork network)`

Description

Constructs a `Layer`.

Parameter

`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`.

Parameter

`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`.

InputLayer class

```
public class com.imsl.datamining.neural.InputLayer extends
com.imsl.datamining.neural.Layer
```

Input layer in a neural network. An `InputLayer` is automatically created by `Network`.

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.

Parameter

`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`.

HiddenLayer class

```
public class com.imsl.datamining.neural.HiddenLayer extends
com.imsl.datamining.neural.Layer
```

Hidden layer in a neural network. This is created by a factory method in `Network`.

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* weight.

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.

Parameter

`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`.

OutputLayer class

```
public class com.imsl.datamining.neural.OutputLayer extends
com.imsl.datamining.neural.Layer
```

Output layer in a neural network. An empty `OutputLayer` is automatically created by `FeedForwardNetwork`.

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*.

Parameter

`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 (p. 2113) to return the Perceptrons in an OutputPerceptron array.

Returns

An OutputPerceptron[] array containing the Nodes in the OutputLayer.

Node class

```
abstract public class com.imsl.datamining.neural.Node implements Serializable
```

A Node in a neural network.

Node is an abstract class that serves as the base class for the concrete classes InputNode and Perceptron.

Method

getLayer

```
public Layer getLayer()
```

Description

Returns the Layer in which this Node exists.

Returns

The Layer associated with this Node.

InputNode class

```
public class com.imsl.datamining.neural.InputNode extends
com.imsl.datamining.neural.Node
```

A Node in the InputLayer.

InputNodes are not created directly. Instead factory methods in InputLayer are used to create InputNodes within the InputLayer. For example, InputLayer.createInput() creates a single InputNode.

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`.

Parameter

`value` – A `double` which specifies the new value of this `InputNode`.

Perceptron class

`public class com.imsl.datamining.neural.Perceptron extends com.imsl.datamining.neural.Node`

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* (μ). 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.

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.

Parameter

`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`.

Parameter

`bias` – A `double` scalar value to which the *bias* is to be set. The `bias` has a default value of 0.

OutputPerceptron class

```
public class com.imsl.datamining.neural.OutputPerceptron extends
com.imsl.datamining.neural.Perceptron
```

A `Perceptron` in the `OutputLayer`. `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`.

Method

getValue

```
public double getValue()
```

Description

Returns the value of the `OutputPerceptron` determined using the current network state and inputs.

Returns

A `double` value of the `OutputPerceptron` determined using the current network state and inputs.

Activation interface

```
public interface com.imsl.datamining.neural.Activation implements Serializable
```

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 $y = g(x)$.

Fields

LINEAR

```
static final public Activation LINEAR
```

The identity activation function, $g(x) = x$.

LOGISTIC

```
static final public Activation LOGISTIC
```

The logistic activation function, $g(x) = \frac{1}{1+e^{-x}}$.

LOGISTIC_TABLE

```
static final public 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.

SOFTMAX

```
static final public Activation SOFTMAX
```

The softmax activation function.

$$\mathrm{softmax_i} = \frac{\mathrm{e}^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}}$$

.

SQUASH

```
static final public Activation SQUASH
```

The squash activation function, $g(x) = \frac{x}{1+|x|}$

TANH

```
static final public Activation TANH
```

The hyperbolic tangent activation function, $g(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.

Methods

derivative

```
public 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

```
public double g(double x)
```

Description

Returns the value of the activation function.

Parameter

`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`.

Link class

`public class com.imsl.datamining.neural.Link implements Serializable`

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 `FeedForwardNetwork.linkAll` which creates `Link` objects connecting every `Node` in each `Layer` to every `Node` in the next `Layer`.

The method `FeedForwardNetwork.link(Node,Node)` creates a `Link` from a `Node` to any `Node` in a later `Layer`.

The method `FeedForwardNetwork.findLink(Node,Node)` returns the `Link` connecting two `Nodes` in the `Network`.

The method `FeedForwardNetwork.remove(Link)` removes a `Link` from the `Network`.

Each `Link` object contains a *weight*. Weights are used in computing `Perceptron` values.

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`.

Parameter

`weight` – A `double` which specifies the weight to attribute to this `Link`.

Trainer interface

```
public interface com.imsl.datamining.neural.Trainer implements Serializable
```

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.

Methods

getErrorGradient

```
public 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

```
public 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

```
public 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

```
public 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 InputLayer. 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 OutputLayer. Each row of yData contains a training pattern.

QuasiNewtonTrainer class

```
public class com.imsl.datamining.neural.QuasiNewtonTrainer implements
com.imsl.datamining.neural.Trainer, Serializable
```

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.

Field

SUM_OF_SQUARES

`static final public 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

clone

`protected Object clone()`

Description

Clones a copy of the trainer.

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	`MinUnconMultiVar.FalseConvergenceException` thrown by optimizer.
5	`MinUnconMultiVar.MaxIterationsException` thrown by optimizer.
6	`MinUnconMultiVar.UnboundedBelowException` thrown by optimizer.

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

```
static public 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 `java.util.logging.Handler.setFormatter` to format the output log.

Returns

The `Formatter` object, if present, or `null`.

getLogger

```
static public 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.

getUseBackPropagation

```
public boolean getUseBackPropagation()
```

Description

Returns the use back propagation setting.

Returns

a `boolean` specifying whether or not back propagation is being used for gradient calculations.

setEpochNumber

```
protected void setEpochNumber(int num)
```

Description

Sets the epoch number for the trainer.

Parameter

`num` – An `int` array containing the epoch number.

setError

```
public void setError(QuasiNewtonTrainer.Error error)
```

Description

Sets the function used to compute the network error.

Parameter

`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`.

---

setGradientTolerance

```
public void setGradientTolerance(double gradientTolerance)
```

Description

Set the gradient tolerance.

Parameter

`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.

Parameter

`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.

Parameter

`maximumTrainingIterations` – An `int` representing the maximum number of training iterations. Default: 100.

---

setParallelMode

```
protected void setParallelMode(ArrayList[] allLogRecords)
```

Description

Sets the trainer to be used in multi-threaded EpochTainer.

Parameter

`allLogRecords` – An `ArrayList` array containing the log records.

---

setStepTolerance

```
public void setStepTolerance(double stepTolerance)
```

Description

Sets the scaled step tolerance.

The second stopping criterion for `com.imsl.math.MinUnconMultiVar` (p. 357) , the optimizer used by this `Trainer`, is that the scaled distance between the last two steps be less than the step tolerance.

Parameter

`stepTolerance` – A `double` which is the step tolerance. Default: 3.66685e-11.

setUseBackPropagation

```
public void setUseBackPropagation(boolean flag)
```

Description

Sets whether or not to use the back propagation algorithm for gradient calculations during network training.

By default, the quasi-newton algorithm optimizes the network using numerical gradients. This method directs the quasi-newton trainer to use the back propagation algorithm for gradient calculations during network training. Depending upon the data and network architecture, one approach is typically faster than the other, or is less sensitive to finding local network optima.

Parameter

`flag` – `boolean` specifies whether or not to use the back propagation algorithm for gradient calculations. Default value is `true`.

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.

QuasiNewtonTrainer.Error interface

```
public interface com.imsl.datamining.neural.QuasiNewtonTrainer.Error implements
Serializable
```

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, $de/d\hat{y}$.

Methods

error

```
public 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

```
public 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.

QuasiNewtonTrainer.Objective class

```
protected class com.imsl.datamining.neural.QuasiNewtonTrainer.Objective
implements com.imsl.math.MinUnconMultiVar.Function
```

The Objective class is passed to the optimizer.

Fields

nFunctionEvaluations

`protected int nFunctionEvaluations`

nObs

`protected int nObs`

nY

`protected int nY`

Method

f

`public double f(double[] weights)`

QuasiNewtonTrainer.GradObjective class

```
protected class com.imsl.datamining.neural.QuasiNewtonTrainer.GradObjective
extends com.imsl.datamining.neural.QuasiNewtonTrainer.Objective implements
com.imsl.math.MinUnconMultiVar.Gradient
```

The Objective class is passed to the optimizer.

Fields

nFunctionEvaluations

`protected int nFunctionEvaluations`

nObs

`protected int nObs`

nY

`protected int nY`

Method

gradient

```
public void gradient(double[] weights, double[] gradient)
```

QuasiNewtonTrainer.BlockObjective class

```
protected class com.imsl.datamining.neural.QuasiNewtonTrainer.BlockObjective
extends com.imsl.datamining.neural.QuasiNewtonTrainer.Objective
```

Constructor

QuasiNewtonTrainer.BlockObjective

```
protected QuasiNewtonTrainer.BlockObjective()
```

Method

f

```
public double f(double[] weights)
```

QuasiNewtonTrainer.BlockGradObjective class

```
protected class
com.imsl.datamining.neural.QuasiNewtonTrainer.BlockGradObjective extends
com.imsl.datamining.neural.QuasiNewtonTrainer.GradObjective
```

Constructor

QuasiNewtonTrainer.BlockGradObjective

```
protected QuasiNewtonTrainer.BlockGradObjective()
```

Methods

f

```
public double f(double[] weights)
```

gradient

```
public void gradient(double[] weights, double[] gradient)
```

LeastSquaresTrainer class

```
public class com.imsl.datamining.neural.LeastSquaresTrainer implements
com.imsl.datamining.neural.Trainer, Serializable
```

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.

Constructor

LeastSquaresTrainer

```
public LeastSquaresTrainer()
```

Description

Creates a `LeastSquaresTrainer`.

Methods

clone

```
protected Object clone()
```

Description

Clones a copy of the trainer.

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 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

```
static public 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 `java.util.logging.Handler.setFormatter` to format the output log.

Returns

A `Formatter` object, if present, or `null` .

getLogger

```
static public 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` .

setEpochNumber

```
protected void setEpochNumber(int num)
```

Description

Sets the epoch number for the trainer.

Parameter

`num` – An `int` array containing the epoch number.

setFalseConvergenceTolerance

```
public void setFalseConvergenceTolerance(double falseConvergenceTolerance)
```

Description

Set the false convergence tolerance.

Parameter

`falseConvergenceTolerance` – a `double` specifying the false convergence tolerance. Default: 1.0e-14.

setGradientTolerance

```
public void setGradientTolerance(double gradientTolerance)
```

Description

Set the *gradient* tolerance.

Parameter

`gradientTolerance` – A `double` specifying the *gradient* tolerance. Default: 2.0e-5.

setInitialTrustRegion

```
public void setInitialTrustRegion(double initialTrustRegion)
```

Description

Sets the intial trust region.

Parameter

`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.

Parameter

`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.

Parameter

`maximumSolverIterations` – An `int` which specifies the maximum number of iterations to be used by the nonlinear least squares solver. Its default value is 1000.

setParallelMode

```
protected void setParallelMode(ArrayList[] allLogRecords)
```

Description

Sets the trainer to be used in multi-threaded EpochTainer.

Parameter

`allLogRecords` – An `ArrayList` array containing the log records.

setRelativeTolerance

```
public void setRelativeTolerance(double relativeTolerance)
```

Description

Sets the relative tolerance.

Parameter

`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.

Parameter

`stepTolerance` – A `double` which specifies the scaled step tolerance to use when changing the *weight*s. 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`.

EpochTrainer class

```
public class com.imsl.datamining.neural.EpochTrainer implements
com.imsl.datamining.neural.Trainer, Serializable
```

Two-stage training using randomly selected training patterns in stage I. The `EpochTrainer`, 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 II 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 II 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 `Network.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.

Constructors

EpochTrainer

`public EpochTrainer(Trainer stage1Trainer)`

Description

Creates a single stage `EpochTrainer`. Stage 2 training is bypassed.

Parameter

`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

`static public 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 `java.util.logging.Handler.setFormatter` to format the output log.

Returns

The `Formatter` object, if present, or `null` otherwise.

getLogger

`static public Logger getLogger()`

Description

Returns the `Logger` object. This is the `Logger` used to trace this class. It is named `com.imsl.datamining.neural.QuasiNewtonTrainer` (p. 2125).

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.

getNumberOfThreads

`public int getNumberOfThreads()`

Description

Gets the number of `java.lang.Thread` instances to use during stage I training.

Returns

An `int` which contains the number of `java.lang.Thread` instances to use.

getRandom

`public 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.

getRandomSampleIndicies

`protected RandomSampleIndicies getRandomSampleIndicies()`

Description

Gets the random number generators used to select random training patterns in stage 1.

Returns

A `RandomSampleIndicies` containing the random number generators.

getStage1Trainer

```
protected Trainer getStage1Trainer()
```

Description

Returns the stage 1 trainer.

Returns

A `Trainer` containing the stage 1 trainer.

getStage2Trainer

```
protected Trainer getStage2Trainer()
```

Description

Returns the stage 1 trainer.

Returns

A `Trainer` containing the stage 2 trainer.

incrementEpochCount

```
protected int incrementEpochCount()
```

Description

Increments the epoch counter.

setEpochSize

```
public void setEpochSize(int epochSize)
```

Description

Sets the number of randomly selected training patterns in stage 1 epoch.

Parameter

`epochSize` – An `int` which specifies the number of sample training patterns in each stage I epoch. The default value is the number of observations in the training data.

setNumberOfEpochs

```
public void setNumberOfEpochs(int numberOfEpochs)
```

Description

Sets the number of epochs.

Parameter

numberOfEpochs – An int which specifies the number of epochs to be used during stage I training. The default value is 10.

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the number of java.lang.Thread instances to be used for parallel processing.

Parameter

numberOfThreads – an int specifying the number of java.lang.Thread instances to be used for parallel processing.

Default: numberOfThreads = 1.

setRandom

```
public void setRandom(Random random)
```

Description

Sets the random number generator used to perturb the initial stage 1 guesses.

Parameter

random – The Random object used to set the random number generator.

setRandomSamples

```
public void setRandomSamples(Random randomA, 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.

BinaryClassification class

```
public class com.imsl.datamining.neural.BinaryClassification implements
Serializable
```

Classifies patterns into two classes.

Uses a FeedForwardNetwork to solve binary classification problems. In these problems, the target output for the network is the probability that the pattern falls into one of two classes. The first class, $P(C_1)$, is usually equal to one and the second class, $P(C_2)$ equal to zero. These probabilities are then used to assign patterns to one of the two classes. Typical applications include determining whether a credit applicant is a good or bad credit risk, and determining whether a person should or should not receive a particular treatment based upon their physical, clinical and laboratory information. This class signals that network training will minimize the binary cross-entropy error, and that network output is the probability that the pattern belongs to the first class, $P(C_1)$. Which is calculated by applying the logistic activation function to the potential of the single output. The probability for the second class is calculated by $P(C_2) = 1 - P(C_1)$.

Constructor

BinaryClassification

```
public BinaryClassification(Network network)
```

Description

Creates a binary classifier.

Parameter

network – is the neural network used for classification. Its OutputPerceptrons should use the logistic activation function, Activation.LOGISTIC.

Methods

computeStatistics

```
public double[] computeStatistics(double[][] xData, int[] yData)
```

Description

Computes the classification error statistics for the supplied network patterns and their associated classifications.

The first element returned is the binary cross-entropy error; the second is the classification error rate. The classification error rate is calculated by comparing the estimated classification probabilities to the target classifications. If the estimated probability for the target class is less than 0.5, then this is tallied as a classification error.

Parameters

`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` – An `int` containing the output classification patterns. The values in `yData` must be 0 or 1.

Returns

A two-element `double` array containing the binary cross-entropy error and the classification error rate.

getError

```
public QuasiNewtonTrainer.Error getError()
```

Description

Returns the error function for use by `QuasiNewtonTrainer` for training a binary classification network.

Returns

an implementation of the binary-entropy error function.

getNetwork

```
public Network getNetwork()
```

Description

Returns the network being used for classification.

Returns

the network set by the constructor.

predictedClass

```
public int predictedClass(double[] x)
```

Description

Calculates the classification probablities for the input pattern `x`, and returns either 0 or 1 identifying the class with the highest probability.

This method is used to classify patterns into one of the two target classes based upon the pattern's values. The predicted classification is the class with the largest probability, i.e. greater than 0.5.

Parameter

`x` – the `double` array containing the network input patterns to classify. The length of `x` should be equal to the number of inputs in the network.

Returns

The classification predicted by the trained network for `x`. This will be either 0 or 1.

probabilities

```
public double[] probabilities(double[] x)
```

Description

Returns classification probabilities for the input pattern `x`.

Calculates the two probabilities for the pattern supplied: $P(C_1)$ and $P(C_2)$. The probability that the pattern belongs to the first class, $P(C_1)$, is estmated using the logistic function of the `OutputPerceptron`'s potential. The probability for the second class is claculated as $P(C_2) = 1 - P(C_1)$. The predicted classification is the class with the largest probability, i.e. greater than 0.5.

Parameter

`x` – a `double` array containing the network input pattern to classify. The length of `x` must equal the number of nodes in the input layer.

Returns

the probability of `x` being in class C_1, followed by the probability of `x` being in class C_2.

train

```
public void train(Trainer trainer, double[][] xData, int[] yData)
```

Description

Trains the classification neural network using supplied trainer and patterns.

Parameters

`trainer` – A `Trainer` object, which is used to train the network. The error function in any `QuasiNewton` trainer included in `trainer` should be set to the error function from this class using the `getError` method provided by this class.

`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` – An `int` array containing the output classification values. These values must be 0 or 1.

Example 1: Binary Classification

This example trains a 3-layer network using 48 training patterns from four nominal input attributes. The first two nominal attributes have two classifications. The third and fourth nominal attributes have three and four classifications respectively. All four attributes are encoded using binary encoding. This results in eleven binary network input columns. The output class is 1 if the first two nominal attributes sum to 1, and 0 otherwise.

The structure of the network consists of eleven input nodes and three layers, with three perceptrons in the first hidden layer, two perceptrons in the second hidden layer, and one perceptron in the output layer.

There are a total of 47 weights in this network, including the six bias weights. The linearactivations function is used for both hidden layers. Since the target output is binary classification the logistic activation function is used in the output layer. Training is conducted using the quasi-newton trainer with the binary-entropy error function provided by the `BinaryClassification` class.

```
import com.imsl.datamining.neural.*;
import java.io.*;
import java.util.logging.*;
import com.imsl.math.*;
import java.util.Random;

//*****************************************************************************
// Two Layer Feed-Forward Network with 11 inputs: 4 nominal with 2,2,3,4
// categories, encoded using binary encoding,  and 1 output target (class).
//
//  new classification training_ex1.c
//*****************************************************************************
public class BinaryClassificationEx1 implements Serializable {

    // Network Settings
    private static int nObs = 48;        // number of training patterns
    private static int nInputs = 11;  // four nominal with 2,2,3,4 categories
    private static int nCategorical = 11;// three categorical attributes
    private static int nOutputs = 1;     // one continuous output (nClasses=2)
    private static int nPerceptrons1 = 3;// perceptrons in 1st hidden layer
    private static int nPerceptrons2 = 2;// perceptrons in 2nd hidden layer
    private static boolean trace = true; // Turns on/off training log

    private static Activation hiddenLayerActivation = Activation.LINEAR;
    private static Activation outputLayerActivation = Activation.LOGISTIC;

    /* 2 classifications */
    private static int[] x1 = {
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
    };

    /* 2 classifications */
    private static int[] x2 = {
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2
    };

    /* 3 classifications */
    private static int[] x3 = {
        1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 1, 1, 1, 1,
        2, 2, 2, 2, 3, 3, 3, 3, 1, 1, 1, 1, 2, 2, 2, 2,
        3, 3, 3, 3, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3
    };

    /* 4 classifications */
    private static int[] x4 = {
        1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4,
        1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4,
        1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4
    };

    // ************************************************************************
    // MAIN
```

```
// **********************************************************************
public static void main(String[] args) throws Exception {
    double xData[][]; // Input  Attributes for Trainer
    int yData[]; // Output Attributes for Trainer
    int i, j;         // array indicies
    String trainLogName = "BinaryClassificationExample.log";

    // ******************************************************************
    // Binary encode 4 categorical variables.
    //        Var x1 contains 2 classes
    //        Var x2 contains 2 classes
    //        Var x3 contains 3 classes
    //        Var x4 contains 4 classes
    // *******************************************************************
    UnsupervisedNominalFilter filter = new UnsupervisedNominalFilter(2);
    int[][] z1 = filter.encode(x1);
    int[][] z2 = filter.encode(x2);
    filter = new UnsupervisedNominalFilter(3);
    int[][] z3 = filter.encode(x3);
    filter = new UnsupervisedNominalFilter(4);
    int[][] z4 = filter.encode(x4);

    /* Concatenate binary encoded z's */
    xData = new double[nObs][nInputs];
    yData = new int[nObs];
    for (i = 0; i < (nObs); i++) {
        for (j = 0; j < nCategorical; j++) {
            xData[i][j] = 0;
            if (j < 2) {
                xData[i][j] = (double) z1[i][j];
            }
            if (j > 1 && j < 4) {
                xData[i][j] = (double) z2[i][j - 2];
            }
            if (j > 3 && j < 7) {
                xData[i][j] = (double) z3[i][j - 4];
            }
            if (j > 6) {
                xData[i][j] = (double) z4[i][j - 7];
            }
        }
        yData[i] = ((x1[i] + x2[i] == 2) ? 1 : 0);
    }

    // ******************************************************************
    // CREATE FEEDFORWARD NETWORK
    // ******************************************************************
    long t0 = System.currentTimeMillis();

    FeedForwardNetwork network = new FeedForwardNetwork();
    network.getInputLayer().createInputs(nInputs);
    network.createHiddenLayer().createPerceptrons(nPerceptrons1);
    network.createHiddenLayer().createPerceptrons(nPerceptrons2);
    network.getOutputLayer().createPerceptrons(nOutputs);

    BinaryClassification classification
```

```
        = new BinaryClassification(network);

network.linkAll();
Random r = new Random(123457L);
network.setRandomWeights(xData, r);
Perceptron perceptrons[] = network.getPerceptrons();
for (i = 0; i < perceptrons.length - 1; i++) {
    perceptrons[i].setActivation(hiddenLayerActivation);
}
perceptrons[perceptrons.length - 1].
        setActivation(outputLayerActivation);

// **********************************************************************
// TRAIN NETWORK USING QUASI-NEWTON TRAINER
// **********************************************************************
QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
trainer.setError(classification.getError());
trainer.setMaximumTrainingIterations(1000);
trainer.setMaximumStepsize(3.0);
trainer.setGradientTolerance(1.0e-20);
trainer.setFalseConvergenceTolerance(1.0e-20);
trainer.setStepTolerance(1.0e-20);
trainer.setRelativeTolerance(1.0e-20);
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.");
    }
}
classification.train(trainer, xData, yData);

// **********************************************************************
// DISPLAY TRAINING STATISTICS
// **********************************************************************
double stats[] = classification.computeStatistics(xData, yData);
System.out.println("***********************************************");
System.out.println("--> Cross-entropy error:        "
        + (float) stats[0]);
System.out.println("--> Classification error rate:  "
        + (float) stats[1]);
System.out.println("***********************************************");
System.out.println("");
// **********************************************************************
// OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS
// **********************************************************************
double weight[] = network.getWeights();
double gradient[] = trainer.getErrorGradient();
double wg[][] = new double[weight.length][2];
for (i = 0; i < weight.length; i++) {
```

```
            wg[i][0] = weight[i];
            wg[i][1] = gradient[i];
        }
        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new java.text.DecimalFormat("0.000000"));
        pmf.setColumnLabels(new String[]{"Weights", "Gradients"});
        new PrintMatrix().print(pmf, wg);

        //  ****************************
        //     forecast the network
        //  ****************************
        double report[][] = new double[nObs][6];
        for (i = 0; i < nObs; i++) {
            report[i][0] = x1[i];
            report[i][1] = x2[i];
            report[i][2] = x3[i];
            report[i][3] = x4[i];
            report[i][4] = yData[i];
            report[i][5] = classification.predictedClass(xData[i]);
        }
        pmf = new PrintMatrixFormat();
        pmf.setColumnLabels(new String[]{
            "X1", "X2", "X3", "X4",
            "Expected", "Predicted"}
        );
        new PrintMatrix("Forecast").print(pmf, report);

        // **************************************************************
        // DISPLAY CLASSIFICATION STATISTICS
        // **************************************************************
        double statsClass[] = classification.computeStatistics(xData, yData);
        // Display Network Errors
        System.out.println("***********************************************");
        System.out.println("--> Cross-Entropy Error:      "
                + (float) statsClass[0]);
        System.out.println("--> Classification Error:     "
                + (float) statsClass[1]);
        System.out.println("***********************************************");
        System.out.println("");

        long t1 = System.currentTimeMillis();
        double time = t1 - t0;
        time = time / 1000;
        System.out.println("****************Time:  " + time);
        System.out.println("trainer.getErrorValue = "
                + trainer.getErrorValue());
    }
}
```

Output

```
--> Training Log Created in BinaryClassificationExample.log
***********************************************
--> Cross-entropy error:        1.8296475E-13
--> Classification error rate:  0.0
```

```
**********************************************

     Weights   Gradients
 0   2.575599  -0.000000
 1   1.770546  -0.000000
 2   1.675687  -0.000000
 3  -5.859796   0.000000
 4  -1.794721   0.000000
 5  -4.925026   0.000000
 6   3.654187   0.000000
 7   2.089872   0.000000
 8   2.485173   0.000000
 9  -5.238608   0.000000
10  -1.396975   0.000000
11  -4.730949   0.000000
12   0.143083   0.000000
13   0.777367   0.000000
14   0.316769   0.000000
15  -3.270781  -0.000000
16   0.283153  -0.000000
17  -0.162338  -0.000000
18   1.153316   0.000000
19   0.782549   0.000000
20  -0.387279   0.000000
21  -2.010958  -0.000000
22   0.273662  -0.000000
23  -0.670019  -0.000000
24   2.096144   0.000000
25  -0.264374   0.000000
26   0.351305   0.000000
27   1.190361   0.000000
28  -0.053966   0.000000
29   0.555192   0.000000
30  -2.001125  -0.000000
31   0.735950  -0.000000
32  -0.829534  -0.000000
33  -4.824521   0.000000
34  -4.824521   0.000000
35  -0.652606   0.000000
36  -0.652606   0.000000
37  -2.921224   0.000000
38  -2.921224   0.000000
39  -1.621591   0.000000
40  -1.621591   0.000000
41  -1.967947   0.000000
42   1.534864   0.000000
43   0.907830   0.000000
44   1.594078  -0.000000
45   1.594078  -0.000000
46  -0.169361   0.000000

                Forecast
    X1  X2  X3  X4  Expected  Predicted
 0  1   1   1   1      1          1
 1  1   1   1   2      1          1
 2  1   1   1   3      1          1
```

```
 3  1   1   1   4      1          1
 4  1   1   2   1      1          1
 5  1   1   2   2      1          1
 6  1   1   2   3      1          1
 7  1   1   2   4      1          1
 8  1   1   3   1      1          1
 9  1   1   3   2      1          1
10  1   1   3   3      1          1
11  1   1   3   4      1          1
12  1   2   1   1      0          0
13  1   2   1   2      0          0
14  1   2   1   3      0          0
15  1   2   1   4      0          0
16  1   2   2   1      0          0
17  1   2   2   2      0          0
18  1   2   2   3      0          0
19  1   2   2   4      0          0
20  1   2   3   1      0          0
21  1   2   3   2      0          0
22  1   2   3   3      0          0
23  1   2   3   4      0          0
24  2   1   1   1      0          0
25  2   1   1   2      0          0
26  2   1   1   3      0          0
27  2   1   1   4      0          0
28  2   1   2   1      0          0
29  2   1   2   2      0          0
30  2   1   2   3      0          0
31  2   1   2   4      0          0
32  2   1   3   1      0          0
33  2   1   3   2      0          0
34  2   1   3   3      0          0
35  2   1   3   4      0          0
36  2   2   1   1      0          0
37  2   2   1   2      0          0
38  2   2   1   3      0          0
39  2   2   1   4      0          0
40  2   2   2   1      0          0
41  2   2   2   2      0          0
42  2   2   2   3      0          0
43  2   2   2   4      0          0
44  2   2   3   1      0          0
45  2   2   3   2      0          0
46  2   2   3   3      0          0
47  2   2   3   4      0          0

**********************************************
--> Cross-Entropy Error:      1.8296475E-13
--> Classification Error:     0.0
**********************************************

****************Time:  0.062
trainer.getErrorValue = 1.8296475445823478E-13
```

Example 2: Binary Classification Network

This example uses a database of a complete set of possible board configurations at the end of tic-tac-toe games, where "x" is assumed to have played first. The target concept is "win for x" (i.e., true when "x" has one of 8 possible ways to create a "three-in-a-row").

There are nine nominal input attributes for each square on the tic-tac-toe board and are encoded such that 0=player x has taken, 1=player o has taken, 2=blank.

Input attributes

1. top-left-square: {x,o,b}
2. top-middle-square: {x,o,b}
3. top-right-square: {x,o,b}
4. middle-left-square: {x,o,b}
5. middle-middle-square: {x,o,b}
6. middle-right-square: {x,o,b}
7. bottom-left-square: {x,o,b}
8. bottom-middle-square: {x,o,b}
9. bottom-right-square: {x,o,b}

The predicted atribute is a win or loose at tic-tac-toe. For this example the first 626 observations are a win and the next 332 are loss.

The structure of the network consists of 27 input nodes and three layers, with five perceptrons in the first hidden layer, three perceptrons in the second hidden layer, and one perceptron in the output layer.

There are a total of 162 weights in this network. The activations functions are logistic for all layers. Since the target output is binary classification the logistic activation function must be used in the output layer. Training is conducted using the quasi-newton trainer using the binary entropy error function provided by the `BinaryClassification` class.

```
import com.imsl.datamining.neural.*;
import java.io.*;
import java.util.logging.*;
import com.imsl.math.*;
import com.imsl.stat.Random;

//*****************************************************************************
// Three Layer Feed-Forward Network with 4 inputs, all
// continuous, and 2 classification categories.
//
//  new classification training_ex4.c
//
// Three Layer Feed-Forward Network with 4 inputs, all
// continuous, and 2 classification categories.
```

```
//
//  This database encodes the complete set of possible board configurations
//   at the end of tic-tac-toe games, where "x" is assumed to have played
//   first.  The target concept is "win for x" (i.e., true when "x" has one
//   of 8 possible ways to create a "three-in-a-row").
//
//  Predicted attribute: win or loose at tic-tac-toe
//     First 626 obs are positive (win) and the next 332 are negative (loss)
//
//  Input Attributes (10 categorical Attributes)
//     Attribute Information:
//        (0=player x has taken, 1=player o has taken, 2=blank)
//
//    1. top-left-square: {x,o,b}
//    2. top-middle-square: {x,o,b}
//    3. top-right-square: {x,o,b}
//    4. middle-left-square: {x,o,b}
//    5. middle-middle-square: {x,o,b}
//    6. middle-right-square: {x,o,b}
//    7. bottom-left-square: {x,o,b}
//    8. bottom-middle-square: {x,o,b}
//    9. bottom-right-square: {x,o,b}
//   10. Class: {positive,negative}
//*****************************************************************************
public class BinaryClassificationEx2 implements Serializable {

    private static int nObs = 958;       // number of training patterns
    private static int nInputs = 27;   // 9 nominal coded as 0=x, 1=O, 2=blank
    private static int nCategorical = 27;// seven categorical attributes
    private static int nContinuous = 0;  // two continuous input attribute
    private static int nOutputs = 1;     // one continuous output (nClasses=2)
    private static int nLayers = 3;      // number of perceptron layers
    private static int nPerceptrons1 = 5;// perceptrons in 1st hidden layer
    private static int nPerceptrons2 = 3;// perceptrons in 2nd hidden layer
    private static boolean trace = true; // Turns on/off training log

    private static Activation hiddenLayerActivation = Activation.LOGISTIC;
    private static Activation outputLayerActivation = Activation.LOGISTIC;

    private static int[][] data = {
        {0, 0, 0, 0, 1, 1, 0, 1, 1}, {0, 0, 0, 0, 1, 1, 1, 0, 1},
        {0, 0, 0, 0, 1, 1, 1, 1, 0}, {0, 0, 0, 0, 1, 1, 1, 2, 2},
        {0, 0, 0, 0, 1, 1, 2, 1, 2}, {0, 0, 0, 0, 1, 1, 2, 2, 1},
        {0, 0, 0, 0, 1, 2, 1, 1, 2}, {0, 0, 0, 0, 1, 2, 1, 2, 1},
        {0, 0, 0, 0, 1, 2, 2, 1, 1}, {0, 0, 0, 0, 2, 1, 1, 1, 2},
        {0, 0, 0, 0, 2, 1, 1, 2, 1}, {0, 0, 0, 0, 2, 1, 2, 1, 1},
        {0, 0, 0, 1, 0, 1, 0, 1, 1}, {0, 0, 0, 1, 0, 1, 1, 0, 1},
        {0, 0, 0, 1, 0, 1, 1, 1, 0}, {0, 0, 0, 1, 0, 1, 1, 2, 2},
        {0, 0, 0, 1, 0, 1, 2, 1, 2}, {0, 0, 0, 1, 0, 1, 2, 2, 1},
        {0, 0, 0, 1, 0, 2, 1, 1, 2}, {0, 0, 0, 1, 0, 2, 1, 2, 1},
        {0, 0, 0, 1, 0, 2, 2, 1, 1}, {0, 0, 0, 1, 1, 0, 0, 1, 1},
        {0, 0, 0, 1, 1, 0, 1, 0, 1}, {0, 0, 0, 1, 1, 0, 1, 1, 0},
        {0, 0, 0, 1, 1, 0, 1, 2, 2}, {0, 0, 0, 1, 1, 0, 2, 1, 2},
        {0, 0, 0, 1, 1, 0, 2, 2, 1}, {0, 0, 0, 1, 1, 2, 0, 1, 2},
        {0, 0, 0, 1, 1, 2, 0, 2, 1}, {0, 0, 0, 1, 1, 2, 1, 0, 2},
        {0, 0, 0, 1, 1, 2, 1, 2, 0}, {0, 0, 0, 1, 1, 2, 2, 0, 1},
```

```
{0, 0, 0, 1, 1, 2, 2, 1, 0}, {0, 0, 0, 1, 1, 2, 2, 2, 2},
{0, 0, 0, 1, 2, 0, 1, 1, 2}, {0, 0, 0, 1, 2, 0, 1, 2, 1},
{0, 0, 0, 1, 2, 0, 2, 1, 1}, {0, 0, 0, 1, 2, 1, 0, 1, 2},
{0, 0, 0, 1, 2, 1, 0, 2, 1}, {0, 0, 0, 1, 2, 1, 1, 0, 2},
{0, 0, 0, 1, 2, 1, 1, 2, 0}, {0, 0, 0, 1, 2, 1, 2, 0, 1},
{0, 0, 0, 1, 2, 1, 2, 1, 0}, {0, 0, 0, 1, 2, 1, 2, 2, 2},
{0, 0, 0, 1, 2, 2, 0, 1, 1}, {0, 0, 0, 1, 2, 2, 1, 0, 1},
{0, 0, 0, 1, 2, 2, 1, 1, 0}, {0, 0, 0, 1, 2, 2, 1, 2, 2},
{0, 0, 0, 1, 2, 2, 2, 1, 2}, {0, 0, 0, 1, 2, 2, 2, 2, 1},
{0, 0, 0, 2, 0, 1, 1, 1, 2}, {0, 0, 0, 2, 0, 1, 1, 2, 1},
{0, 0, 0, 2, 0, 1, 2, 1, 1}, {0, 0, 0, 2, 1, 0, 1, 1, 2},
{0, 0, 0, 2, 1, 0, 1, 2, 1}, {0, 0, 0, 2, 1, 0, 2, 1, 1},
{0, 0, 0, 2, 1, 1, 0, 1, 2}, {0, 0, 0, 2, 1, 1, 0, 2, 1},
{0, 0, 0, 2, 1, 1, 1, 0, 2}, {0, 0, 0, 2, 1, 1, 1, 2, 0},
{0, 0, 0, 2, 1, 1, 2, 0, 1}, {0, 0, 0, 2, 1, 1, 2, 1, 0},
{0, 0, 0, 2, 1, 1, 2, 2, 2}, {0, 0, 0, 2, 1, 2, 0, 1, 1},
{0, 0, 0, 2, 1, 2, 1, 0, 1}, {0, 0, 0, 2, 1, 2, 1, 1, 0},
{0, 0, 0, 2, 1, 2, 1, 2, 2}, {0, 0, 0, 2, 1, 2, 2, 1, 2},
{0, 0, 0, 2, 1, 2, 2, 2, 1}, {0, 0, 0, 2, 2, 1, 0, 1, 1},
{0, 0, 0, 2, 2, 1, 1, 0, 1}, {0, 0, 0, 2, 2, 1, 1, 1, 0},
{0, 0, 0, 2, 2, 1, 1, 2, 2}, {0, 0, 0, 2, 2, 1, 2, 1, 2},
{0, 0, 0, 2, 2, 1, 2, 2, 1}, {0, 0, 0, 2, 2, 2, 1, 1, 2},
{0, 0, 0, 2, 2, 2, 1, 2, 1}, {0, 0, 0, 2, 2, 2, 2, 1, 1},
{0, 0, 1, 0, 0, 1, 1, 1, 0}, {0, 0, 1, 0, 1, 0, 0, 1, 1},
{0, 0, 1, 0, 1, 1, 0, 1, 0}, {0, 0, 1, 0, 1, 1, 0, 2, 2},
{0, 0, 1, 0, 1, 2, 0, 1, 2}, {0, 0, 1, 0, 1, 2, 0, 2, 1},
{0, 0, 1, 0, 2, 1, 0, 1, 2}, {0, 0, 1, 0, 2, 2, 0, 1, 1},
{0, 0, 1, 1, 0, 0, 1, 0, 1}, {0, 0, 1, 1, 0, 0, 1, 1, 0},
{0, 0, 1, 1, 0, 1, 0, 1, 0}, {0, 0, 1, 1, 0, 1, 1, 0, 0},
{0, 0, 1, 1, 0, 1, 2, 0, 2}, {0, 0, 1, 1, 0, 1, 2, 2, 0},
{0, 0, 1, 1, 0, 2, 1, 0, 2}, {0, 0, 1, 1, 0, 2, 1, 2, 0},
{0, 0, 1, 1, 0, 2, 2, 0, 1}, {0, 0, 1, 1, 0, 2, 2, 1, 0},
{0, 0, 1, 2, 0, 1, 1, 0, 2}, {0, 0, 1, 2, 0, 1, 1, 2, 0},
{0, 0, 1, 2, 0, 1, 2, 1, 0}, {0, 0, 1, 2, 0, 2, 1, 0, 1},
{0, 0, 1, 2, 0, 2, 1, 1, 0}, {0, 0, 2, 0, 1, 1, 0, 1, 2},
{0, 0, 2, 0, 1, 1, 0, 2, 1}, {0, 0, 2, 0, 1, 2, 0, 1, 1},
{0, 0, 2, 0, 2, 1, 0, 1, 1}, {0, 0, 2, 1, 0, 1, 1, 0, 2},
{0, 0, 2, 1, 0, 1, 1, 2, 0}, {0, 0, 2, 1, 0, 1, 2, 0, 1},
{0, 0, 2, 1, 0, 1, 2, 1, 0}, {0, 0, 2, 1, 0, 2, 1, 0, 1},
{0, 0, 2, 1, 0, 2, 1, 1, 0}, {0, 0, 2, 2, 0, 1, 1, 0, 1},
{0, 0, 2, 2, 0, 1, 1, 1, 0}, {0, 1, 0, 0, 0, 1, 0, 1, 1},
{0, 1, 0, 0, 0, 1, 1, 1, 0}, {0, 1, 0, 0, 1, 1, 0, 0, 1},
{0, 1, 0, 0, 1, 1, 0, 2, 2}, {0, 1, 0, 0, 1, 2, 0, 2, 1},
{0, 1, 0, 0, 2, 1, 0, 1, 2}, {0, 1, 0, 0, 2, 1, 0, 2, 1},
{0, 1, 0, 0, 2, 2, 0, 1, 1}, {0, 1, 0, 1, 0, 0, 0, 1, 1},
{0, 1, 0, 1, 0, 0, 1, 1, 0}, {0, 1, 0, 1, 0, 1, 0, 0, 1},
{0, 1, 0, 1, 0, 1, 0, 1, 0}, {0, 1, 0, 1, 0, 1, 0, 2, 2},
{0, 1, 0, 1, 0, 1, 1, 0, 0}, {0, 1, 0, 1, 0, 1, 2, 2, 0},
{0, 1, 0, 1, 0, 2, 0, 1, 2}, {0, 1, 0, 1, 0, 2, 0, 2, 1},
{0, 1, 0, 1, 0, 2, 1, 2, 0}, {0, 1, 0, 1, 0, 2, 2, 1, 0},
{0, 1, 0, 1, 1, 0, 1, 0, 0}, {0, 1, 0, 1, 1, 0, 2, 2, 0},
{0, 1, 0, 1, 2, 0, 1, 2, 0}, {0, 1, 0, 1, 2, 0, 2, 1, 0},
{0, 1, 0, 2, 0, 1, 0, 1, 2}, {0, 1, 0, 2, 0, 1, 0, 2, 1},
{0, 1, 0, 2, 0, 1, 1, 2, 0}, {0, 1, 0, 2, 0, 1, 2, 1, 0},
{0, 1, 0, 2, 0, 2, 0, 1, 1}, {0, 1, 0, 2, 0, 2, 1, 1, 0},
{0, 1, 0, 2, 1, 0, 1, 2, 0}, {0, 1, 0, 2, 2, 0, 1, 1, 0},
```

```
{0, 1, 1, 0, 0, 0, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 1, 0, 1},
{0, 1, 1, 0, 0, 0, 1, 1, 0}, {0, 1, 1, 0, 0, 0, 1, 2, 2},
{0, 1, 1, 0, 0, 0, 2, 1, 2}, {0, 1, 1, 0, 0, 0, 2, 2, 1},
{0, 1, 1, 0, 0, 1, 0, 1, 0}, {0, 1, 1, 0, 0, 1, 0, 2, 2},
{0, 1, 1, 0, 0, 1, 1, 0, 0}, {0, 1, 1, 0, 0, 1, 2, 2, 0},
{0, 1, 1, 0, 0, 2, 0, 1, 2}, {0, 1, 1, 0, 0, 2, 0, 2, 1},
{0, 1, 1, 0, 0, 2, 1, 2, 0}, {0, 1, 1, 0, 0, 2, 2, 1, 0},
{0, 1, 1, 0, 1, 0, 0, 0, 1}, {0, 1, 1, 0, 1, 0, 0, 2, 2},
{0, 1, 1, 0, 1, 1, 0, 0, 0}, {0, 1, 1, 0, 1, 2, 0, 0, 2},
{0, 1, 1, 0, 1, 2, 0, 2, 0}, {0, 1, 1, 0, 2, 0, 0, 1, 2},
{0, 1, 1, 0, 2, 0, 0, 2, 1}, {0, 1, 1, 0, 2, 1, 0, 0, 2},
{0, 1, 1, 0, 2, 1, 0, 2, 0}, {0, 1, 1, 0, 2, 2, 0, 0, 1},
{0, 1, 1, 0, 2, 2, 0, 1, 0}, {0, 1, 1, 0, 2, 2, 0, 2, 2},
{0, 1, 1, 1, 0, 0, 0, 1, 0}, {0, 1, 1, 1, 0, 0, 1, 0, 0},
{0, 1, 1, 1, 0, 0, 2, 2, 0}, {0, 1, 1, 1, 0, 1, 0, 0, 0},
{0, 1, 1, 1, 0, 2, 0, 2, 0}, {0, 1, 1, 1, 0, 2, 2, 0, 0},
{0, 1, 1, 1, 1, 0, 0, 0, 0}, {0, 1, 1, 1, 2, 2, 0, 0, 0},
{0, 1, 1, 2, 0, 0, 1, 2, 0}, {0, 1, 1, 2, 0, 0, 2, 1, 0},
{0, 1, 1, 2, 0, 1, 0, 2, 0}, {0, 1, 1, 2, 0, 1, 2, 0, 0},
{0, 1, 1, 2, 0, 2, 0, 1, 0}, {0, 1, 1, 2, 0, 2, 1, 0, 0},
{0, 1, 1, 2, 0, 2, 2, 2, 0}, {0, 1, 1, 2, 1, 2, 0, 0, 0},
{0, 1, 1, 2, 2, 1, 0, 0, 0}, {0, 1, 2, 0, 0, 0, 1, 1, 2},
{0, 1, 2, 0, 0, 0, 1, 2, 1}, {0, 1, 2, 0, 0, 0, 2, 1, 1},
{0, 1, 2, 0, 0, 1, 0, 1, 2}, {0, 1, 2, 0, 0, 1, 0, 2, 1},
{0, 1, 2, 0, 0, 1, 1, 2, 0}, {0, 1, 2, 0, 0, 1, 2, 1, 0},
{0, 1, 2, 0, 0, 2, 0, 1, 1}, {0, 1, 2, 0, 0, 2, 1, 1, 0},
{0, 1, 2, 0, 1, 0, 0, 2, 1}, {0, 1, 2, 0, 1, 1, 0, 0, 2},
{0, 1, 2, 0, 1, 1, 0, 2, 0}, {0, 1, 2, 0, 1, 2, 0, 0, 1},
{0, 1, 2, 0, 1, 2, 0, 2, 2}, {0, 1, 2, 0, 2, 0, 0, 1, 1},
{0, 1, 2, 0, 2, 1, 0, 0, 1}, {0, 1, 2, 0, 2, 1, 0, 1, 0},
{0, 1, 2, 0, 2, 1, 0, 2, 2}, {0, 1, 2, 0, 2, 2, 0, 1, 2},
{0, 1, 2, 0, 2, 2, 0, 2, 1}, {0, 1, 2, 1, 0, 0, 1, 2, 0},
{0, 1, 2, 1, 0, 0, 2, 1, 0}, {0, 1, 2, 1, 0, 1, 0, 2, 0},
{0, 1, 2, 1, 0, 1, 2, 0, 0}, {0, 1, 2, 1, 0, 2, 0, 1, 0},
{0, 1, 2, 1, 0, 2, 1, 0, 0}, {0, 1, 2, 1, 0, 2, 2, 2, 0},
{0, 1, 2, 1, 1, 2, 0, 0, 0}, {0, 1, 2, 1, 2, 1, 0, 0, 0},
{0, 1, 2, 2, 0, 0, 1, 1, 0}, {0, 1, 2, 2, 0, 1, 0, 1, 0},
{0, 1, 2, 2, 0, 1, 1, 0, 0}, {0, 1, 2, 2, 0, 1, 2, 2, 0},
{0, 1, 2, 2, 0, 2, 1, 2, 0}, {0, 1, 2, 2, 0, 2, 2, 1, 0},
{0, 1, 2, 2, 1, 1, 0, 0, 0}, {0, 2, 0, 0, 1, 1, 0, 1, 2},
{0, 2, 0, 0, 1, 1, 0, 2, 1}, {0, 2, 0, 0, 1, 2, 0, 1, 1},
{0, 2, 0, 0, 2, 1, 0, 1, 1}, {0, 2, 0, 1, 0, 1, 0, 1, 2},
{0, 2, 0, 1, 0, 1, 0, 2, 1}, {0, 2, 0, 1, 0, 1, 1, 2, 0},
{0, 2, 0, 1, 0, 1, 2, 1, 0}, {0, 2, 0, 1, 0, 2, 0, 1, 1},
{0, 2, 0, 1, 0, 2, 1, 1, 0}, {0, 2, 0, 1, 1, 0, 1, 2, 0},
{0, 2, 0, 1, 1, 0, 2, 1, 0}, {0, 2, 0, 1, 2, 0, 1, 1, 0},
{0, 2, 0, 2, 0, 1, 0, 1, 1}, {0, 2, 0, 2, 0, 1, 1, 1, 0},
{0, 2, 0, 2, 1, 0, 1, 1, 0}, {0, 2, 1, 0, 0, 0, 1, 1, 2},
{0, 2, 1, 0, 0, 0, 1, 2, 1}, {0, 2, 1, 0, 0, 0, 2, 1, 1},
{0, 2, 1, 0, 0, 1, 0, 1, 2}, {0, 2, 1, 0, 0, 1, 1, 2, 0},
{0, 2, 1, 0, 0, 1, 2, 1, 0}, {0, 2, 1, 0, 0, 2, 0, 1, 1},
{0, 2, 1, 0, 0, 2, 1, 1, 0}, {0, 2, 1, 0, 1, 0, 0, 1, 2},
{0, 2, 1, 0, 1, 0, 0, 2, 1}, {0, 2, 1, 0, 1, 1, 0, 0, 2},
{0, 2, 1, 0, 1, 1, 0, 2, 0}, {0, 2, 1, 0, 1, 2, 0, 0, 1},
{0, 2, 1, 0, 1, 2, 0, 1, 0}, {0, 2, 1, 0, 1, 2, 0, 2, 2},
{0, 2, 1, 0, 2, 0, 0, 1, 1}, {0, 2, 1, 0, 2, 1, 0, 1, 0},
```

```
{0, 2, 1, 0, 2, 1, 0, 2, 2}, {0, 2, 1, 0, 2, 2, 0, 1, 2},
{0, 2, 1, 0, 2, 2, 0, 2, 1}, {0, 2, 1, 1, 0, 0, 1, 2, 0},
{0, 2, 1, 1, 0, 0, 2, 1, 0}, {0, 2, 1, 1, 0, 1, 0, 2, 0},
{0, 2, 1, 1, 0, 1, 2, 0, 0}, {0, 2, 1, 1, 0, 2, 0, 1, 0},
{0, 2, 1, 1, 0, 2, 1, 0, 0}, {0, 2, 1, 1, 0, 2, 2, 2, 0},
{0, 2, 1, 1, 1, 2, 0, 0, 0}, {0, 2, 1, 1, 2, 1, 0, 0, 0},
{0, 2, 1, 2, 0, 0, 1, 1, 0}, {0, 2, 1, 2, 0, 1, 0, 1, 0},
{0, 2, 1, 2, 0, 1, 1, 0, 0}, {0, 2, 1, 2, 0, 1, 2, 2, 0},
{0, 2, 1, 2, 0, 2, 1, 2, 0}, {0, 2, 1, 2, 0, 2, 2, 1, 0},
{0, 2, 1, 2, 1, 1, 0, 0, 0}, {0, 2, 2, 0, 0, 1, 0, 1, 1},
{0, 2, 2, 0, 0, 1, 1, 1, 0}, {0, 2, 2, 0, 1, 0, 0, 1, 1},
{0, 2, 2, 0, 1, 1, 0, 0, 1}, {0, 2, 2, 0, 1, 1, 0, 1, 0},
{0, 2, 2, 0, 1, 1, 0, 2, 2}, {0, 2, 2, 0, 1, 2, 0, 1, 2},
{0, 2, 2, 0, 1, 2, 0, 2, 1}, {0, 2, 2, 0, 2, 1, 0, 1, 2},
{0, 2, 2, 0, 2, 1, 0, 2, 1}, {0, 2, 2, 0, 2, 2, 0, 1, 1},
{0, 2, 2, 1, 0, 0, 1, 1, 0}, {0, 2, 2, 1, 0, 1, 0, 1, 0},
{0, 2, 2, 1, 0, 1, 1, 0, 0}, {0, 2, 2, 1, 0, 1, 2, 2, 0},
{0, 2, 2, 1, 0, 2, 1, 2, 0}, {0, 2, 2, 1, 0, 2, 2, 1, 0},
{0, 2, 2, 2, 0, 1, 1, 2, 0}, {0, 2, 2, 2, 0, 1, 2, 1, 0},
{0, 2, 2, 2, 0, 2, 1, 1, 0}, {1, 0, 0, 0, 0, 1, 0, 1, 1},
{1, 0, 0, 0, 0, 1, 1, 0, 1}, {1, 0, 0, 0, 1, 0, 1, 1, 0},
{1, 0, 0, 1, 0, 0, 0, 1, 1}, {1, 0, 0, 1, 0, 1, 0, 0, 1},
{1, 0, 0, 1, 0, 1, 0, 1, 0}, {1, 0, 0, 1, 0, 1, 0, 2, 2},
{1, 0, 0, 1, 0, 1, 2, 0, 2}, {1, 0, 0, 1, 0, 2, 0, 1, 2},
{1, 0, 0, 1, 0, 2, 0, 2, 1}, {1, 0, 0, 1, 0, 2, 2, 0, 1},
{1, 0, 0, 1, 1, 0, 0, 1, 0}, {1, 0, 0, 1, 1, 0, 2, 2, 0},
{1, 0, 0, 1, 2, 0, 2, 1, 0}, {1, 0, 0, 2, 0, 1, 0, 1, 2},
{1, 0, 0, 2, 0, 1, 0, 2, 1}, {1, 0, 0, 2, 0, 1, 1, 0, 2},
{1, 0, 0, 2, 0, 1, 2, 0, 1}, {1, 0, 0, 2, 0, 2, 0, 1, 1},
{1, 0, 0, 2, 0, 2, 1, 0, 1}, {1, 0, 0, 2, 1, 0, 1, 2, 0},
{1, 0, 0, 2, 1, 0, 2, 1, 0}, {1, 0, 0, 2, 2, 0, 1, 1, 0},
{1, 0, 1, 0, 0, 0, 0, 1, 1}, {1, 0, 1, 0, 0, 0, 1, 0, 1},
{1, 0, 1, 0, 0, 0, 1, 1, 0}, {1, 0, 1, 0, 0, 0, 1, 2, 2},
{1, 0, 1, 0, 0, 0, 2, 1, 2}, {1, 0, 1, 0, 0, 0, 2, 2, 1},
{1, 0, 1, 0, 0, 1, 1, 0, 0}, {1, 0, 1, 0, 0, 1, 2, 0, 2},
{1, 0, 1, 0, 0, 2, 1, 0, 2}, {1, 0, 1, 0, 0, 2, 2, 0, 1},
{1, 0, 1, 0, 1, 1, 0, 0, 0}, {1, 0, 1, 1, 0, 0, 0, 0, 1},
{1, 0, 1, 1, 0, 0, 2, 0, 2}, {1, 0, 1, 1, 0, 1, 0, 0, 0},
{1, 0, 1, 1, 0, 2, 0, 0, 2}, {1, 0, 1, 1, 0, 2, 2, 0, 0},
{1, 0, 1, 1, 1, 0, 0, 0, 0}, {1, 0, 1, 1, 2, 2, 0, 0, 0},
{1, 0, 1, 2, 0, 0, 1, 0, 2}, {1, 0, 1, 2, 0, 0, 2, 0, 1},
{1, 0, 1, 2, 0, 1, 0, 0, 2}, {1, 0, 1, 2, 0, 1, 2, 0, 0},
{1, 0, 1, 2, 0, 2, 0, 0, 1}, {1, 0, 1, 2, 0, 2, 1, 0, 0},
{1, 0, 1, 2, 0, 2, 2, 0, 2}, {1, 0, 1, 2, 1, 2, 0, 0, 0},
{1, 0, 1, 2, 2, 1, 0, 0, 0}, {1, 0, 2, 0, 0, 0, 1, 1, 2},
{1, 0, 2, 0, 0, 0, 1, 2, 1}, {1, 0, 2, 0, 0, 0, 2, 1, 1},
{1, 0, 2, 0, 0, 1, 1, 0, 2}, {1, 0, 2, 0, 0, 1, 2, 0, 1},
{1, 0, 2, 0, 0, 2, 1, 0, 1}, {1, 0, 2, 1, 0, 0, 2, 0, 1},
{1, 0, 2, 1, 0, 1, 0, 0, 2}, {1, 0, 2, 1, 0, 1, 2, 0, 0},
{1, 0, 2, 1, 0, 2, 0, 0, 1}, {1, 0, 2, 1, 0, 2, 2, 0, 2},
{1, 0, 2, 1, 1, 2, 0, 0, 0}, {1, 0, 2, 1, 2, 1, 0, 0, 0},
{1, 0, 2, 2, 0, 0, 1, 0, 1}, {1, 0, 2, 2, 0, 1, 0, 0, 1},
{1, 0, 2, 2, 0, 1, 1, 0, 0}, {1, 0, 2, 2, 0, 1, 2, 0, 2},
{1, 0, 2, 2, 0, 2, 1, 0, 2}, {1, 0, 2, 2, 0, 2, 2, 0, 1},
{1, 0, 2, 2, 1, 1, 0, 0, 0}, {1, 1, 0, 0, 0, 0, 0, 1, 1},
{1, 1, 0, 0, 0, 0, 1, 0, 1}, {1, 1, 0, 0, 0, 0, 1, 1, 0},
```

```
{1, 1, 0, 0, 0, 0, 1, 2, 2}, {1, 1, 0, 0, 0, 0, 2, 1, 2},
{1, 1, 0, 0, 0, 0, 2, 2, 1}, {1, 1, 0, 0, 0, 1, 0, 0, 1},
{1, 1, 0, 0, 0, 1, 0, 1, 0}, {1, 1, 0, 0, 0, 1, 0, 2, 2},
{1, 1, 0, 0, 0, 2, 0, 1, 2}, {1, 1, 0, 0, 0, 2, 0, 2, 1},
{1, 1, 0, 0, 1, 0, 1, 0, 0}, {1, 1, 0, 0, 1, 0, 2, 2, 0},
{1, 1, 0, 0, 1, 1, 0, 0, 0}, {1, 1, 0, 0, 2, 0, 1, 2, 0},
{1, 1, 0, 0, 2, 0, 2, 1, 0}, {1, 1, 0, 1, 0, 0, 0, 0, 1},
{1, 1, 0, 1, 0, 0, 0, 1, 0}, {1, 1, 0, 1, 0, 0, 0, 2, 2},
{1, 1, 0, 1, 0, 0, 2, 2, 0}, {1, 1, 0, 1, 0, 1, 0, 0, 0},
{1, 1, 0, 1, 0, 2, 0, 0, 2}, {1, 1, 0, 1, 0, 2, 0, 2, 0},
{1, 1, 0, 1, 1, 0, 0, 0, 0}, {1, 1, 0, 1, 2, 0, 0, 2, 0},
{1, 1, 0, 1, 2, 0, 2, 0, 0}, {1, 1, 0, 1, 2, 2, 0, 0, 0},
{1, 1, 0, 2, 0, 0, 0, 1, 2}, {1, 1, 0, 2, 0, 0, 0, 2, 1},
{1, 1, 0, 2, 0, 0, 1, 2, 0}, {1, 1, 0, 2, 0, 0, 2, 1, 0},
{1, 1, 0, 2, 0, 1, 0, 0, 2}, {1, 1, 0, 2, 0, 1, 0, 2, 0},
{1, 1, 0, 2, 0, 2, 0, 0, 1}, {1, 1, 0, 2, 0, 2, 0, 1, 0},
{1, 1, 0, 2, 0, 2, 0, 2, 2}, {1, 1, 0, 2, 1, 0, 0, 2, 0},
{1, 1, 0, 2, 1, 0, 2, 0, 0}, {1, 1, 0, 2, 1, 2, 0, 0, 0},
{1, 1, 0, 2, 2, 0, 0, 1, 0}, {1, 1, 0, 2, 2, 0, 1, 0, 0},
{1, 1, 0, 2, 2, 0, 2, 2, 0}, {1, 1, 0, 2, 2, 1, 0, 0, 0},
{1, 1, 2, 0, 0, 0, 0, 1, 2}, {1, 1, 2, 0, 0, 0, 0, 2, 1},
{1, 1, 2, 0, 0, 0, 1, 0, 2}, {1, 1, 2, 0, 0, 0, 1, 2, 0},
{1, 1, 2, 0, 0, 0, 2, 0, 1}, {1, 1, 2, 0, 0, 0, 2, 1, 0},
{1, 1, 2, 0, 0, 0, 2, 2, 2}, {1, 1, 2, 0, 1, 2, 0, 0, 0},
{1, 1, 2, 0, 2, 1, 0, 0, 0}, {1, 1, 2, 1, 0, 2, 0, 0, 0},
{1, 1, 2, 1, 2, 0, 0, 0, 0}, {1, 1, 2, 2, 0, 1, 0, 0, 0},
{1, 1, 2, 2, 1, 0, 0, 0, 0}, {1, 1, 2, 2, 2, 2, 0, 0, 0},
{1, 2, 0, 0, 0, 0, 1, 1, 2}, {1, 2, 0, 0, 0, 0, 1, 2, 1},
{1, 2, 0, 0, 0, 0, 2, 1, 1}, {1, 2, 0, 0, 0, 1, 0, 1, 2},
{1, 2, 0, 0, 0, 1, 0, 2, 1}, {1, 2, 0, 0, 0, 2, 0, 1, 1},
{1, 2, 0, 0, 1, 0, 1, 2, 0}, {1, 2, 0, 0, 1, 0, 2, 1, 0},
{1, 2, 0, 0, 2, 0, 1, 1, 0}, {1, 2, 0, 1, 0, 0, 0, 1, 2},
{1, 2, 0, 1, 0, 0, 0, 2, 1}, {1, 2, 0, 1, 0, 0, 2, 1, 0},
{1, 2, 0, 1, 0, 1, 0, 0, 2}, {1, 2, 0, 1, 0, 1, 0, 2, 0},
{1, 2, 0, 1, 0, 2, 0, 0, 1}, {1, 2, 0, 1, 0, 2, 0, 1, 0},
{1, 2, 0, 1, 0, 2, 0, 2, 2}, {1, 2, 0, 1, 1, 0, 0, 2, 0},
{1, 2, 0, 1, 1, 0, 2, 0, 0}, {1, 2, 0, 1, 1, 2, 0, 0, 0},
{1, 2, 0, 1, 2, 0, 0, 1, 0}, {1, 2, 0, 1, 2, 0, 2, 2, 0},
{1, 2, 0, 1, 2, 1, 0, 0, 0}, {1, 2, 0, 2, 0, 0, 0, 1, 1},
{1, 2, 0, 2, 0, 0, 1, 1, 0}, {1, 2, 0, 2, 0, 1, 0, 0, 1},
{1, 2, 0, 2, 0, 1, 0, 1, 0}, {1, 2, 0, 2, 0, 1, 0, 2, 2},
{1, 2, 0, 2, 0, 2, 0, 1, 2}, {1, 2, 0, 2, 0, 2, 0, 2, 1},
{1, 2, 0, 2, 1, 0, 0, 1, 0}, {1, 2, 0, 2, 1, 0, 1, 0, 0},
{1, 2, 0, 2, 1, 0, 2, 2, 0}, {1, 2, 0, 2, 1, 1, 0, 0, 0},
{1, 2, 0, 2, 2, 0, 1, 2, 0}, {1, 2, 0, 2, 2, 0, 2, 1, 0},
{1, 2, 1, 0, 0, 0, 0, 1, 2}, {1, 2, 1, 0, 0, 0, 0, 2, 1},
{1, 2, 1, 0, 0, 0, 1, 0, 2}, {1, 2, 1, 0, 0, 0, 1, 2, 0},
{1, 2, 1, 0, 0, 0, 2, 0, 1}, {1, 2, 1, 0, 0, 0, 2, 1, 0},
{1, 2, 1, 0, 0, 0, 2, 2, 2}, {1, 2, 1, 0, 1, 2, 0, 0, 0},
{1, 2, 1, 0, 2, 1, 0, 0, 0}, {1, 2, 1, 1, 0, 2, 0, 0, 0},
{1, 2, 1, 1, 2, 0, 0, 0, 0}, {1, 2, 1, 2, 0, 1, 0, 0, 0},
{1, 2, 1, 2, 1, 0, 0, 0, 0}, {1, 2, 1, 2, 2, 2, 0, 0, 0},
{1, 2, 2, 0, 0, 0, 0, 1, 1}, {1, 2, 2, 0, 0, 0, 1, 0, 1},
{1, 2, 2, 0, 0, 0, 1, 1, 0}, {1, 2, 2, 0, 0, 0, 1, 2, 2},
{1, 2, 2, 0, 0, 0, 2, 1, 2}, {1, 2, 2, 0, 0, 0, 2, 2, 1},
{1, 2, 2, 0, 1, 1, 0, 0, 0}, {1, 2, 2, 1, 0, 1, 0, 0, 0},
```

```
{1, 2, 2, 1, 1, 0, 0, 0, 0}, {1, 2, 2, 1, 2, 2, 0, 0, 0},
{1, 2, 2, 2, 1, 2, 0, 0, 0}, {1, 2, 2, 2, 2, 1, 0, 0, 0},
{2, 0, 0, 1, 0, 1, 0, 1, 2}, {2, 0, 0, 1, 0, 1, 0, 2, 1},
{2, 0, 0, 1, 0, 1, 1, 0, 2}, {2, 0, 0, 1, 0, 1, 2, 0, 1},
{2, 0, 0, 1, 0, 2, 0, 1, 1}, {2, 0, 0, 1, 0, 2, 1, 0, 1},
{2, 0, 0, 1, 1, 0, 1, 2, 0}, {2, 0, 0, 1, 1, 0, 2, 1, 0},
{2, 0, 0, 1, 2, 0, 1, 1, 0}, {2, 0, 0, 2, 0, 1, 0, 1, 1},
{2, 0, 0, 2, 0, 1, 1, 0, 1}, {2, 0, 0, 2, 1, 0, 1, 1, 0},
{2, 0, 1, 0, 0, 0, 1, 1, 2}, {2, 0, 1, 0, 0, 0, 1, 2, 1},
{2, 0, 1, 0, 0, 0, 2, 1, 1}, {2, 0, 1, 0, 0, 1, 1, 0, 2},
{2, 0, 1, 0, 0, 2, 1, 0, 1}, {2, 0, 1, 1, 0, 0, 1, 0, 2},
{2, 0, 1, 1, 0, 0, 2, 0, 1}, {2, 0, 1, 1, 0, 1, 0, 0, 2},
{2, 0, 1, 1, 0, 1, 2, 0, 0}, {2, 0, 1, 1, 0, 2, 0, 0, 1},
{2, 0, 1, 1, 0, 2, 1, 0, 0}, {2, 0, 1, 1, 0, 2, 2, 0, 2},
{2, 0, 1, 1, 1, 2, 0, 0, 0}, {2, 0, 1, 1, 2, 1, 0, 0, 0},
{2, 0, 1, 2, 0, 0, 1, 0, 1}, {2, 0, 1, 2, 0, 1, 1, 0, 0},
{2, 0, 1, 2, 0, 1, 2, 0, 2}, {2, 0, 1, 2, 0, 2, 1, 0, 2},
{2, 0, 1, 2, 0, 2, 2, 0, 1}, {2, 0, 1, 2, 1, 1, 0, 0, 0},
{2, 0, 2, 0, 0, 1, 1, 0, 1}, {2, 0, 2, 1, 0, 0, 1, 0, 1},
{2, 0, 2, 1, 0, 1, 0, 0, 1}, {2, 0, 2, 1, 0, 1, 1, 0, 0},
{2, 0, 2, 1, 0, 1, 2, 0, 2}, {2, 0, 2, 1, 0, 2, 1, 0, 2},
{2, 0, 2, 1, 0, 2, 2, 0, 1}, {2, 0, 2, 2, 0, 1, 1, 0, 2},
{2, 0, 2, 2, 0, 1, 2, 0, 1}, {2, 0, 2, 2, 0, 2, 1, 0, 1},
{2, 1, 0, 0, 0, 0, 1, 1, 2}, {2, 1, 0, 0, 0, 0, 1, 2, 1},
{2, 1, 0, 0, 0, 0, 2, 1, 1}, {2, 1, 0, 0, 0, 1, 0, 1, 2},
{2, 1, 0, 0, 0, 1, 0, 2, 1}, {2, 1, 0, 0, 0, 2, 0, 1, 1},
{2, 1, 0, 0, 1, 0, 1, 2, 0}, {2, 1, 0, 0, 2, 0, 1, 1, 0},
{2, 1, 0, 1, 0, 0, 0, 1, 2}, {2, 1, 0, 1, 0, 0, 0, 2, 1},
{2, 1, 0, 1, 0, 0, 1, 2, 0}, {2, 1, 0, 1, 0, 0, 2, 1, 0},
{2, 1, 0, 1, 0, 1, 0, 0, 2}, {2, 1, 0, 1, 0, 1, 0, 2, 0},
{2, 1, 0, 1, 0, 2, 0, 0, 1}, {2, 1, 0, 1, 0, 2, 0, 1, 0},
{2, 1, 0, 1, 0, 2, 0, 2, 2}, {2, 1, 0, 1, 1, 0, 0, 2, 0},
{2, 1, 0, 1, 1, 0, 2, 0, 0}, {2, 1, 0, 1, 1, 2, 0, 0, 0},
{2, 1, 0, 1, 2, 0, 0, 1, 0}, {2, 1, 0, 1, 2, 0, 1, 0, 0},
{2, 1, 0, 1, 2, 0, 2, 2, 0}, {2, 1, 0, 1, 2, 1, 0, 0, 0},
{2, 1, 0, 2, 0, 0, 0, 1, 1}, {2, 1, 0, 2, 0, 0, 1, 1, 0},
{2, 1, 0, 2, 0, 1, 0, 0, 1}, {2, 1, 0, 2, 0, 1, 0, 1, 0},
{2, 1, 0, 2, 0, 1, 0, 2, 2}, {2, 1, 0, 2, 0, 2, 0, 1, 2},
{2, 1, 0, 2, 0, 2, 0, 2, 1}, {2, 1, 0, 2, 1, 0, 1, 0, 0},
{2, 1, 0, 2, 1, 0, 2, 2, 0}, {2, 1, 0, 2, 1, 1, 0, 0, 0},
{2, 1, 0, 2, 2, 0, 1, 2, 0}, {2, 1, 0, 2, 2, 0, 2, 1, 0},
{2, 1, 1, 0, 0, 0, 0, 1, 2}, {2, 1, 1, 0, 0, 0, 0, 2, 1},
{2, 1, 1, 0, 0, 0, 1, 0, 2}, {2, 1, 1, 0, 0, 0, 1, 2, 0},
{2, 1, 1, 0, 0, 0, 2, 0, 1}, {2, 1, 1, 0, 0, 0, 2, 1, 0},
{2, 1, 1, 0, 0, 0, 2, 2, 2}, {2, 1, 1, 0, 1, 2, 0, 0, 0},
{2, 1, 1, 0, 2, 1, 0, 0, 0}, {2, 1, 1, 1, 0, 2, 0, 0, 0},
{2, 1, 1, 1, 2, 0, 0, 0, 0}, {2, 1, 1, 2, 0, 1, 0, 0, 0},
{2, 1, 1, 2, 1, 0, 0, 0, 0}, {2, 1, 1, 2, 2, 2, 0, 0, 0},
{2, 1, 2, 0, 0, 0, 0, 1, 1}, {2, 1, 2, 0, 0, 0, 1, 0, 1},
{2, 1, 2, 0, 0, 0, 1, 1, 0}, {2, 1, 2, 0, 0, 0, 1, 2, 2},
{2, 1, 2, 0, 0, 0, 2, 1, 2}, {2, 1, 2, 0, 0, 0, 2, 2, 1},
{2, 1, 2, 0, 1, 1, 0, 0, 0}, {2, 1, 2, 1, 0, 1, 0, 0, 0},
{2, 1, 2, 1, 1, 0, 0, 0, 0}, {2, 1, 2, 1, 2, 2, 0, 0, 0},
{2, 1, 2, 2, 1, 2, 0, 0, 0}, {2, 1, 2, 2, 2, 1, 0, 0, 0},
{2, 2, 0, 0, 0, 1, 0, 1, 1}, {2, 2, 0, 0, 1, 0, 1, 1, 0},
{2, 2, 0, 1, 0, 0, 0, 1, 1}, {2, 2, 0, 1, 0, 0, 1, 1, 0},
```

```
{2, 2, 0, 1, 0, 1, 0, 0, 1}, {2, 2, 0, 1, 0, 1, 0, 1, 0},
{2, 2, 0, 1, 0, 1, 0, 2, 2}, {2, 2, 0, 1, 0, 2, 0, 1, 2},
{2, 2, 0, 1, 0, 2, 0, 2, 1}, {2, 2, 0, 1, 1, 0, 0, 1, 0},
{2, 2, 0, 1, 1, 0, 1, 0, 0}, {2, 2, 0, 1, 1, 0, 2, 2, 0},
{2, 2, 0, 1, 2, 0, 1, 2, 0}, {2, 2, 0, 1, 2, 0, 2, 1, 0},
{2, 2, 0, 2, 0, 1, 0, 1, 2}, {2, 2, 0, 2, 0, 1, 0, 2, 1},
{2, 2, 0, 2, 0, 2, 0, 1, 1}, {2, 2, 0, 2, 1, 0, 1, 2, 0},
{2, 2, 0, 2, 1, 0, 2, 1, 0}, {2, 2, 0, 2, 2, 0, 1, 1, 0},
{2, 2, 1, 0, 0, 0, 0, 1, 1}, {2, 2, 1, 0, 0, 0, 1, 0, 1},
{2, 2, 1, 0, 0, 0, 1, 1, 0}, {2, 2, 1, 0, 0, 0, 1, 2, 2},
{2, 2, 1, 0, 0, 0, 2, 1, 2}, {2, 2, 1, 0, 0, 0, 2, 2, 1},
{2, 2, 1, 0, 1, 1, 0, 0, 0}, {2, 2, 1, 1, 0, 1, 0, 0, 0},
{2, 2, 1, 1, 1, 0, 0, 0, 0}, {2, 2, 1, 1, 2, 2, 0, 0, 0},
{2, 2, 1, 2, 1, 2, 0, 0, 0}, {2, 2, 1, 2, 2, 1, 0, 0, 0},
{2, 2, 2, 0, 0, 0, 1, 1, 2}, {2, 2, 2, 0, 0, 0, 1, 2, 1},
{2, 2, 2, 0, 0, 0, 2, 1, 1}, {2, 2, 2, 1, 1, 2, 0, 0, 0},
{2, 2, 2, 1, 2, 1, 0, 0, 0}, {2, 2, 2, 2, 1, 1, 0, 0, 0},
{0, 0, 1, 0, 0, 1, 1, 2, 1}, {0, 0, 1, 0, 0, 1, 2, 1, 1},
{0, 0, 1, 0, 0, 2, 1, 1, 1}, {0, 0, 1, 0, 1, 0, 1, 1, 2},
{0, 0, 1, 0, 1, 0, 1, 2, 1}, {0, 0, 1, 0, 1, 1, 1, 0, 2},
{0, 0, 1, 0, 1, 1, 1, 2, 0}, {0, 0, 1, 0, 1, 1, 2, 0, 1},
{0, 0, 1, 0, 1, 2, 1, 0, 1}, {0, 0, 1, 0, 1, 2, 1, 1, 0},
{0, 0, 1, 0, 1, 2, 1, 2, 2}, {0, 0, 1, 0, 2, 0, 1, 1, 1},
{0, 0, 1, 0, 2, 1, 1, 0, 1}, {0, 0, 1, 0, 2, 1, 2, 2, 1},
{0, 0, 1, 1, 0, 1, 0, 2, 1}, {0, 0, 1, 1, 1, 0, 1, 0, 2},
{0, 0, 1, 1, 1, 0, 1, 2, 0}, {0, 0, 1, 1, 1, 1, 0, 0, 2},
{0, 0, 1, 1, 1, 1, 0, 2, 0}, {0, 0, 1, 1, 1, 1, 2, 0, 0},
{0, 0, 1, 1, 1, 2, 1, 0, 0}, {0, 0, 1, 1, 2, 1, 0, 0, 1},
{0, 0, 1, 2, 0, 0, 1, 1, 1}, {0, 0, 1, 2, 0, 1, 0, 1, 1},
{0, 0, 1, 2, 0, 1, 2, 2, 1}, {0, 0, 1, 2, 1, 0, 1, 0, 1},
{0, 0, 1, 2, 1, 0, 1, 1, 0}, {0, 0, 1, 2, 1, 0, 1, 2, 2},
{0, 0, 1, 2, 1, 1, 0, 0, 1}, {0, 0, 1, 2, 1, 1, 1, 0, 0},
{0, 0, 1, 2, 1, 2, 1, 0, 2}, {0, 0, 1, 2, 1, 2, 1, 2, 0},
{0, 0, 1, 2, 2, 1, 0, 2, 1}, {0, 0, 1, 2, 2, 1, 2, 0, 1},
{0, 0, 2, 0, 0, 1, 1, 1, 1}, {0, 0, 2, 0, 1, 0, 1, 1, 1},
{0, 0, 2, 0, 2, 2, 1, 1, 1}, {0, 0, 2, 1, 0, 0, 1, 1, 1},
{0, 0, 2, 1, 1, 1, 0, 0, 1}, {0, 0, 2, 1, 1, 1, 0, 1, 0},
{0, 0, 2, 1, 1, 1, 0, 2, 2}, {0, 0, 2, 1, 1, 1, 1, 0, 0},
{0, 0, 2, 1, 1, 1, 2, 0, 2}, {0, 0, 2, 1, 1, 1, 2, 2, 0},
{0, 0, 2, 2, 0, 2, 1, 1, 1}, {0, 0, 2, 2, 2, 0, 1, 1, 1},
{0, 1, 0, 0, 0, 2, 1, 1, 1}, {0, 1, 0, 0, 1, 0, 1, 1, 2},
{0, 1, 0, 0, 1, 0, 2, 1, 1}, {0, 1, 0, 0, 1, 1, 2, 1, 0},
{0, 1, 0, 0, 1, 2, 1, 1, 0}, {0, 1, 0, 0, 1, 2, 2, 1, 2},
{0, 1, 0, 0, 2, 0, 1, 1, 1}, {0, 1, 0, 1, 1, 0, 0, 1, 2},
{0, 1, 0, 1, 1, 1, 0, 0, 2}, {0, 1, 0, 1, 1, 1, 0, 2, 0},
{0, 1, 0, 1, 1, 1, 2, 0, 0}, {0, 1, 0, 1, 1, 2, 0, 1, 0},
{0, 1, 0, 2, 0, 0, 1, 1, 1}, {0, 1, 0, 2, 1, 0, 0, 1, 1},
{0, 1, 0, 2, 1, 0, 2, 1, 2}, {0, 1, 0, 2, 1, 1, 0, 1, 0},
{0, 1, 0, 2, 1, 2, 0, 1, 2}, {0, 1, 0, 2, 1, 2, 2, 1, 0},
{0, 1, 1, 0, 0, 1, 2, 0, 1}, {0, 1, 1, 0, 1, 0, 1, 0, 2},
{0, 1, 1, 0, 1, 0, 1, 2, 0}, {0, 1, 1, 0, 1, 0, 2, 1, 0},
{0, 1, 1, 0, 1, 2, 1, 0, 0}, {0, 1, 1, 2, 0, 1, 0, 0, 1},
{0, 1, 1, 2, 1, 0, 0, 1, 0}, {0, 1, 1, 2, 1, 0, 1, 0, 0},
{0, 1, 2, 0, 1, 0, 1, 1, 0}, {0, 1, 2, 0, 1, 0, 2, 1, 2},
{0, 1, 2, 0, 1, 2, 2, 1, 0}, {0, 1, 2, 1, 1, 0, 0, 1, 0},
{0, 1, 2, 2, 1, 0, 0, 1, 2}, {0, 1, 2, 2, 1, 0, 2, 1, 0},
```

```
{0, 1, 2, 2, 1, 2, 0, 1, 0}, {0, 2, 0, 0, 0, 1, 1, 1, 1},
{0, 2, 0, 0, 1, 0, 1, 1, 1}, {0, 2, 0, 0, 2, 2, 1, 1, 1},
{0, 2, 0, 1, 0, 0, 1, 1, 1}, {0, 2, 0, 1, 1, 1, 0, 0, 1},
{0, 2, 0, 1, 1, 1, 0, 1, 0}, {0, 2, 0, 1, 1, 1, 0, 2, 2},
{0, 2, 0, 1, 1, 1, 1, 0, 0}, {0, 2, 0, 1, 1, 1, 2, 0, 2},
{0, 2, 0, 1, 1, 1, 2, 2, 0}, {0, 2, 0, 2, 0, 2, 1, 1, 1},
{0, 2, 0, 2, 2, 0, 1, 1, 1}, {0, 2, 1, 0, 0, 1, 1, 0, 1},
{0, 2, 1, 0, 0, 1, 2, 2, 1}, {0, 2, 1, 0, 1, 0, 1, 0, 1},
{0, 2, 1, 0, 1, 0, 1, 1, 0}, {0, 2, 1, 0, 1, 0, 1, 2, 2},
{0, 2, 1, 0, 1, 1, 1, 0, 0}, {0, 2, 1, 0, 1, 2, 1, 0, 2},
{0, 2, 1, 0, 1, 2, 1, 2, 0}, {0, 2, 1, 0, 2, 1, 2, 0, 1},
{0, 2, 1, 1, 0, 1, 0, 0, 1}, {0, 2, 1, 1, 1, 0, 1, 0, 0},
{0, 2, 1, 2, 0, 1, 0, 2, 1}, {0, 2, 1, 2, 0, 1, 2, 0, 1},
{0, 2, 1, 2, 1, 0, 1, 0, 2}, {0, 2, 1, 2, 1, 0, 1, 2, 0},
{0, 2, 1, 2, 1, 2, 1, 0, 0}, {0, 2, 1, 2, 2, 1, 0, 0, 1},
{0, 2, 2, 0, 0, 2, 1, 1, 1}, {0, 2, 2, 0, 2, 0, 1, 1, 1},
{0, 2, 2, 1, 1, 1, 0, 0, 2}, {0, 2, 2, 1, 1, 1, 0, 2, 0},
{0, 2, 2, 1, 1, 1, 2, 0, 0}, {0, 2, 2, 2, 0, 0, 1, 1, 1},
{1, 0, 0, 0, 0, 2, 1, 1, 1}, {1, 0, 0, 0, 1, 0, 1, 2, 1},
{1, 0, 0, 0, 1, 0, 2, 1, 1}, {1, 0, 0, 0, 1, 1, 0, 2, 1},
{1, 0, 0, 0, 1, 1, 2, 0, 1}, {1, 0, 0, 0, 1, 2, 0, 1, 1},
{1, 0, 0, 0, 1, 2, 1, 0, 1}, {1, 0, 0, 0, 1, 2, 2, 2, 1},
{1, 0, 0, 0, 2, 0, 1, 1, 1}, {1, 0, 0, 1, 0, 0, 1, 1, 2},
{1, 0, 0, 1, 0, 0, 1, 2, 1}, {1, 0, 0, 1, 0, 1, 1, 2, 0},
{1, 0, 0, 1, 0, 2, 1, 1, 0}, {1, 0, 0, 1, 0, 2, 1, 2, 2},
{1, 0, 0, 1, 1, 0, 0, 2, 1}, {1, 0, 0, 1, 1, 0, 1, 0, 2},
{1, 0, 0, 1, 1, 0, 2, 0, 1}, {1, 0, 0, 1, 1, 1, 0, 0, 2},
{1, 0, 0, 1, 1, 1, 0, 2, 0}, {1, 0, 0, 1, 1, 1, 2, 0, 0},
{1, 0, 0, 1, 1, 2, 0, 0, 1}, {1, 0, 0, 1, 1, 2, 1, 0, 0},
{1, 0, 0, 1, 2, 0, 1, 0, 1}, {1, 0, 0, 1, 2, 0, 1, 2, 2},
{1, 0, 0, 1, 2, 1, 1, 0, 0}, {1, 0, 0, 1, 2, 2, 1, 0, 2},
{1, 0, 0, 1, 2, 2, 1, 2, 0}, {1, 0, 0, 2, 0, 0, 1, 1, 1},
{1, 0, 0, 2, 1, 0, 0, 1, 1}, {1, 0, 0, 2, 1, 0, 1, 0, 1},
{1, 0, 0, 2, 1, 0, 2, 2, 1}, {1, 0, 0, 2, 1, 1, 0, 0, 1},
{1, 0, 0, 2, 1, 2, 0, 2, 1}, {1, 0, 0, 2, 1, 2, 2, 0, 1},
{1, 0, 1, 0, 0, 1, 0, 2, 1}, {1, 0, 1, 0, 1, 0, 0, 2, 1},
{1, 0, 1, 0, 1, 0, 1, 0, 2}, {1, 0, 1, 0, 1, 0, 1, 2, 0},
{1, 0, 1, 0, 1, 0, 2, 0, 1}, {1, 0, 1, 0, 1, 2, 0, 0, 1},
{1, 0, 1, 0, 1, 2, 1, 0, 0}, {1, 0, 1, 0, 2, 1, 0, 0, 1},
{1, 0, 1, 1, 0, 0, 1, 2, 0}, {1, 0, 1, 1, 2, 0, 1, 0, 0},
{1, 0, 1, 2, 1, 0, 0, 0, 1}, {1, 0, 1, 2, 1, 0, 1, 0, 0},
{1, 0, 2, 0, 1, 0, 0, 1, 1}, {1, 0, 2, 0, 1, 0, 1, 0, 1},
{1, 0, 2, 0, 1, 0, 2, 2, 1}, {1, 0, 2, 0, 1, 1, 0, 0, 1},
{1, 0, 2, 0, 1, 2, 0, 2, 1}, {1, 0, 2, 0, 1, 2, 2, 0, 1},
{1, 0, 2, 1, 0, 0, 1, 1, 0}, {1, 0, 2, 1, 0, 0, 1, 2, 2},
{1, 0, 2, 1, 0, 2, 1, 2, 0}, {1, 0, 2, 1, 1, 0, 0, 0, 1},
{1, 0, 2, 1, 1, 0, 1, 0, 0}, {1, 0, 2, 1, 2, 0, 1, 0, 2},
{1, 0, 2, 1, 2, 0, 1, 2, 0}, {1, 0, 2, 1, 2, 2, 1, 0, 0},
{1, 0, 2, 2, 1, 0, 0, 2, 1}, {1, 0, 2, 2, 1, 0, 2, 0, 1},
{1, 0, 2, 2, 1, 2, 0, 0, 1}, {1, 1, 0, 0, 1, 0, 0, 1, 2},
{1, 1, 0, 0, 1, 0, 0, 2, 1}, {1, 1, 0, 0, 1, 0, 2, 0, 1},
{1, 1, 0, 0, 1, 2, 0, 0, 1}, {1, 1, 0, 0, 1, 2, 0, 1, 0},
{1, 1, 0, 1, 0, 0, 1, 0, 2}, {1, 1, 0, 1, 0, 2, 1, 0, 0},
{1, 1, 0, 2, 1, 0, 0, 0, 1}, {1, 1, 1, 0, 0, 1, 0, 0, 2},
{1, 1, 1, 0, 0, 1, 0, 2, 0}, {1, 1, 1, 0, 0, 1, 2, 0, 0},
{1, 1, 1, 0, 0, 2, 0, 0, 1}, {1, 1, 1, 0, 0, 2, 0, 1, 0},
```

```
{1, 1, 1, 0, 0, 2, 0, 2, 2}, {1, 1, 1, 0, 0, 2, 1, 0, 0},
{1, 1, 1, 0, 0, 2, 2, 0, 2}, {1, 1, 1, 0, 0, 2, 2, 2, 0},
{1, 1, 1, 0, 1, 0, 0, 0, 2}, {1, 1, 1, 0, 1, 0, 0, 2, 0},
{1, 1, 1, 0, 1, 0, 2, 0, 0}, {1, 1, 1, 0, 2, 0, 0, 0, 1},
{1, 1, 1, 0, 2, 0, 0, 1, 0}, {1, 1, 1, 0, 2, 0, 0, 2, 2},
{1, 1, 1, 0, 2, 0, 1, 0, 0}, {1, 1, 1, 0, 2, 0, 2, 0, 2},
{1, 1, 1, 0, 2, 0, 2, 2, 0}, {1, 1, 1, 0, 2, 2, 0, 0, 2},
{1, 1, 1, 0, 2, 2, 0, 2, 0}, {1, 1, 1, 0, 2, 2, 2, 0, 0},
{1, 1, 1, 1, 0, 0, 0, 0, 2}, {1, 1, 1, 1, 0, 0, 0, 2, 0},
{1, 1, 1, 1, 0, 0, 2, 0, 0}, {1, 1, 1, 2, 0, 0, 0, 0, 1},
{1, 1, 1, 2, 0, 0, 0, 1, 0}, {1, 1, 1, 2, 0, 0, 0, 2, 2},
{1, 1, 1, 2, 0, 0, 1, 0, 0}, {1, 1, 1, 2, 0, 0, 2, 0, 2},
{1, 1, 1, 2, 0, 0, 2, 2, 0}, {1, 1, 1, 2, 0, 2, 0, 0, 2},
{1, 1, 1, 2, 0, 2, 0, 2, 0}, {1, 1, 1, 2, 0, 2, 2, 0, 0},
{1, 1, 1, 2, 2, 0, 0, 0, 2}, {1, 1, 1, 2, 2, 0, 0, 2, 0},
{1, 1, 1, 2, 2, 0, 2, 0, 0}, {1, 1, 2, 0, 1, 0, 0, 0, 1},
{1, 1, 2, 0, 1, 0, 0, 1, 0}, {1, 1, 2, 1, 0, 0, 1, 0, 0},
{1, 2, 0, 0, 1, 0, 0, 1, 1}, {1, 2, 0, 0, 1, 0, 1, 0, 1},
{1, 2, 0, 0, 1, 0, 2, 2, 1}, {1, 2, 0, 0, 1, 1, 0, 0, 1},
{1, 2, 0, 0, 1, 2, 0, 2, 1}, {1, 2, 0, 0, 1, 2, 2, 0, 1},
{1, 2, 0, 1, 0, 0, 1, 0, 1}, {1, 2, 0, 1, 0, 0, 1, 2, 2},
{1, 2, 0, 1, 0, 1, 1, 0, 0}, {1, 2, 0, 1, 0, 2, 1, 0, 2},
{1, 2, 0, 1, 0, 2, 1, 2, 0}, {1, 2, 0, 1, 1, 0, 0, 0, 1},
{1, 2, 0, 1, 2, 0, 1, 0, 2}, {1, 2, 0, 1, 2, 2, 1, 0, 0},
{1, 2, 0, 2, 1, 0, 0, 2, 1}, {1, 2, 0, 2, 1, 0, 2, 0, 1},
{1, 2, 0, 2, 1, 2, 0, 0, 1}, {1, 2, 1, 0, 0, 1, 0, 0, 1},
{1, 2, 1, 0, 1, 0, 0, 0, 1}, {1, 2, 1, 0, 1, 0, 1, 0, 0},
{1, 2, 1, 1, 0, 0, 1, 0, 0}, {1, 2, 2, 0, 1, 0, 0, 2, 1},
{1, 2, 2, 0, 1, 0, 2, 0, 1}, {1, 2, 2, 0, 1, 2, 0, 0, 1},
{1, 2, 2, 1, 0, 0, 1, 0, 2}, {1, 2, 2, 1, 0, 0, 1, 2, 0},
{1, 2, 2, 1, 0, 2, 1, 0, 0}, {1, 2, 2, 1, 2, 0, 1, 0, 0},
{1, 2, 2, 2, 1, 0, 0, 0, 1}, {2, 0, 0, 0, 0, 1, 1, 1, 1},
{2, 0, 0, 0, 1, 0, 1, 1, 1}, {2, 0, 0, 0, 2, 2, 1, 1, 1},
{2, 0, 0, 1, 0, 0, 1, 1, 1}, {2, 0, 0, 1, 1, 1, 0, 0, 1},
{2, 0, 0, 1, 1, 1, 0, 1, 0}, {2, 0, 0, 1, 1, 1, 0, 2, 2},
{2, 0, 0, 1, 1, 1, 1, 0, 0}, {2, 0, 0, 1, 1, 1, 2, 0, 2},
{2, 0, 0, 1, 1, 1, 2, 2, 0}, {2, 0, 0, 2, 0, 2, 1, 1, 1},
{2, 0, 0, 2, 2, 0, 1, 1, 1}, {2, 0, 1, 0, 0, 1, 0, 1, 1},
{2, 0, 1, 0, 0, 1, 2, 2, 1}, {2, 0, 1, 0, 1, 0, 1, 0, 1},
{2, 0, 1, 0, 1, 0, 1, 1, 0}, {2, 0, 1, 0, 1, 0, 1, 2, 2},
{2, 0, 1, 0, 1, 1, 0, 0, 1}, {2, 0, 1, 0, 1, 1, 1, 0, 0},
{2, 0, 1, 0, 1, 2, 1, 0, 2}, {2, 0, 1, 0, 1, 2, 1, 2, 0},
{2, 0, 1, 0, 2, 1, 0, 2, 1}, {2, 0, 1, 0, 2, 1, 2, 0, 1},
{2, 0, 1, 1, 1, 0, 1, 0, 0}, {2, 0, 1, 2, 0, 1, 0, 2, 1},
{2, 0, 1, 2, 1, 0, 1, 0, 2}, {2, 0, 1, 2, 1, 0, 1, 2, 0},
{2, 0, 1, 2, 1, 2, 1, 0, 0}, {2, 0, 1, 2, 2, 1, 0, 0, 1},
{2, 0, 2, 0, 0, 2, 1, 1, 1}, {2, 0, 2, 0, 2, 0, 1, 1, 1},
{2, 0, 2, 1, 1, 1, 0, 0, 2}, {2, 0, 2, 1, 1, 1, 0, 2, 0},
{2, 0, 2, 1, 1, 1, 2, 0, 0}, {2, 0, 2, 2, 0, 0, 1, 1, 1},
{2, 1, 0, 0, 1, 0, 0, 1, 1}, {2, 1, 0, 0, 1, 0, 2, 1, 2},
{2, 1, 0, 0, 1, 1, 0, 1, 0}, {2, 1, 0, 0, 1, 2, 0, 1, 2},
{2, 1, 0, 0, 1, 2, 2, 1, 0}, {2, 1, 0, 2, 1, 0, 0, 1, 2},
{2, 1, 0, 2, 1, 2, 0, 1, 0}, {2, 1, 1, 0, 0, 1, 0, 0, 1},
{2, 1, 1, 0, 1, 0, 0, 1, 0}, {2, 1, 1, 0, 1, 0, 1, 0, 0},
{2, 1, 2, 0, 1, 0, 0, 1, 2}, {2, 1, 2, 0, 1, 0, 2, 1, 0},
{2, 1, 2, 0, 1, 2, 0, 1, 0}, {2, 1, 2, 2, 1, 0, 0, 1, 0},
```

```
        {2, 2, 0, 0, 0, 2, 1, 1, 1}, {2, 2, 0, 0, 2, 0, 1, 1, 1},
        {2, 2, 0, 1, 1, 1, 0, 0, 2}, {2, 2, 0, 1, 1, 1, 0, 2, 0},
        {2, 2, 0, 1, 1, 1, 2, 0, 0}, {2, 2, 0, 2, 0, 0, 1, 1, 1},
        {2, 2, 1, 0, 0, 1, 0, 2, 1}, {2, 2, 1, 0, 0, 1, 2, 0, 1},
        {2, 2, 1, 0, 1, 0, 1, 0, 2}, {2, 2, 1, 0, 1, 0, 1, 2, 0},
        {2, 2, 1, 0, 1, 2, 1, 0, 0}, {2, 2, 1, 0, 2, 1, 0, 0, 1},
        {2, 2, 1, 2, 0, 1, 0, 0, 1}, {2, 2, 1, 2, 1, 0, 1, 0, 0},
        {0, 0, 1, 1, 0, 0, 0, 1, 1}, {0, 0, 1, 1, 1, 0, 0, 0, 1},
        {0, 0, 1, 1, 1, 0, 0, 1, 0}, {0, 1, 0, 0, 0, 1, 1, 0, 1},
        {0, 1, 0, 0, 1, 0, 1, 0, 1}, {0, 1, 0, 0, 1, 1, 1, 0, 0},
        {0, 1, 0, 1, 0, 0, 1, 0, 1}, {0, 1, 0, 1, 1, 0, 0, 0, 1},
        {0, 1, 1, 1, 0, 0, 0, 0, 1}, {1, 0, 0, 0, 0, 1, 1, 1, 0},
        {1, 0, 0, 0, 1, 1, 0, 1, 0}, {1, 0, 0, 0, 1, 1, 1, 0, 0},
        {1, 0, 1, 0, 0, 1, 0, 1, 0}, {1, 0, 1, 0, 1, 0, 0, 1, 0},
        {1, 0, 1, 1, 0, 0, 0, 1, 0}, {1, 1, 0, 0, 0, 1, 1, 0, 0}
    };
    private static double weights[] = {
        -0.00000000000000063401, 0.00000000000000055700, 0.00000000000000012769,
        -0.52573653474162341000, 0.43427498705107342000, 0.09146154769055023200,
        0.00000000000000138130, -0.00000000000000118053, -0.00000000000000050631,
        0.52573653474162607000, -0.43427498705107603000, -0.09146154769055094000,
        -0.00000000000000057743, 0.00000000000000037314, -0.00000000000000023441,
        0.52573653474162907000, -0.43427498705107787000, -0.09146154769055155100,
        -0.00000000000000405476, 0.00000000000000339568, 0.00000000000000053496,
        -0.52573653474162763000, 0.43427498705107587000, 0.09146154769055155100,
        -0.00000000000000116499, 0.00000000000000111960, 0.00000000000000004464,
        0.59181480684449950000, -0.48617039139374285000, -0.10564441545075645000,
        0.33659693927260309000, -0.28023189914604213000, -0.05636504012656110000,
        -0.00000000000000339401, 0.00000000000000312093, 0.00000000000000057542,
        0.33659693927260292000, -0.28023189914604213000, -0.05636504012656087800,
        0.00000000000000099480, -0.00000000000000067295, -0.00000000000000003901,
        -0.33659693927260537000, 0.28023189914604435000, 0.05636504012656118300,
        -0.00000000000000284785, 0.00000000000000269180, 0.00000000000000026089,
        -0.33659693927260426000, 0.28023189914604330000, 0.05636504012656121800,
        -0.59181480684449039000, 0.48617039139373414000, 0.10564441545075609000,
        0.00000000000000098567, -0.00000000000000095474, -0.00000000000000021207,
        -0.33659693927260698000, 0.28023189914604579000, 0.05636504012656142600,
        -0.59181480684449372000, 0.48617039139373774000, 0.10564441545075645000,
        0.33659693927260514000, -0.28023189914604435000, -0.05636504012656100300,
        -0.00000000000000010012, 0.00000000000000001702, 0.00000000000000012437,
        -0.33659693927260204000, 0.28023189914604152000, 0.05636504012656010100,
        0.59181480684449428000, -0.48617039139373813000, -0.10564441545075638000,
        0.33659693927260081000, -0.28023189914603991000, -0.05636504012656074600,
        0.00000000000000216976, -0.00000000000000195478, -0.00000000000000023527,
        0.39961448116107012000, -0.35734834346184241000, -0.04226613769922773400,
        -0.33634249144114892000, 0.28239332896420155000, 0.05394916247694748300,
        0.39961448116106396000, -0.35734834346183769000, -0.04226613769922723400,
        -0.33634249144114703000, 0.28239332896420027000, 0.05394916247694724100,
        -0.21667948075941171000, 0.12935693076722185000, 0.08732254999219028800,
        -0.33634249144114398000, 0.28239332896419722000, 0.05394916247694688700,
        0.39961448116106157000, -0.35734834346183453000, -0.04226613769922710200,
        -0.33634249144114919000, 0.28239332896420105000, 0.05394916247694810100,
        0.39961448116107307000, -0.35734834346184485000, -0.04226613769922824700,
        -0.54188833749531484000, 0.49456532031183192000, 0.04732301718348254400,
        0.00000000000000042643, -0.00000000000000052416, -0.00000000000000028161,
        0.54188833749532672000, -0.49456532031184147000, -0.04732301718348516700,
```

```
        0.00000000000000208148, -0.00000000000000170526, -0.00000000000000039120,
        -0.00000000000001165642, 0.00000000000000998830, 0.00000000000000133016,
        -0.00000000000000389738, 0.00000000000000286692, 0.00000000000000081238,
        0.54188833749532805000, -0.49456532031184208000, -0.04732301718348581200,
        -0.00000000000000308117, 0.00000000000000212213, 0.00000000000000117840,
        -0.54188833749532439000, 0.49456532031183975000, 0.04732301718348420900,
        0.20000000000000001000, 0.20000000000000001000, 0.20000000000000001000,
        0.20000000000000001000, 0.20000000000000001000, 0.20000000000000001000,
        0.20000000000000001000, 0.20000000000000001000, 0.20000000000000001000,
        0.20000000000000001000, 0.20000000000000001000, 0.20000000000000001000,
        0.20000000000000001000, 0.20000000000000001000, 0.20000000000000001000,
        0.33333333333333331000, 0.33333333333333331000, 0.33333333333333331000,
        0.00000000000000093850, -0.00000000000000054323, -0.00000000000000011761,
        -0.03290466729806285100, 0.00000000000000063771, 0.00000000000000000000,
        0.00000000000000000000, 0.00000000000000000000, 0.00000000000000000000};

    // ***********************************************************************
    // MAIN
    // ***********************************************************************
    public static void main(String[] args) throws Exception {

        double xData[][]; // Input  Attributes for Trainer
        int yData[]; // Output Attributes for Trainer
        int i, j;         // array indicies
        String trainLogName = "BinaryClassificationNetworkEx2.log";
        int[][] z;

        // *******************************************************************
        // PREPROCESS TRAINING PATTERNS
        // *******************************************************************
        long t0 = System.currentTimeMillis();

        xData = new double[nObs][nInputs];
        yData = new int[nObs];

        /* Perform Binary Filtering. */
        for (i = 0; i < data.length; i++) {
            for (j = 0; j < data[0].length; j++) {
                data[i][j]++;
            }
        }
        int xx[] = new int[nObs];
        UnsupervisedNominalFilter filter = new UnsupervisedNominalFilter(3);
        for (i = 0; i < 9; i++) {
            // Copy each variable to a temp var
            for (j = 0; j < nObs; j++) {
                xx[j] = data[j][i];
            }
            //  Perform binary filter on temp var
            z = filter.encode(xx);
            //  Copy binary encoded var to xData
            for (j = 0; j < nObs; j++) {
                for (int k = 0; k < 3; k++) {
                    xData[j][k + (i * 3)] = (double) z[j][k];
                }
            }
```

```
        }

        for (i = 0; i < nObs; i++) {
            yData[i] = (i >= 626 ? 0 : 1);
        }

        // ******************************************************************
        // CREATE FEEDFORWARD NETWORK
        // ******************************************************************
        FeedForwardNetwork network = new FeedForwardNetwork();
        network.getInputLayer().createInputs(nInputs);
        network.createHiddenLayer().createPerceptrons(nPerceptrons1);
        network.createHiddenLayer().createPerceptrons(nPerceptrons2);
        network.getOutputLayer().createPerceptrons(nOutputs);
        network.linkAll();
        network.setWeights(weights);
        Perceptron perceptrons[] = network.getPerceptrons();
        for (i = 0; i < perceptrons.length - 1; i++) {
            perceptrons[i].setActivation(hiddenLayerActivation);
        }
        // ******************************************************************
        // SET OUTPUT LAYER ACTIVATION FUNCTION TO LOGISTIC
        // FOR BINARY CLASSIFICATION
        // ******************************************************************
        perceptrons[perceptrons.length - 1].
                setActivation(outputLayerActivation);

        BinaryClassification classification
                = new BinaryClassification(network);

        QuasiNewtonTrainer stageITrainer = new QuasiNewtonTrainer();
        QuasiNewtonTrainer stageIITrainer = new QuasiNewtonTrainer();
        stageITrainer.setError(classification.getError());
        stageIITrainer.setError(classification.getError());
        stageITrainer.setMaximumTrainingIterations(8000);
        stageITrainer.setMaximumStepsize(10.0);
        stageIITrainer.setMaximumStepsize(10.0);
        stageITrainer.setRelativeTolerance(10e-20);
        stageIITrainer.setRelativeTolerance(10e-20);
        stageIITrainer.setMaximumTrainingIterations(8000);
        EpochTrainer trainer = new EpochTrainer(stageITrainer, stageIITrainer);

        // Set Training Parameters
        trainer.setNumberOfEpochs(20);
        trainer.setEpochSize(nObs);

        // Set random number seeds to produce repeatable output
        trainer.setRandom(new Random(5555));
        trainer.setRandomSamples(new Random(5555), new Random(5555));

        // 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.");
        }
    }
    classification.train(trainer, xData, yData);
    System.out.println("trainer.getErrorValue = "
            + trainer.getErrorValue());
    System.out.println("StageITrainer.getErrorValue = "
            + stageITrainer.getErrorValue());
    System.out.println("StageIITrainer.getErrorValue = "
            + stageIITrainer.getErrorValue());

    // ******************************************************************
    // DISPLAY TRAINING STATISTICS
    // ******************************************************************
    double stats[] = classification.computeStatistics(xData, yData);
    System.out.println("***********************************************");
    System.out.println("--> Cross-entropy error:        "
            + (float) stats[0]);
    System.out.println("--> Classification error rate:  "
            + (float) stats[1]);
    System.out.println("***********************************************");
    System.out.println("");

    // ******************************************************************
    // OBTAIN AND DISPLAY NETWORK WEIGHTS AND GRADIENTS
    // ******************************************************************
    double weight[] = network.getWeights();
    double gradient[] = trainer.getErrorGradient();
    double wg[][] = new double[weight.length][2];
    for (i = 0; i < weight.length; i++) {
        wg[i][0] = weight[i];
        wg[i][1] = gradient[i];
    }
    PrintMatrixFormat pmf = new PrintMatrixFormat();
    pmf.setNumberFormat(new java.text.DecimalFormat("0.000000"));
    pmf.setColumnLabels(new String[]{"Weights", "Gradients"});
    new PrintMatrix().print(pmf, wg);

    // ******************************************************************
    //     forecast the network
    // ******************************************************************
    double report[][] = new double[nObs][2];
    for (i = 0; i < 50; i++) {
        report[i][0] = yData[i];
        report[i][1] = classification.predictedClass(xData[i]);
    }

    pmf = new PrintMatrixFormat();
    pmf.setColumnLabels(new String[]{"Expected", "Predicted"});
    new PrintMatrix("Forecast").print(pmf, report);
```

```
        long t1 = System.currentTimeMillis();
        double time = t1 - t0;
        time = time / 1000;
        System.out.println("****************Time:  " + time);
        System.out.println("trainer.getErrorValue = "
                + trainer.getErrorValue());
        System.out.println("StageITrainer.getErrorValue = "
                + stageITrainer.getErrorValue());
        System.out.println("StageIITrainer.getErrorValue = "
                + stageIITrainer.getErrorValue());
    }
}
```

Output

```
--> Training Log Created in BinaryClassificationNetworkEx2.log
trainer.getErrorValue = 1.4521421559936876
StageITrainer.getErrorValue = 482.27809835973795
StageIITrainer.getErrorValue = 1.4521421559936876
**********************************************
--> Cross-entropy error:        1.4521421
--> Classification error rate:  0.0020876827
**********************************************

      Weights    Gradients
  0    2.794812  -0.000037
  1   11.881356  -0.000066
  2  -11.053385   0.002202
  3   19.260234   0.359990
  4   28.373581  -0.000341
  5  -13.253239  -0.027393
  6   -7.082690   0.015784
  7   13.221642   0.003962
  8  -23.283604  -1.405288
  9  -30.615046  -0.000000
 10   11.737416   0.000001
 11    3.299586  -0.000000
 12  -10.950633   0.000203
 13    2.016166  -0.000140
 14  -20.363654  -0.000000
 15   -8.838920  -0.027421
 16  -12.499779   0.000510
 17   -6.608869   0.006352
 18   17.696999  -1.106129
 19   28.843672  -0.000341
 20    8.045185  -0.000000
 21   10.177528   0.000000
 22   -0.954535   0.000000
 23  -23.413666  -0.000000
 24  -39.003675   0.000000
 25    3.247172  -0.000008
 26    9.765645   0.015208
 27    0.018801   0.000015
 28    4.576900   0.060691
 29  -12.150770   0.000000
```

```
30  -11.276935  -0.000000
31   18.142833   0.015712
32   -4.480160   0.001997
33   21.404600  -1.312999
34  -43.192160  -0.000379
35   12.501761  -0.027452
36  -14.572506   0.000006
37  -13.945651   0.004352
38  -25.733512   0.267603
39   30.022170   0.000038
40    0.869418   0.000023
41    4.846458  -0.000000
42    7.585565   0.000018
43    3.099542  -0.000042
44   -9.807495  -0.000000
45  -18.722817  -0.000026
46    7.144900   0.015791
47  -10.414587   0.003950
48   18.917071  -2.248996
49    1.892356   0.000000
50   12.926283  -0.027369
51   -2.409669  -0.000072
52    5.131728   0.002344
53  -23.509053   1.200103
54  -11.747324  -0.000341
55    6.533518  -0.000034
56    3.184332   0.000000
57   -4.375824   0.000073
58    2.806478   0.003455
59  -12.541122   0.000000
60   28.572629  -0.027369
61   58.397173  -0.000000
62   -1.683803   0.004585
63   21.994267   0.417573
64   -8.148093  -0.000000
65  -28.281093  -0.000060
66  -42.698239   0.015791
67   -5.212427   0.001782
68  -24.822189  -1.463011
69    3.390353  -0.000341
70    1.738623  -0.000000
71   -6.313312  -0.000072
72   -1.223654   0.000000
73    0.774095  -0.000000
74  -18.669978  -0.000000
75  -26.851295  -0.027369
76  -12.628134   0.015201
77   -6.891812   0.002198
78   20.327156   1.260888
79  -24.811981  -0.000341
80   21.850027  -0.000022
81   -0.353371   0.000511
82    4.425916   0.004020
83  -23.040663  -2.306232
84    5.876584   0.000000
85    8.143971  -0.000038
```

```
 86   21.663435   0.000007
 87   -6.285839   0.000148
 88   -0.355297  -0.000094
 89   -4.267329  -0.000000
 90   -7.707898  -0.027393
 91   10.756301   0.000583
 92  -42.148634  -0.000000
 93   19.135975  -0.673146
 94   10.182175   0.000000
 95    4.003089  -0.000036
 96  -22.043461  -0.000066
 97   33.052701   0.006367
 98  -23.321219  -0.429493
 99    8.738982  -0.000341
100    5.451673  -0.000000
101   20.219032   0.015201
102    0.676898  -0.000000
103    2.691869   0.057201
104  -41.170953   0.000000
105   -8.584660  -0.000036
106    1.630423   0.000006
107   -3.854815   0.006352
108   16.548650  -0.579643
109   45.844790  -0.000379
110    2.217830  -0.027393
111    4.890955   0.000583
112    2.642988   0.000015
113  -23.801442  -0.526478
114  -38.698959   0.000000
115    8.331494  -0.000000
116    1.005177   0.015129
117   -8.016478   0.000000
118    5.529196   0.060684
119  -28.818014   0.000038
120    3.604061  -0.027419
121    4.143800   0.000511
122  -10.174349   0.003961
123   20.432096  -1.465940
124   -3.479734   0.000038
125   -5.448064   0.000023
126   -3.779805   0.015201
127   10.354714   0.002390
128  -23.042045   0.420524
129   -8.400539  -0.000379
130    2.973240  -0.000034
131    8.433972   0.000007
132  -11.045431   0.000016
133    1.312826  -0.000023
134  -10.385525  -0.000000
135  -54.999564  -0.019621
136    5.476551  -0.000000
137   -7.417770   0.000000
138   42.092030  -0.018658
139   13.732960  -0.000000
140   23.967486   0.000000
141  -21.369439  -0.031979
```

```
142    3.040890   0.000000
143   -3.357516  -0.000000
144   97.218390  -0.039325
145    4.920216   0.000000
146   34.621998  -0.000012
147   62.555082  -0.907987
148    3.421771   0.000000
149    4.222951  -0.006693
150  183.127735  -0.011089
151  -48.952878  -0.020804
152   13.720097  -0.020303
153    1.395545  -0.027429
154    8.430987   0.015718
155   -9.235164   0.006367
156   -2.045833  -1.045438
157  -22.715138  -0.000341
158  -62.659356  -0.949897
159   13.949309   0.000000
160   -0.675295  -0.006693
161  -60.351363  -0.020804

        Forecast
     Expected  Predicted
  0     1          1
  1     1          1
  2     1          1
  3     1          1
  4     1          1
  5     1          1
  6     1          1
  7     1          1
  8     1          1
  9     1          1
 10     1          1
 11     1          1
 12     1          1
 13     1          1
 14     1          1
 15     1          1
 16     1          1
 17     1          1
 18     1          1
 19     1          1
 20     1          1
 21     1          1
 22     1          1
 23     1          1
 24     1          1
 25     1          1
 26     1          1
 27     1          1
 28     1          1
 29     1          1
 30     1          1
 31     1          1
 32     1          1
```

```
33      1           1
34      1           1
35      1           1
36      1           1
37      1           1
38      1           1
39      1           1
40      1           1
41      1           1
42      1           1
43      1           1
44      1           1
45      1           1
46      1           1
47      1           1
48      1           1
49      1           1
50      0           0
51      0           0
52      0           0
53      0           0
54      0           0
55      0           0
56      0           0
57      0           0
58      0           0
59      0           0
60      0           0
61      0           0
62      0           0
63      0           0
64      0           0
65      0           0
66      0           0
67      0           0
68      0           0
69      0           0
70      0           0
71      0           0
72      0           0
73      0           0
74      0           0
75      0           0
76      0           0
77      0           0
78      0           0
79      0           0
80      0           0
81      0           0
82      0           0
83      0           0
84      0           0
85      0           0
86      0           0
87      0           0
88      0           0
```

```
 89     0         0
 90     0         0
 91     0         0
 92     0         0
 93     0         0
 94     0         0
 95     0         0
 96     0         0
 97     0         0
 98     0         0
 99     0         0
100     0         0
101     0         0
102     0         0
103     0         0
104     0         0
105     0         0
106     0         0
107     0         0
108     0         0
109     0         0
110     0         0
111     0         0
112     0         0
113     0         0
114     0         0
115     0         0
116     0         0
117     0         0
118     0         0
119     0         0
120     0         0
121     0         0
122     0         0
123     0         0
124     0         0
125     0         0
126     0         0
127     0         0
128     0         0
129     0         0
130     0         0
131     0         0
132     0         0
133     0         0
134     0         0
135     0         0
136     0         0
137     0         0
138     0         0
139     0         0
140     0         0
141     0         0
142     0         0
143     0         0
144     0         0
```

145	0	0
146	0	0
147	0	0
148	0	0
149	0	0
150	0	0
151	0	0
152	0	0
153	0	0
154	0	0
155	0	0
156	0	0
157	0	0
158	0	0
159	0	0
160	0	0
161	0	0
162	0	0
163	0	0
164	0	0
165	0	0
166	0	0
167	0	0
168	0	0
169	0	0
170	0	0
171	0	0
172	0	0
173	0	0
174	0	0
175	0	0
176	0	0
177	0	0
178	0	0
179	0	0
180	0	0
181	0	0
182	0	0
183	0	0
184	0	0
185	0	0
186	0	0
187	0	0
188	0	0
189	0	0
190	0	0
191	0	0
192	0	0
193	0	0
194	0	0
195	0	0
196	0	0
197	0	0
198	0	0
199	0	0
200	0	0

```
201        0            0
202        0            0
203        0            0
204        0            0
205        0            0
206        0            0
207        0            0
208        0            0
209        0            0
210        0            0
211        0            0
212        0            0
213        0            0
214        0            0
215        0            0
216        0            0
217        0            0
218        0            0
219        0            0
220        0            0
221        0            0
222        0            0
223        0            0
224        0            0
225        0            0
226        0            0
227        0            0
228        0            0
229        0            0
230        0            0
231        0            0
232        0            0
233        0            0
234        0            0
235        0            0
236        0            0
237        0            0
238        0            0
239        0            0
240        0            0
241        0            0
242        0            0
243        0            0
244        0            0
245        0            0
246        0            0
247        0            0
248        0            0
249        0            0
250        0            0
251        0            0
252        0            0
253        0            0
254        0            0
255        0            0
256        0            0
```

257	0	0
258	0	0
259	0	0
260	0	0
261	0	0
262	0	0
263	0	0
264	0	0
265	0	0
266	0	0
267	0	0
268	0	0
269	0	0
270	0	0
271	0	0
272	0	0
273	0	0
274	0	0
275	0	0
276	0	0
277	0	0
278	0	0
279	0	0
280	0	0
281	0	0
282	0	0
283	0	0
284	0	0
285	0	0
286	0	0
287	0	0
288	0	0
289	0	0
290	0	0
291	0	0
292	0	0
293	0	0
294	0	0
295	0	0
296	0	0
297	0	0
298	0	0
299	0	0
300	0	0
301	0	0
302	0	0
303	0	0
304	0	0
305	0	0
306	0	0
307	0	0
308	0	0
309	0	0
310	0	0
311	0	0
312	0	0

```
313    0    0
314    0    0
315    0    0
316    0    0
317    0    0
318    0    0
319    0    0
320    0    0
321    0    0
322    0    0
323    0    0
324    0    0
325    0    0
326    0    0
327    0    0
328    0    0
329    0    0
330    0    0
331    0    0
332    0    0
333    0    0
334    0    0
335    0    0
336    0    0
337    0    0
338    0    0
339    0    0
340    0    0
341    0    0
342    0    0
343    0    0
344    0    0
345    0    0
346    0    0
347    0    0
348    0    0
349    0    0
350    0    0
351    0    0
352    0    0
353    0    0
354    0    0
355    0    0
356    0    0
357    0    0
358    0    0
359    0    0
360    0    0
361    0    0
362    0    0
363    0    0
364    0    0
365    0    0
366    0    0
367    0    0
368    0    0
```

369	0	0
370	0	0
371	0	0
372	0	0
373	0	0
374	0	0
375	0	0
376	0	0
377	0	0
378	0	0
379	0	0
380	0	0
381	0	0
382	0	0
383	0	0
384	0	0
385	0	0
386	0	0
387	0	0
388	0	0
389	0	0
390	0	0
391	0	0
392	0	0
393	0	0
394	0	0
395	0	0
396	0	0
397	0	0
398	0	0
399	0	0
400	0	0
401	0	0
402	0	0
403	0	0
404	0	0
405	0	0
406	0	0
407	0	0
408	0	0
409	0	0
410	0	0
411	0	0
412	0	0
413	0	0
414	0	0
415	0	0
416	0	0
417	0	0
418	0	0
419	0	0
420	0	0
421	0	0
422	0	0
423	0	0
424	0	0

```
425     0       0
426     0       0
427     0       0
428     0       0
429     0       0
430     0       0
431     0       0
432     0       0
433     0       0
434     0       0
435     0       0
436     0       0
437     0       0
438     0       0
439     0       0
440     0       0
441     0       0
442     0       0
443     0       0
444     0       0
445     0       0
446     0       0
447     0       0
448     0       0
449     0       0
450     0       0
451     0       0
452     0       0
453     0       0
454     0       0
455     0       0
456     0       0
457     0       0
458     0       0
459     0       0
460     0       0
461     0       0
462     0       0
463     0       0
464     0       0
465     0       0
466     0       0
467     0       0
468     0       0
469     0       0
470     0       0
471     0       0
472     0       0
473     0       0
474     0       0
475     0       0
476     0       0
477     0       0
478     0       0
479     0       0
480     0       0
```

481	0	0
482	0	0
483	0	0
484	0	0
485	0	0
486	0	0
487	0	0
488	0	0
489	0	0
490	0	0
491	0	0
492	0	0
493	0	0
494	0	0
495	0	0
496	0	0
497	0	0
498	0	0
499	0	0
500	0	0
501	0	0
502	0	0
503	0	0
504	0	0
505	0	0
506	0	0
507	0	0
508	0	0
509	0	0
510	0	0
511	0	0
512	0	0
513	0	0
514	0	0
515	0	0
516	0	0
517	0	0
518	0	0
519	0	0
520	0	0
521	0	0
522	0	0
523	0	0
524	0	0
525	0	0
526	0	0
527	0	0
528	0	0
529	0	0
530	0	0
531	0	0
532	0	0
533	0	0
534	0	0
535	0	0
536	0	0

```
537     0          0
538     0          0
539     0          0
540     0          0
541     0          0
542     0          0
543     0          0
544     0          0
545     0          0
546     0          0
547     0          0
548     0          0
549     0          0
550     0          0
551     0          0
552     0          0
553     0          0
554     0          0
555     0          0
556     0          0
557     0          0
558     0          0
559     0          0
560     0          0
561     0          0
562     0          0
563     0          0
564     0          0
565     0          0
566     0          0
567     0          0
568     0          0
569     0          0
570     0          0
571     0          0
572     0          0
573     0          0
574     0          0
575     0          0
576     0          0
577     0          0
578     0          0
579     0          0
580     0          0
581     0          0
582     0          0
583     0          0
584     0          0
585     0          0
586     0          0
587     0          0
588     0          0
589     0          0
590     0          0
591     0          0
592     0          0
```

593	0	0
594	0	0
595	0	0
596	0	0
597	0	0
598	0	0
599	0	0
600	0	0
601	0	0
602	0	0
603	0	0
604	0	0
605	0	0
606	0	0
607	0	0
608	0	0
609	0	0
610	0	0
611	0	0
612	0	0
613	0	0
614	0	0
615	0	0
616	0	0
617	0	0
618	0	0
619	0	0
620	0	0
621	0	0
622	0	0
623	0	0
624	0	0
625	0	0
626	0	0
627	0	0
628	0	0
629	0	0
630	0	0
631	0	0
632	0	0
633	0	0
634	0	0
635	0	0
636	0	0
637	0	0
638	0	0
639	0	0
640	0	0
641	0	0
642	0	0
643	0	0
644	0	0
645	0	0
646	0	0
647	0	0
648	0	0

```
649     0           0
650     0           0
651     0           0
652     0           0
653     0           0
654     0           0
655     0           0
656     0           0
657     0           0
658     0           0
659     0           0
660     0           0
661     0           0
662     0           0
663     0           0
664     0           0
665     0           0
666     0           0
667     0           0
668     0           0
669     0           0
670     0           0
671     0           0
672     0           0
673     0           0
674     0           0
675     0           0
676     0           0
677     0           0
678     0           0
679     0           0
680     0           0
681     0           0
682     0           0
683     0           0
684     0           0
685     0           0
686     0           0
687     0           0
688     0           0
689     0           0
690     0           0
691     0           0
692     0           0
693     0           0
694     0           0
695     0           0
696     0           0
697     0           0
698     0           0
699     0           0
700     0           0
701     0           0
702     0           0
703     0           0
704     0           0
```

705	0	0
706	0	0
707	0	0
708	0	0
709	0	0
710	0	0
711	0	0
712	0	0
713	0	0
714	0	0
715	0	0
716	0	0
717	0	0
718	0	0
719	0	0
720	0	0
721	0	0
722	0	0
723	0	0
724	0	0
725	0	0
726	0	0
727	0	0
728	0	0
729	0	0
730	0	0
731	0	0
732	0	0
733	0	0
734	0	0
735	0	0
736	0	0
737	0	0
738	0	0
739	0	0
740	0	0
741	0	0
742	0	0
743	0	0
744	0	0
745	0	0
746	0	0
747	0	0
748	0	0
749	0	0
750	0	0
751	0	0
752	0	0
753	0	0
754	0	0
755	0	0
756	0	0
757	0	0
758	0	0
759	0	0
760	0	0

```
761     0        0
762     0        0
763     0        0
764     0        0
765     0        0
766     0        0
767     0        0
768     0        0
769     0        0
770     0        0
771     0        0
772     0        0
773     0        0
774     0        0
775     0        0
776     0        0
777     0        0
778     0        0
779     0        0
780     0        0
781     0        0
782     0        0
783     0        0
784     0        0
785     0        0
786     0        0
787     0        0
788     0        0
789     0        0
790     0        0
791     0        0
792     0        0
793     0        0
794     0        0
795     0        0
796     0        0
797     0        0
798     0        0
799     0        0
800     0        0
801     0        0
802     0        0
803     0        0
804     0        0
805     0        0
806     0        0
807     0        0
808     0        0
809     0        0
810     0        0
811     0        0
812     0        0
813     0        0
814     0        0
815     0        0
816     0        0
```

817	0	0
818	0	0
819	0	0
820	0	0
821	0	0
822	0	0
823	0	0
824	0	0
825	0	0
826	0	0
827	0	0
828	0	0
829	0	0
830	0	0
831	0	0
832	0	0
833	0	0
834	0	0
835	0	0
836	0	0
837	0	0
838	0	0
839	0	0
840	0	0
841	0	0
842	0	0
843	0	0
844	0	0
845	0	0
846	0	0
847	0	0
848	0	0
849	0	0
850	0	0
851	0	0
852	0	0
853	0	0
854	0	0
855	0	0
856	0	0
857	0	0
858	0	0
859	0	0
860	0	0
861	0	0
862	0	0
863	0	0
864	0	0
865	0	0
866	0	0
867	0	0
868	0	0
869	0	0
870	0	0
871	0	0
872	0	0

```
873      0          0
874      0          0
875      0          0
876      0          0
877      0          0
878      0          0
879      0          0
880      0          0
881      0          0
882      0          0
883      0          0
884      0          0
885      0          0
886      0          0
887      0          0
888      0          0
889      0          0
890      0          0
891      0          0
892      0          0
893      0          0
894      0          0
895      0          0
896      0          0
897      0          0
898      0          0
899      0          0
900      0          0
901      0          0
902      0          0
903      0          0
904      0          0
905      0          0
906      0          0
907      0          0
908      0          0
909      0          0
910      0          0
911      0          0
912      0          0
913      0          0
914      0          0
915      0          0
916      0          0
917      0          0
918      0          0
919      0          0
920      0          0
921      0          0
922      0          0
923      0          0
924      0          0
925      0          0
926      0          0
927      0          0
928      0          0
```

```
929     0           0
930     0           0
931     0           0
932     0           0
933     0           0
934     0           0
935     0           0
936     0           0
937     0           0
938     0           0
939     0           0
940     0           0
941     0           0
942     0           0
943     0           0
944     0           0
945     0           0
946     0           0
947     0           0
948     0           0
949     0           0
950     0           0
951     0           0
952     0           0
953     0           0
954     0           0
955     0           0
956     0           0
957     0           0

****************Time:  3.214
trainer.getErrorValue = 1.4521421559936876
StageITrainer.getErrorValue = 482.27809835973795
StageIITrainer.getErrorValue = 1.4521421559936876
```

MultiClassification class

`public class com.imsl.datamining.neural.MultiClassification implements Serializable`

Classifies patterns into three or more classes.

Extends neural network analysis to solving multi-classification problems. In these problems, the target output for the network is the probability that the pattern falls into each of several classes, where the number of classes is 3 or greater. These probabilities are then used to assign patterns to one of the target classes. Typical applications include determining the credit classification for a business (excellent, good, fair or poor), and determining which of three or more treatments a patient should receive based upon their physical, clinical and laboratory information. This class signals that network training will minimize

the multi-classification cross-entropy error, and that network outputs are the probabilities that the pattern belongs to each of the target classes. These probabilities are scaled to sum to 1.0 using softmax activation.

Constructor

MultiClassification

```
public MultiClassification(Network network)
```

Description

Creates a classifier.

Parameter

`network` – is the neural network used for classification. It's `OutputPerceptrons` should use linear activation functions, `Activation.LINEAR`. The number of `OutputPerceptrons` should equal the number of classes.

Methods

computeStatistics

```
public double[] computeStatistics(double[][] xData, int[] yData)
```

Description

Computes classification statistics for the supplied network patterns and their associated classifications.

Method `computeStatistics` returns a two element array where the first element returned is the cross-entropy error; the second is the classification error rate. The classification error rate is calculated by comparing the estimated classification probabilities to the target classifications. If the estimated probability for the target class is not the largest for among the target classes, then the pattern is tallied as a classification error.

Parameters

`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` – An `int` containing the output classification patterns. The values in `yData` must be in the range of one to the number of `OutputPerceptrons` in the network.

Returns

A `double` array containing the two statistics described above.

getError

```
public QuasiNewtonTrainer.Error getError()
```

Description

Returns the error function for use by `QuasiNewtonTrainer` for training a classification network. This error function combines the softmax activation function and the cross-entropy error function.

Returns

an implementation of the multi-classification cross-entropy error function.

getNetwork

```
public Network getNetwork()
```

Description

Returns the network being used for classification.

Returns

the `network` set by the constructor.

predictedClass

```
public int predictedClass(double[] x)
```

Description

Calculates the classification probablities for the input pattern `x`, and returns the class with the highest probability.

This method classifies patterns into one of the target classes based upon the patterns values.

Parameter

`x` – The `double` array containing the network input patterns to classify. The length of `x` should equal the number of inputs in the network.

Returns

The classification predicted by the trained network for `x`. This will be one of the integers 1,2,...,*nClasses*, where *nClasses* is equal to `nOuptuts`. `nOuptuts` is the number of outputs in the network representing the number classes.

probabilities

```
public double[] probabilities(double[] x)
```

Description

Returns classification probabilities for the input pattern `x`.

The number of probabilities is equal to the number of target classes, which is the number of outputs in the `FeedForwardNetwork`. Each are calculated using the softmax activation for each of the `OutputPerceptrons`. The softmax function transforms the outputs potential z to the probability y by

$$y_i = \mathrm{softmax}_\mathrm{i} = \frac{\mathrm{e}^{Z_i}}{\sum_{j=1}^{C} e^{Z_j}}$$

Parameter

`x` – a `double` array containing the input patterns to classify. The length of `x` must be equal to the number of `InputNodes`.

Returns

A `double` containing the scaled probabilities.

train

```
public void train(Trainer trainer, double[][] xData, int[] yData)
```

Description

Trains the classification neural network using supplied training patterns.

Parameters

`trainer` – A `Trainer` object, which is used to train the network. The error function in any `QuasiNewtonTrainer` included in `trainer` should be set to the error function from this class using the `getError` method.

`xData` – A `double` matrix containing the input training patterns. The number of columns in `xData` must equal the number of `Nodes` in the `InputLayer`. Each row of `xData` contains a training pattern.

`yData` – An `int` array containing the output classification patterns. These values must be in the range of one to the number of `OutputPerceptrons` in the network.

Example 1: MultiClassification

This example trains a 3-layer network using Fisher's Iris data with four continuous input attributes and three output classifications. This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field. The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant.

The structure of the network consists of four input nodes and three layers, with four perceptrons in the first hidden layer, three perceptrons in the second hidden layer and three in the output layer.

The four input attributes represent

1. Sepal length
2. Sepal width
3. Petal length
4. Petal width

The output attribute represents the class of the iris plant and are encoded using binary encoding.

1. Iris Setosa
2. Iris Versicolour

3. Iris Virginica

There are a total of 46 weights in this network, including the bias weights. All hidden layers use the logistic activation function. Since the target output is multi-classification the softmax activation function is used in the output layer and the `MultiClassification` error function class is used by the trainer. The error class `MultiClassification` combines the cross-entropy error claculations and the softmax function.

```
import com.imsl.datamining.neural.*;
import com.imsl.math.*;
import java.io.*;
import java.util.logging.*;

//*****************************************************************************
// Three Layer Feed-Forward Network with 4 inputs, all
// continuous, and 3 classification categories.
//
//  new classification training_ex5.c
//
// This is perhaps the best known database to be found in the pattern
//     recognition literature.  Fisher's paper is a classic in the field.
//     The data set contains 3 classes of 50 instances each,
//     where each class refers to a type of iris plant.  One class is
//     linearly separable from the other 2; the latter are NOT linearly
//     separable from each other.
//
//  Predicted attribute: class of iris plant.
//     1=Iris Setosa, 2=Iris Versicolour, and 3=Iris Virginica
//
//  Input Attributes (4 Continuous Attributes)
//     X1: Sepal length, X2: Sepal width, X3: Petal length,
//         and X4: Petal width
//*****************************************************************************
public class MultiClassificationEx1 implements Serializable {

    private static int nObs = 150;   // number of training patterns
    private static int nInputs = 4;  // 9 nominal coded as 0=x, 1=o, 2=blank
    private static int nOutputs = 3; // one continuous output (nClasses=2)
    private static boolean trace = true; // Turns on/off training log

    // irisData[]:  The raw data matrix.  This is a 2-D matrix with 150 rows
    //              and 5 columns.  The first 4 columns are the continuous
    //              input attributes and the 5th column is the
    //              classification category (1-3).  These data contain no
    //              categorical input attributes.
    private static double[][] irisData = {
        {5.1, 3.5, 1.4, 0.2, 1}, {4.9, 3.0, 1.4, 0.2, 1},
        {4.7, 3.2, 1.3, 0.2, 1}, {4.6, 3.1, 1.5, 0.2, 1},
        {5.0, 3.6, 1.4, 0.2, 1}, {5.4, 3.9, 1.7, 0.4, 1},
        {4.6, 3.4, 1.4, 0.3, 1}, {5.0, 3.4, 1.5, 0.2, 1},
        {4.4, 2.9, 1.4, 0.2, 1}, {4.9, 3.1, 1.5, 0.1, 1},
        {5.4, 3.7, 1.5, 0.2, 1}, {4.8, 3.4, 1.6, 0.2, 1},
        {4.8, 3.0, 1.4, 0.1, 1}, {4.3, 3.0, 1.1, 0.1, 1},
        {5.8, 4.0, 1.2, 0.2, 1}, {5.7, 4.4, 1.5, 0.4, 1},
```

```
{5.4, 3.9, 1.3, 0.4, 1}, {5.1, 3.5, 1.4, 0.3, 1},
{5.7, 3.8, 1.7, 0.3, 1}, {5.1, 3.8, 1.5, 0.3, 1},
{5.4, 3.4, 1.7, 0.2, 1}, {5.1, 3.7, 1.5, 0.4, 1},
{4.6, 3.6, 1.0, 0.2, 1}, {5.1, 3.3, 1.7, 0.5, 1},
{4.8, 3.4, 1.9, 0.2, 1}, {5.0, 3.0, 1.6, 0.2, 1},
{5.0, 3.4, 1.6, 0.4, 1}, {5.2, 3.5, 1.5, 0.2, 1},
{5.2, 3.4, 1.4, 0.2, 1}, {4.7, 3.2, 1.6, 0.2, 1},
{4.8, 3.1, 1.6, 0.2, 1}, {5.4, 3.4, 1.5, 0.4, 1},
{5.2, 4.1, 1.5, 0.1, 1}, {5.5, 4.2, 1.4, 0.2, 1},
{4.9, 3.1, 1.5, 0.1, 1}, {5.0, 3.2, 1.2, 0.2, 1},
{5.5, 3.5, 1.3, 0.2, 1}, {4.9, 3.1, 1.5, 0.1, 1},
{4.4, 3.0, 1.3, 0.2, 1}, {5.1, 3.4, 1.5, 0.2, 1},
{5.0, 3.5, 1.3, 0.3, 1}, {4.5, 2.3, 1.3, 0.3, 1},
{4.4, 3.2, 1.3, 0.2, 1}, {5.0, 3.5, 1.6, 0.6, 1},
{5.1, 3.8, 1.9, 0.4, 1}, {4.8, 3.0, 1.4, 0.3, 1},
{5.1, 3.8, 1.6, 0.2, 1}, {4.6, 3.2, 1.4, 0.2, 1},
{5.3, 3.7, 1.5, 0.2, 1}, {5.0, 3.3, 1.4, 0.2, 1},
{7.0, 3.2, 4.7, 1.4, 2}, {6.4, 3.2, 4.5, 1.5, 2},
{6.9, 3.1, 4.9, 1.5, 2}, {5.5, 2.3, 4.0, 1.3, 2},
{6.5, 2.8, 4.6, 1.5, 2}, {5.7, 2.8, 4.5, 1.3, 2},
{6.3, 3.3, 4.7, 1.6, 2}, {4.9, 2.4, 3.3, 1.0, 2},
{6.6, 2.9, 4.6, 1.3, 2}, {5.2, 2.7, 3.9, 1.4, 2},
{5.0, 2.0, 3.5, 1.0, 2}, {5.9, 3.0, 4.2, 1.5, 2},
{6.0, 2.2, 4.0, 1.0, 2}, {6.1, 2.9, 4.7, 1.4, 2},
{5.6, 2.9, 3.6, 1.3, 2}, {6.7, 3.1, 4.4, 1.4, 2},
{5.6, 3.0, 4.5, 1.5, 2}, {5.8, 2.7, 4.1, 1.0, 2},
{6.2, 2.2, 4.5, 1.5, 2}, {5.6, 2.5, 3.9, 1.1, 2},
{5.9, 3.2, 4.8, 1.8, 2}, {6.1, 2.8, 4.0, 1.3, 2},
{6.3, 2.5, 4.9, 1.5, 2}, {6.1, 2.8, 4.7, 1.2, 2},
{6.4, 2.9, 4.3, 1.3, 2}, {6.6, 3.0, 4.4, 1.4, 2},
{6.8, 2.8, 4.8, 1.4, 2}, {6.7, 3.0, 5.0, 1.7, 2},
{6.0, 2.9, 4.5, 1.5, 2}, {5.7, 2.6, 3.5, 1.0, 2},
{5.5, 2.4, 3.8, 1.1, 2}, {5.5, 2.4, 3.7, 1.0, 2},
{5.8, 2.7, 3.9, 1.2, 2}, {6.0, 2.7, 5.1, 1.6, 2},
{5.4, 3.0, 4.5, 1.5, 2}, {6.0, 3.4, 4.5, 1.6, 2},
{6.7, 3.1, 4.7, 1.5, 2}, {6.3, 2.3, 4.4, 1.3, 2},
{5.6, 3.0, 4.1, 1.3, 2}, {5.5, 2.5, 4.0, 1.3, 2},
{5.5, 2.6, 4.4, 1.2, 2}, {6.1, 3.0, 4.6, 1.4, 2},
{5.8, 2.6, 4.0, 1.2, 2}, {5.0, 2.3, 3.3, 1.0, 2},
{5.6, 2.7, 4.2, 1.3, 2}, {5.7, 3.0, 4.2, 1.2, 2},
{5.7, 2.9, 4.2, 1.3, 2}, {6.2, 2.9, 4.3, 1.3, 2},
{5.1, 2.5, 3.0, 1.1, 2}, {5.7, 2.8, 4.1, 1.3, 2},
{6.3, 3.3, 6.0, 2.5, 3}, {5.8, 2.7, 5.1, 1.9, 3},
{7.1, 3.0, 5.9, 2.1, 3}, {6.3, 2.9, 5.6, 1.8, 3},
{6.5, 3.0, 5.8, 2.2, 3}, {7.6, 3.0, 6.6, 2.1, 3},
{4.9, 2.5, 4.5, 1.7, 3}, {7.3, 2.9, 6.3, 1.8, 3},
{6.7, 2.5, 5.8, 1.8, 3}, {7.2, 3.6, 6.1, 2.5, 3},
{6.5, 3.2, 5.1, 2.0, 3}, {6.4, 2.7, 5.3, 1.9, 3},
{6.8, 3.0, 5.5, 2.1, 3}, {5.7, 2.5, 5.0, 2.0, 3},
{5.8, 2.8, 5.1, 2.4, 3}, {6.4, 3.2, 5.3, 2.3, 3},
{6.5, 3.0, 5.5, 1.8, 3}, {7.7, 3.8, 6.7, 2.2, 3},
{7.7, 2.6, 6.9, 2.3, 3}, {6.0, 2.2, 5.0, 1.5, 3},
{6.9, 3.2, 5.7, 2.3, 3}, {5.6, 2.8, 4.9, 2.0, 3},
{7.7, 2.8, 6.7, 2.0, 3}, {6.3, 2.7, 4.9, 1.8, 3},
{6.7, 3.3, 5.7, 2.1, 3}, {7.2, 3.2, 6.0, 1.8, 3},
{6.2, 2.8, 4.8, 1.8, 3}, {6.1, 3.0, 4.9, 1.8, 3},
```

```
        {6.4, 2.8, 5.6, 2.1, 3}, {7.2, 3.0, 5.8, 1.6, 3},
        {7.4, 2.8, 6.1, 1.9, 3}, {7.9, 3.8, 6.4, 2.0, 3},
        {6.4, 2.8, 5.6, 2.2, 3}, {6.3, 2.8, 5.1, 1.5, 3},
        {6.1, 2.6, 5.6, 1.4, 3}, {7.7, 3.0, 6.1, 2.3, 3},
        {6.3, 3.4, 5.6, 2.4, 3}, {6.4, 3.1, 5.5, 1.8, 3},
        {6.0, 3.0, 4.8, 1.8, 3}, {6.9, 3.1, 5.4, 2.1, 3},
        {6.7, 3.1, 5.6, 2.4, 3}, {6.9, 3.1, 5.1, 2.3, 3},
        {5.8, 2.7, 5.1, 1.9, 3}, {6.8, 3.2, 5.9, 2.3, 3},
        {6.7, 3.3, 5.7, 2.5, 3}, {6.7, 3.0, 5.2, 2.3, 3},
        {6.3, 2.5, 5.0, 1.9, 3}, {6.5, 3.0, 5.2, 2.0, 3},
        {6.2, 3.4, 5.4, 2.3, 3}, {5.9, 3.0, 5.1, 1.8, 3}
    };

    public static void main(String[] args) throws Exception {
        double xData[][] = new double[nObs][nInputs];
        int yData[] = new int[nObs];

        for (int i = 0; i < nObs; i++) {
            for (int j = 0; j < nInputs; j++) {
                xData[i][j] = irisData[i][j];
            }
            yData[i] = (int) irisData[i][4];
        }

        // Create network
        FeedForwardNetwork network = new FeedForwardNetwork();
        network.getInputLayer().createInputs(nInputs);
        network.createHiddenLayer().
                createPerceptrons(4, Activation.LOGISTIC, 0.0);
        network.createHiddenLayer().
                createPerceptrons(3, Activation.LOGISTIC, 0.0);
        network.getOutputLayer().
                createPerceptrons(nOutputs, Activation.SOFTMAX, 0.0);
        network.linkAll();

        MultiClassification classification = new MultiClassification(network);

        // Create trainer
        QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
        trainer.setError(classification.getError());
        trainer.setMaximumTrainingIterations(1000);

        // If tracing is requested setup training logger
        if (trace) {
            Handler handler
                    = new FileHandler("ClassificationNetworkTraining.log");
            Logger logger = Logger.getLogger("com.imsl.datamining.neural");
            logger.setLevel(Level.FINEST);
            logger.addHandler(handler);
            handler.setFormatter(QuasiNewtonTrainer.getFormatter());
        }
        // Train Network
        long t0 = System.currentTimeMillis();
        classification.train(trainer, xData, yData);

        // Display Network Errors
```

```
double stats[] = classification.computeStatistics(xData, yData);
System.out.println("****************************************************");
System.out.println("--> Cross-entropy error:        "
        + (float) stats[0]);
System.out.println("--> Classification error rate:  "
        + (float) stats[1]);
System.out.println("****************************************************");
System.out.println("");

double weight[] = network.getWeights();
double gradient[] = trainer.getErrorGradient();
double wg[][] = new double[weight.length][2];
for (int i = 0; i < weight.length; i++) {
    wg[i][0] = weight[i];
    wg[i][1] = gradient[i];
}
PrintMatrixFormat pmf = new PrintMatrixFormat();
pmf.setNumberFormat(new java.text.DecimalFormat("0.000000"));
pmf.setColumnLabels(new String[]{"Weights", "Gradients"});
new PrintMatrix().print(pmf, wg);

double report[][] = new double[nObs][nInputs + 2];
for (int i = 0; i < nObs; i++) {
    for (int j = 0; j < nInputs; j++) {
        report[i][j] = xData[i][j];
    }
    report[i][nInputs] = irisData[i][4];
    report[i][nInputs + 1] = classification.predictedClass(xData[i]);
}
pmf = new PrintMatrixFormat();
pmf.setColumnLabels(new String[]{
    "Sepal Length",
    "Sepal Width",
    "Petal Length",
    "Petal Width",
    "Expected",
    "Predicted"}
);
new PrintMatrix("Forecast").print(pmf, report);

// **********************************************************************
// DISPLAY CLASSIFICATION STATISTICS
// **********************************************************************
double statsClass[] = classification.computeStatistics(xData, yData);
// Display Network Errors
System.out.println("****************************************************");
System.out.println("--> Cross-Entropy Error:      "
        + (float) statsClass[0]);
System.out.println("--> Classification Error:     "
        + (float) statsClass[1]);
System.out.println("****************************************************");
System.out.println("");
long t1 = System.currentTimeMillis();
double time = t1 - t0;
time = time / 1000;
System.out.println("****************Time:  " + time);
```

```
        System.out.println("Cross-Entropy Error Value = "
                + trainer.getErrorValue());
    }
}
```

Output

```
**********************************************
--> Cross-entropy error:        4.653512
--> Classification error rate:  0.006666667
**********************************************

       Weights     Gradients
 0     -42.381828   0.030801
 1     193.055878   0.000000
 2     -30.384656   0.000000
 3      95.352605   0.000000
 4     -33.692976   0.012782
 5    -282.844912   0.000000
 6    -422.581218   0.000000
 7    -317.896968   0.000000
 8      60.505928   0.023458
 9     -94.286590   0.000000
10     109.828939   0.000000
11    -168.351914   0.000000
12      42.250439   0.008464
13     691.012987   0.000000
14     602.474794   0.000000
15     694.122349   0.000000
16      -3.036409  -1.035514
17    -151.673802  -5.466157
18      75.471899   0.000003
19       3.479083  -1.035303
20      46.614896  -5.466182
21      56.347637   0.032349
22     153.010906  -1.035303
23     204.597137  -5.466182
24      21.557752   0.032349
25      64.075577  -1.035303
26      67.599168  -5.466182
27      78.904088   0.032349
28   -5672.118029   0.000000
29    1244.905099   0.010077
30    4428.212930  -0.010077
31   -5600.671740   0.000000
32    1746.525173   0.004710
33    3855.146566  -0.004710
34   -5562.390356   0.000000
35    1230.760279   0.010431
36    4332.630076  -0.010431
37     -15.417798   0.004103
38     328.841061   0.000000
39     323.847338   0.000000
40     306.946067   0.000000
```

```
41    -214.124377  -1.035303
42    -167.320245  -5.466182
43    -156.514239   0.032349
44   13108.354735   0.000000
45   -2985.466557   0.010413
46  -10122.888178  -0.010413
```

	Sepal Length	Sepal Width	Forecast Petal Length	Petal Width	Expected	Predicted
0	5.1	3.5	1.4	0.2	1	1
1	4.9	3	1.4	0.2	1	1
2	4.7	3.2	1.3	0.2	1	1
3	4.6	3.1	1.5	0.2	1	1
4	5	3.6	1.4	0.2	1	1
5	5.4	3.9	1.7	0.4	1	1
6	4.6	3.4	1.4	0.3	1	1
7	5	3.4	1.5	0.2	1	1
8	4.4	2.9	1.4	0.2	1	1
9	4.9	3.1	1.5	0.1	1	1
10	5.4	3.7	1.5	0.2	1	1
11	4.8	3.4	1.6	0.2	1	1
12	4.8	3	1.4	0.1	1	1
13	4.3	3	1.1	0.1	1	1
14	5.8	4	1.2	0.2	1	1
15	5.7	4.4	1.5	0.4	1	1
16	5.4	3.9	1.3	0.4	1	1
17	5.1	3.5	1.4	0.3	1	1
18	5.7	3.8	1.7	0.3	1	1
19	5.1	3.8	1.5	0.3	1	1
20	5.4	3.4	1.7	0.2	1	1
21	5.1	3.7	1.5	0.4	1	1
22	4.6	3.6	1	0.2	1	1
23	5.1	3.3	1.7	0.5	1	1
24	4.8	3.4	1.9	0.2	1	1
25	5	3	1.6	0.2	1	1
26	5	3.4	1.6	0.4	1	1
27	5.2	3.5	1.5	0.2	1	1
28	5.2	3.4	1.4	0.2	1	1
29	4.7	3.2	1.6	0.2	1	1
30	4.8	3.1	1.6	0.2	1	1
31	5.4	3.4	1.5	0.4	1	1
32	5.2	4.1	1.5	0.1	1	1
33	5.5	4.2	1.4	0.2	1	1
34	4.9	3.1	1.5	0.1	1	1
35	5	3.2	1.2	0.2	1	1
36	5.5	3.5	1.3	0.2	1	1
37	4.9	3.1	1.5	0.1	1	1
38	4.4	3	1.3	0.2	1	1
39	5.1	3.4	1.5	0.2	1	1
40	5	3.5	1.3	0.3	1	1
41	4.5	2.3	1.3	0.3	1	1
42	4.4	3.2	1.3	0.2	1	1
43	5	3.5	1.6	0.6	1	1
44	5.1	3.8	1.9	0.4	1	1
45	4.8	3	1.4	0.3	1	1
46	5.1	3.8	1.6	0.2	1	1

47	4.6	3.2	1.4	0.2	1	1
48	5.3	3.7	1.5	0.2	1	1
49	5	3.3	1.4	0.2	1	1
50	7	3.2	4.7	1.4	2	2
51	6.4	3.2	4.5	1.5	2	2
52	6.9	3.1	4.9	1.5	2	2
53	5.5	2.3	4	1.3	2	2
54	6.5	2.8	4.6	1.5	2	2
55	5.7	2.8	4.5	1.3	2	2
56	6.3	3.3	4.7	1.6	2	2
57	4.9	2.4	3.3	1	2	2
58	6.6	2.9	4.6	1.3	2	2
59	5.2	2.7	3.9	1.4	2	2
60	5	2	3.5	1	2	2
61	5.9	3	4.2	1.5	2	2
62	6	2.2	4	1	2	2
63	6.1	2.9	4.7	1.4	2	2
64	5.6	2.9	3.6	1.3	2	2
65	6.7	3.1	4.4	1.4	2	2
66	5.6	3	4.5	1.5	2	2
67	5.8	2.7	4.1	1	2	2
68	6.2	2.2	4.5	1.5	2	2
69	5.6	2.5	3.9	1.1	2	2
70	5.9	3.2	4.8	1.8	2	2
71	6.1	2.8	4	1.3	2	2
72	6.3	2.5	4.9	1.5	2	2
73	6.1	2.8	4.7	1.2	2	2
74	6.4	2.9	4.3	1.3	2	2
75	6.6	3	4.4	1.4	2	2
76	6.8	2.8	4.8	1.4	2	2
77	6.7	3	5	1.7	2	2
78	6	2.9	4.5	1.5	2	2
79	5.7	2.6	3.5	1	2	2
80	5.5	2.4	3.8	1.1	2	2
81	5.5	2.4	3.7	1	2	2
82	5.8	2.7	3.9	1.2	2	2
83	6	2.7	5.1	1.6	2	3
84	5.4	3	4.5	1.5	2	2
85	6	3.4	4.5	1.6	2	2
86	6.7	3.1	4.7	1.5	2	2
87	6.3	2.3	4.4	1.3	2	2
88	5.6	3	4.1	1.3	2	2
89	5.5	2.5	4	1.3	2	2
90	5.5	2.6	4.4	1.2	2	2
91	6.1	3	4.6	1.4	2	2
92	5.8	2.6	4	1.2	2	2
93	5	2.3	3.3	1	2	2
94	5.6	2.7	4.2	1.3	2	2
95	5.7	3	4.2	1.2	2	2
96	5.7	2.9	4.2	1.3	2	2
97	6.2	2.9	4.3	1.3	2	2
98	5.1	2.5	3	1.1	2	2
99	5.7	2.8	4.1	1.3	2	2
100	6.3	3.3	6	2.5	3	3
101	5.8	2.7	5.1	1.9	3	3
102	7.1	3	5.9	2.1	3	3

```
103     6.3           2.9          5.6           1.8          3          3
104     6.5           3            5.8           2.2          3          3
105     7.6           3            6.6           2.1          3          3
106     4.9           2.5          4.5           1.7          3          3
107     7.3           2.9          6.3           1.8          3          3
108     6.7           2.5          5.8           1.8          3          3
109     7.2           3.6          6.1           2.5          3          3
110     6.5           3.2          5.1           2            3          3
111     6.4           2.7          5.3           1.9          3          3
112     6.8           3            5.5           2.1          3          3
113     5.7           2.5          5             2            3          3
114     5.8           2.8          5.1           2.4          3          3
115     6.4           3.2          5.3           2.3          3          3
116     6.5           3            5.5           1.8          3          3
117     7.7           3.8          6.7           2.2          3          3
118     7.7           2.6          6.9           2.3          3          3
119     6             2.2          5             1.5          3          3
120     6.9           3.2          5.7           2.3          3          3
121     5.6           2.8          4.9           2            3          3
122     7.7           2.8          6.7           2            3          3
123     6.3           2.7          4.9           1.8          3          3
124     6.7           3.3          5.7           2.1          3          3
125     7.2           3.2          6             1.8          3          3
126     6.2           2.8          4.8           1.8          3          3
127     6.1           3            4.9           1.8          3          3
128     6.4           2.8          5.6           2.1          3          3
129     7.2           3            5.8           1.6          3          3
130     7.4           2.8          6.1           1.9          3          3
131     7.9           3.8          6.4           2            3          3
132     6.4           2.8          5.6           2.2          3          3
133     6.3           2.8          5.1           1.5          3          3
134     6.1           2.6          5.6           1.4          3          3
135     7.7           3            6.1           2.3          3          3
136     6.3           3.4          5.6           2.4          3          3
137     6.4           3.1          5.5           1.8          3          3
138     6             3            4.8           1.8          3          3
139     6.9           3.1          5.4           2.1          3          3
140     6.7           3.1          5.6           2.4          3          3
141     6.9           3.1          5.1           2.3          3          3
142     5.8           2.7          5.1           1.9          3          3
143     6.8           3.2          5.9           2.3          3          3
144     6.7           3.3          5.7           2.5          3          3
145     6.7           3            5.2           2.3          3          3
146     6.3           2.5          5             1.9          3          3
147     6.5           3            5.2           2            3          3
148     6.2           3.4          5.4           2.3          3          3
149     5.9           3            5.1           1.8          3          3

***********************************************
--> Cross-Entropy Error:      4.653512
--> Classification Error:     0.006666667
***********************************************

****************Time:  0.546
Cross-Entropy Error Value = 4.653511831588219
```

Example 2: MultiClassification

This example trains a 2-layer network using three binary inputs (X0, X1, X2) and one three-level classification (Y). Where

Y = 0 if X1 = 1

Y = 1 if X2 = 1

Y = 2 if X3 = 1

```
import com.imsl.datamining.neural.*;
import com.imsl.math.*;
import java.io.*;
import java.util.logging.*;

//*****************************************************************************
// Two-Layer FFN with 3 binary inputs (X0, X1, X2) and one three-level
// classification variable (Y)
// Y = 0 if X1 = 1
// Y = 1 if X2 = 1
// Y = 2 if X3 = 1
//  (training_ex6)
//*****************************************************************************
public class MultiClassificationEx2 implements Serializable {

    private static int nObs = 6;     // number of training patterns
    private static int nInputs = 3;  // 3 inputs, all categorical
    private static int nOutputs = 3; // output
    private static boolean trace = true; // Turns on/off training log
    private static double xData[][] = {
        {1, 0, 0}, {1, 0, 0}, {0, 1, 0}, {0, 1, 0}, {0, 0, 1}, {0, 0, 1}
    };
    private static int yData[] = {1, 1, 2, 2, 3, 3};

    private static double weights[] = {
        1.29099444873580580000, -0.64549722436790280000, -0.64549722436790291000,
        0.00000000000000000000, 1.11803398874989490000, -1.11803398874989470000,
        0.57735026918962584000, 0.57735026918962584000, 0.57735026918962584000,
        0.33333333333333331000, 0.33333333333333331000, 0.33333333333333331000,
        0.33333333333333331000, 0.33333333333333331000, 0.33333333333333331000,
        0.33333333333333331000, 0.33333333333333331000, 0.33333333333333331000,
        -0.00000000000000005851, -0.00000000000000005851, -0.57735026918962573000,
        0.00000000000000000000, 0.00000000000000000000, 0.00000000000000000000
    };

    public static void main(String[] args) throws Exception {
        FeedForwardNetwork network = new FeedForwardNetwork();
        network.getInputLayer().createInputs(nInputs);
        network.createHiddenLayer().
                createPerceptrons(3, Activation.LINEAR, 0.0);
        network.getOutputLayer().
                createPerceptrons(nOutputs, Activation.SOFTMAX, 0.0);
        network.linkAll();
        network.setWeights(weights);
```

```
MultiClassification classification = new MultiClassification(network);

QuasiNewtonTrainer trainer = new QuasiNewtonTrainer();
trainer.setError(classification.getError());
trainer.setMaximumTrainingIterations(1000);
trainer.setFalseConvergenceTolerance(1.0e-20);
trainer.setGradientTolerance(1.0e-20);
trainer.setRelativeTolerance(1.0e-20);
trainer.setStepTolerance(1.0e-20);

// If tracing is requested setup training logger
if (trace) {
    Handler handler = new FileHandler("ClassificationNetworkEx2.log");
    Logger logger = Logger.getLogger("com.imsl.datamining.neural");
    logger.setLevel(Level.FINEST);
    logger.addHandler(handler);
    handler.setFormatter(QuasiNewtonTrainer.getFormatter());
}
// Train Network
classification.train(trainer, xData, yData);

// Display Network Errors
double stats[] = classification.computeStatistics(xData, yData);
System.out.println("***********************************************");
System.out.println("--> Cross-Entropy Error:      "
        + (float) stats[0]);
System.out.println("--> Classification Error:     "
        + (float) stats[1]);
System.out.println("***********************************************");
System.out.println();

double weight[] = network.getWeights();
double gradient[] = trainer.getErrorGradient();
double wg[][] = new double[weight.length][2];
for (int i = 0; i < weight.length; i++) {
    wg[i][0] = weight[i];
    wg[i][1] = gradient[i];
}
PrintMatrixFormat pmf = new PrintMatrixFormat();
pmf.setNumberFormat(new java.text.DecimalFormat("0.000000"));
pmf.setColumnLabels(new String[]{"Weights", "Gradients"});
new PrintMatrix().print(pmf, wg);

double report[][] = new double[nObs][nInputs + nOutputs + 2];
for (int i = 0; i < nObs; i++) {
    for (int j = 0; j < nInputs; j++) {
        report[i][j] = xData[i][j];
    }
    report[i][nInputs] = yData[i];
    double p[] = classification.probabilities(xData[i]);
    for (int j = 0; j < nOutputs; j++) {
        report[i][nInputs + 1 + j] = p[j];
    }
    report[i][nInputs + nOutputs + 1]
            = classification.predictedClass(xData[i]);
}
```

```
        pmf = new PrintMatrixFormat();
        pmf.setColumnLabels(new String[]{"X1", "X2", "X3", "Y", "P(C1)",
            "P(C2)", "P(C3)", "Predicted"});
        new PrintMatrix("Forecast").print(pmf, report);
        System.out.println("Cross-Entropy Error Value = "
                + trainer.getErrorValue());

        // ******************************************************************
        // DISPLAY CLASSIFICATION STATISTICS
        // ******************************************************************
        double statsClass[] = classification.computeStatistics(xData, yData);
        // Display Network Errors
        System.out.println("***********************************************");
        System.out.println("--> Cross-Entropy Error:      "
                + (float) statsClass[0]);
        System.out.println("--> Classification Error:     "
                + (float) statsClass[1]);
        System.out.println("***********************************************");
        System.out.println("");
    }
}
```

Output

```
***********************************************
--> Cross-Entropy Error:      0.0
--> Classification Error:     0.0
***********************************************

     Weights   Gradients
 0   3.401208  -0.000000
 1  -4.126657   0.000000
 2  -2.201606  -0.000000
 3  -2.009527   0.000000
 4   3.173323  -0.000000
 5  -4.200377  -0.000000
 6   0.028736  -0.000000
 7   2.657051   0.000000
 8   4.868134  -0.000000
 9   3.711295  -0.000000
10  -2.723536  -0.000000
11   0.012241   0.000000
12  -4.996359   0.000000
13   4.296983   0.000000
14   1.699376  -0.000000
15  -1.993114   0.000000
16  -4.048833   0.000000
17   7.041948  -0.000000
18  -0.447927  -0.000000
19   0.653830   0.000000
20  -0.925019  -0.000000
21  -0.078963   0.000000
22   0.247835   0.000000
23  -0.168872  -0.000000
```

```
                     Forecast
   X1  X2  X3  Y  P(C1)  P(C2)  P(C3)  Predicted
0  1   0   0   1    1      0      0        1
1  1   0   0   1    1      0      0        1
2  0   1   0   2    0      1      0        2
3  0   1   0   2    0      1      0        2
4  0   0   1   3    0      0      1        3
5  0   0   1   3    0      0      1        3

Cross-Entropy Error Value = 0.0
***********************************************
--> Cross-Entropy Error:      0.0
--> Classification Error:     0.0
***********************************************
```

ScaleFilter class

`public class com.imsl.datamining.neural.ScaleFilter implements Serializable`

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:

$$z = r(x - \text{realMin}) + \text{targetMin},$$

where

$$r = \frac{\text{targetMax} - \text{targetMin}}{\text{realMax} - \text{realMin}}.$$

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

$$\text{MAD} = \text{median}\{\mid x - \text{median}\{x\} \mid\}$$

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 ith variable and the median of those values.

If the method decode is called then an unscaling operation is conducted by inverting using:

$$x = \frac{(z - \text{targetMin})}{r} + \text{realMin}.$$

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:

$$x = \frac{spread \cdot (z - \text{targetMin})}{r} + spread \cdot \text{realMin} + center$$

Fields

BOUNDED_SCALING

```
static final public int BOUNDED_SCALING
```

Flag to indicate bounded scaling.

BOUNDED_Z_SCORE_SCALING_MEAN_STDEV

```
static final public int BOUNDED_Z_SCORE_SCALING_MEAN_STDEV
```

Flag to indicate bounded z-score scaling using the mean and standard deviation.

BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD

```
static final public int BOUNDED_Z_SCORE_SCALING_MEDIAN_MAD
```

Flag to indicate bounded z-score scaling using the median and mean absolute difference.

NO_SCALING

```
static final public int NO_SCALING
```

Flag to indicate no scaling.

UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV

```
static final public int UNBOUNDED_Z_SCORE_SCALING_MEAN_STDEV
```

Flag to indicate unbounded z-score scaling using the mean and standard deviation.

UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD

```
static final public int UNBOUNDED_Z_SCORE_SCALING_MEDIAN_MAD
```

Flag to indicate unbounded z-score scaling using the median and mean absolute difference.

Constructor

ScaleFilter

`public ScaleFilter(int scalingMethod)`

Description

Constructor for `ScaleFilter`.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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, measuresof center and spreadfor **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

Thecalculations 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.math.*;
import com.imsl.datamining.neural.*;

public class ScaleFilterEx1 {

    public static void main(String args[]) {
        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);
        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
```

UnsupervisedNominalFilter class

`public class com.imsl.datamining.neural.UnsupervisedNominalFilter implements Serializable`

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[]`.

Constructor

UnsupervisedNominalFilter

```
public UnsupervisedNominalFilter(int nClasses)
```

Description

Constructor for `UnsupervisedNominalFilter`.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

`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.

Parameter

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

public class UnsupervisedNominalFilterEx1 {

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

UnsupervisedOrdinalFilter class

`public class com.imsl.datamining.neural.UnsupervisedOrdinalFilter implements Serializable`

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.

Fields

TRANSFORM_ASIN_SQRT

```
static final public int TRANSFORM_ASIN_SQRT
```

Flag to indicate the arcsine square root transform will be applied to the percentages.

TRANSFORM_NONE

```
static final public int TRANSFORM_NONE
```

Flag to indicate no transformation of percentages.

TRANSFORM_SQRT

```
static final public int TRANSFORM_SQRT
```

Flag to indicate the 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:
`com.imsl.datamining.neural.UnsupervisedOrdinalFilter.TRANSFORM_NONE` (p. 2216) ,
`com.imsl.datamining.neural.UnsupervisedOrdinalFilter.TRANSFORM_SQRT` (p. 2216) ,
`com.imsl.datamining.neural.UnsupervisedOrdinalFilter.TRANSFORM_ASIN_SQRT` (p. 2216)

Methods

decode

```
public int decode(double z)
```

Description

Decodes an encoded ordinal variable.

Parameter

`z` – A `double` containing the encoded value to be decoded.

Returns

An `int` containing the ordinal category associated with `y`.

decode

```
public int[] decode(double[] z)
```

Description

Decodes an array of encoded ordinal values.

Parameter

`z` – 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.

Parameter

`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.

Parameter

`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.

Parameter

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

public class UnsupervisedOrdinalFilterEx1 {

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

TimeSeriesFilter class

`public class com.imsl.datamining.neural.TimeSeriesFilter implements Serializable`

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 $= 0, 1, \ldots \text{nLags}$.

The output data array, z, can be symbolically represented as:

$$z = |x(0) : x(1) : x(2) : \ldots : x(\text{nLags} - 1)|,$$

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:

$$z = \begin{bmatrix} 1 & 6 & 2 & 7 & 3 & 8 \\ 2 & 7 & 3 & 8 & 4 & 9 \\ 3 & 8 & 4 & 9 & 5 & 10 \end{bmatrix}$$

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.math.*;
import com.imsl.datamining.neural.*;

public class TimeSeriesFilterEx1 {

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

TimeSeriesClassFilter class

`public class com.imsl.datamining.neural.TimeSeriesClassFilter implements Serializable`

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[1]`=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.

Constructor

TimeSeriesClassFilter

`public TimeSeriesClassFilter(int nClasses)`

Description

Constructor for `TimeSeriesClassFilter`.

Parameter

`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.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[]) {
        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 32: Decision Trees

Types

class TreeNode . . . 2229
class Tree . . . 2234
class DecisionTree . . . 2237
class DecisionTreeInfoGain . . . 2265
class ALACART . . . 2269
class C45 . . . 2275
class CHAID . . . 2283
class QUEST . . . 2289
class RandomTrees . . . 2298

Usage Notes

Decision Trees - An Overview

Decision trees are data mining methods for predicting a single response variable based on multiple predictor variables. If the response variable is categorical or discrete, the data mining problem is a classification problem, whereas if the response is continuous, the problem is a type of egression problem. Decision trees are generally applicable in both situations.

A simple example involves the decision to play golf or not, depending on the weather. The training data, from Quinlan (1993), is given in Table 30.1 and a decision tree fit to the data is shown in Figure 30.1. Other examples include predicting the chance of survival for heart attack patients based on age, blood pressure and other vital signs; scoring loan applications based on credit history, income, and education; classifying an email as spam based on its characteristics, and so on.

Tree-growing algorithms have similar steps: starting with all observations in a root node, a predictor variable is selected to split the dataset into two or more child nodes or branches. The form of the split depends on the type of predictor and on specifics of the algorithm. If the predictor is categorical, taking discrete values {A, B, C, D} for example, the split may consist of two or more proper subsets, such as {A}, {B, C}, and {D}. If the predictor is continuous, a split will consist of two or more intervals, such as $X \leq 2$, $X > 2$. The splitting procedure is then repeated for each child node and continued in such manner until one of several possible stopping conditions is met. The result of the decision tree algorithm

is a tree structure with a root and a certain number of branches (or nodes). Each branch defines a subset or partition of the data and, conditional on that subset of data, a predicted value for the response variable. A traversal of a branch of the tree thus leads to a prediction, or *decision* about the response variable. To predict a new observation, *out-of-sample*, we find the terminal node to which the observation belongs by traversing the tree and finding the data subset (branch) that contains the observation.

For example, the decision tree in Figure 30.1 can be expressed as a set of rules: If the weather is sunny, don't play golf. If the weather is overcast, play golf. If the weather is rainy and there is no wind, play golf. On the other hand, if it is rainy and windy, don't play golf.

Decision trees are intuitive and can be very effective predictive tools. As with any predictive model, a decision tree should be tested on hold-out datasets or refined using K-fold cross-validation to prevent over-fitting.

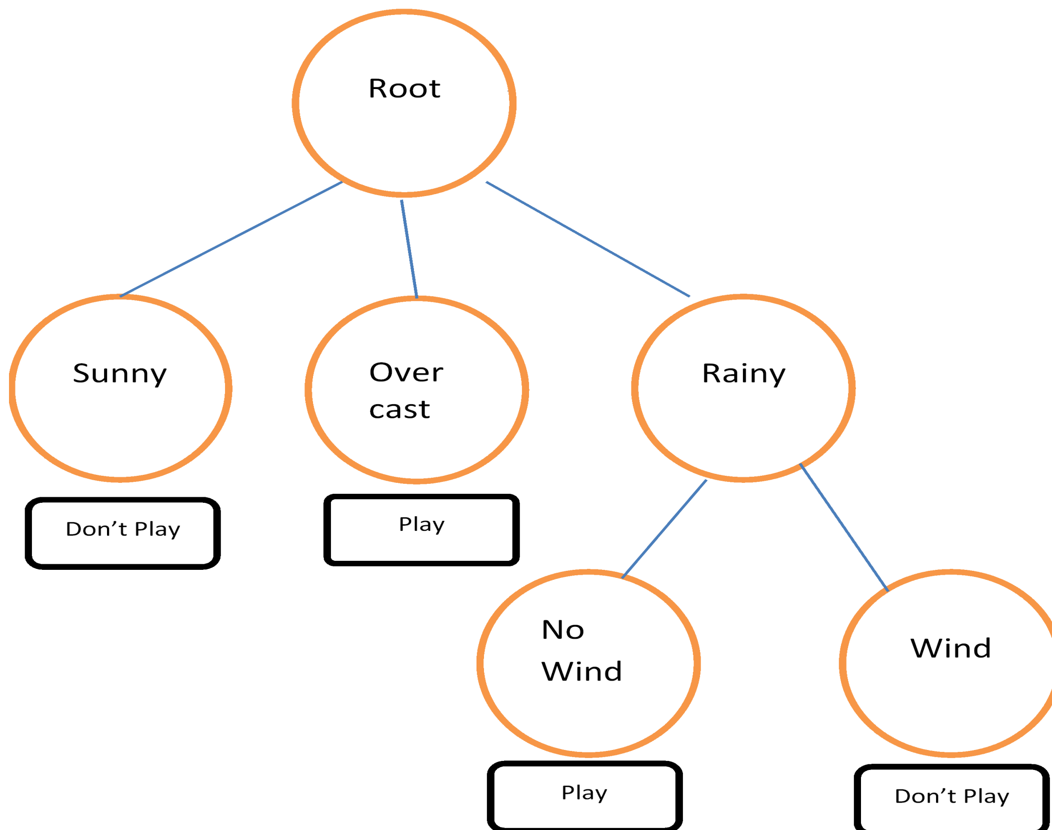


Figure 30.1 - Play Golf? This tree has size=6 nodes, 4 of them terminal nodes, and a height or depth of 2.

Outlook	Temperature	Humidity	Wind	Play
sunny	85	85	false	don't play
sunny	80	90	true	don't play
overcast	83	78	false	play
rainy	70	96	false	play
rainy	68	80	false	play
rainy	65	70	true	don't play
overcast	64	65	true	play
sunny	72	95	false	don't play
sunny	69	70	false	play
rainy	75	80	false	play
sunny	75	70	true	play
overcast	72	90	true	play
overcast	81	75	false	play
rainy	71	80	true	don't play
overcast	81	75	false	play
rainy	71	80	true	don't play

Table 30.1 - Golf training data

TreeNode class

`public class com.imsl.datamining.decisionTree.TreeNode implements Serializable, Cloneable`

A `com.imsl.datamining.decisionTree.DecisionTree` (p. 2237) node that is a child node of `com.imsl.datamining.decisionTree.Tree` (p. 2234) .

Constructor

TreeNode

`public TreeNode()`

Description

Constructs a `DecisionTreeNode` object.

Methods

clone

`protected TreeNode clone()`

Description

Returns a clone of a `TreeNode`.

Returns

a `TreeNode` object that is a clone of the node

getChildId

`public int getChildId(int i)`

Description

Returns the id of a child node.

Parameter

`i` – an `int` specifying the index of the child node

Returns

an `int`, the id of the child node at index `i`

getChildrenIds

`public int[] getChildrenIds()`

Description

Returns the array of child node id's.

Note that the length of this array may exceed the actual number of child nodes. Use `com.imsl.datamining.decisionTree.TreeNode.getNumberOfChildren` (p. 2232) to determine the actual number of children.

Returns

an `int` array containing id's of the child nodes

getCost

`public double getCost()`

Description

Returns the misclassification cost of a node (in-sample cost measure at the current node).

Returns

a `double`, the misclassification cost of a node

getNodeId

`public int getNodeId()`

Description

Returns the id of the current node.

The root node has `nodeId` = 0.

Returns

an `int`, the node id

getNodeSplitCriteriaValue

```
public double getNodeSplitCriteriaValue()
```

Description

Returns the value of the split criteria at the node.

Returns

a `double`, the value of the split criteria at the node

getNodeSplitValue

```
public double getNodeSplitValue()
```

Description

Returns the value around which the node may be split, if the split variable is of a continuous type.

Returns

a `double`, the value around which the node may be split

getNodeValueIndicator

```
public int getNodeValueIndicator(int i)
```

Description

Returns the indicator value for the `i`-th value of the split variable in the current node, if the split variable is categorical

Returns

an `int`, that if 0, the value is not in the node; if 1, it is in the node, meaning that it is in the subset that defines the data partition of the node

getNodeValuesIndicator

```
public int[] getNodeValuesIndicator()
```

Description

Returns the array indicating which values of the split variable apply in the current node, if the split variable is of discrete type.

The array at index `i` will be 1 if the `i`-th value of the discrete variable belongs in the node. If it does not belong to the node, the array has value 0.

Returns

an `int` array containing the indicators

getNodeVariableId

```
public int getNodeVariableId()
```

Description

Returns the id of the variable that defines the split in the current node.

Returns

an `int`, the variable that defines the split in the current node

getNumberOfCases

```
public double getNumberOfCases()
```

Description

Returns the number of cases in the training data that fall into the current node.

Returns

a `double`, the number of cases (or total case weight) in the training data that fall into the current node

getNumberOfChildren

```
public int getNumberOfChildren()
```

Description

Returns the number of child nodes associated with the current node.

Returns

an `int`, the number of child nodes associated with this `TreeNode` instance

getParentId

```
public int getParentId()
```

Description

Returns the id of the parent node of the current node.

Returns

an `int`, the id of the parent node

getPredictedClass

```
public int getPredictedClass()
```

Description

Returns the predicted class at the current node, for response variables of categorical type.

Returns

an `int`, the predicted class

getPredictedVal

```
public double getPredictedVal()
```

Description

Returns the predicted value at the current node for response variables of continuous type.

Returns

a `double`, the predicted value at the current node

getSurrogateInfo

```
public double[] getSurrogateInfo()
```

Description

Returns the surrogate split information array.

Returns

a `double` array containing the surrogate split values

getSurrogateInfo

```
public double getSurrogateInfo(int i)
```

Description

Returns a value from the surrogate split information array.

Parameter

`i` – an `int` specifying which value to return

Returns

a `double`, the value at index `i`

getYProb

```
public double getYProb(int i)
```

Description

Returns a class probability at the current node, if the response variable is of categorical type.

Parameter

`i` – an `int` specifying the index of the class

Returns

a `double`, the probability of the class at index `i`

getYProbs

```
public double[] getYProbs()
```

Description

Returns the class probabilities at the current node, if the response variable is of categorical type.

Returns

a `double` array containing the class probabilities at the current node

Tree class

`public class com.imsl.datamining.decisionTree.Tree implements Serializable, Cloneable`

Serves as the root node of a decision tree and contains information about the relationship of child nodes.

Constructor

Tree

`public Tree(int nRows, int minObsAllowedInChild, int maxNumberOfCategories, int maxNumberOfNodes, int nClasses, int nPreds, PredictiveModel.VariableType varType, int[] predNValues, PredictiveModel.VariableType[] predType, int nyMissing)`

Description

Creates the root node of a decision tree and contains information about the relationship of child nodes.

Parameters

`nRows` – an `int`, the number of observations (rows) in the input data

`minObsAllowedInChild` – an `int`, the number of observations necessary in all potential child nodes before a node may be split

`maxNumberOfCategories` – an `int`, the maximum number of categories allowed for categorical predictor variables

`maxNumberOfNodes` – an `int`, the maximum number of nodes allowed in the tree

`nClasses` – an `int`, the number of classes assumed by the response variable, if the response variable is categorical

`nPreds` – an `int`, the number of predictors used in the model

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array of length equal to the number of variables active in the model, which is less than or equal to the available columns in the data, (`xy[i].length`)

`predNValues` – an `int` array containing the number of values of each predictor variable

`predType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array of length equal to the number of predictors containing the type of each predictor variable

`nyMissing` – an `int` equal to the number of missing values in the response variable

Methods

clone

`protected Tree clone()`

Description

Returns a clone of this object.

Returns

a `Tree` that is a copy of this object

getNode

`public TreeNode getNode(int i)`

Description

Returns a copy of the specified node of the decision tree.

Parameter

`i` – an `int` that specifies the index of a `TreeNode` to be returned

Returns

a `TreeNode` corresponding to the specified index

getNodes

`public TreeNode[] getNodes()`

Description

Returns nodes within a decision tree.

Returns

a `TreeNode` array containing a copy of the `TreeNodes` within the decision tree

getNumberOfClasses

`public int getNumberOfClasses()`

Description

Returns the number of classes assumed by the response variable, if the response variable is categorical.

Returns

an `int`, the number of classes assumed by the response variable, if the response variable is categorical

getNumberOfLevels

`public int getNumberOfLevels()`

Description

Returns the number of levels or depth of a tree.

Returns

an `int` specifying the number of levels or depth of a tree

getNumberOfNodes

```
public int getNumberOfNodes()
```

Description

Returns the number of nodes (size of a tree).

Returns

an `int`, the number of nodes or size of the tree

getNumberOfPredictors

```
public int getNumberOfPredictors()
```

Description

Returns the number of predictors used in the model.

Returns

an `int`, the number of predictors used in the model

getNumberOfSurrogateSplits

```
public int getNumberOfSurrogateSplits()
```

Description

Returns the number of surrogate splits searched for at each tree node.

Surrogate splits are relevant only for classes that implement the `com.imsl.datamining.decisionTree.DecisionTreeSurrogateMethod` interface.

Returns

an `int`, the number of surrogate splits searched for at each node

getPredictorNumberOfValues

```
public int[] getPredictorNumberOfValues()
```

Description

Returns the number of distinct values of each predictor variable.

For continuous predictor variables, the value is set to 0 and is not meaningful.

Returns

an `int` array containing the number of values of each predictor variable

getPredictorType

```
public PredictiveModel.VariableType getPredictorType(int i)
```

Description

Returns the `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) of a predictor variable.

Parameter

i – an int, the index of the predictor

Returns

a VariableType, the variable type of the predictor at index i

getResponseType

```
public PredictiveModel.VariableType getResponseType()
```

Description

Returns the com.imsl.datamining.PredictiveModel.VariableType (p. 1960) of the response variable.

Returns

a PredictiveModel.VariableType, indicates the response variable type

getTerminalNodeIndicators

```
public boolean[] getTerminalNodeIndicators()
```

Description

Returns the terminal node indicator array.

Returns

a boolean array indicating which nodes are terminal nodes and which are not. If true at index i, then the node at index i is a terminal node.

isTerminalNode

```
public boolean isTerminalNode(int i)
```

Description

Returns the terminal node indicator of the node at the given index.

Parameter

i – an int specifying the index of the node

Returns

a boolean, that if true, then the node at index i is a terminal node

DecisionTree class

```
abstract public class com.imsl.datamining.decisionTree.DecisionTree extends
com.imsl.datamining.PredictiveModel implements Serializable, Cloneable
```

Abstract class for generating a decision tree for a single response variable and one or more predictor variables.

Tree Generation Methods

This package contains four of the most widely used algorithms for decision trees (`com.imsl.datamining.decisionTree.C45` (p. 2275) , `com.imsl.datamining.decisionTree.ALACART` (p. 2269) , `com.imsl.datamining.decisionTree.CHAID` (p. 2283) , and `com.imsl.datamining.decisionTree.QUEST` (p. 2289)). The user may also provide an alternate algorithm by extending the `com.imsl.datamining.decisionTree.DecisionTree` (p. 2237) or `com.imsl.datamining.decisionTree.DecisionTreeInfoGain` (p. 2265) abstract class and implementing the abstract method `com.imsl.datamining.decisionTree.DecisionTree` (p. 2237) .

Optimal Tree Size

Minimum Cost-complexity Pruning

A strategy to address overfitting is to grow the tree as large as possible, and then use some logic to prune it back. Let T represent a decision tree generated by any of the methods above. The idea (from Breiman, et. al.) is to find the smallest sub-tree of T that minimizes the cost-complexity measure:

$$R_\delta(T) = R(T) + \delta|\tilde{T}|,$$

$\tilde{T}$ denotes the set of terminal nodes. $|\tilde{T}|$ represents the number of terminal nodes, and $\delta \geq 0$ is a cost-complexity parameter. For a categorical target variable

$$R(T) = \sum_{t \in \tilde{T}} R(t) = \sum_{t \in \tilde{T}} r(t)p(t)$$

$$r(t) = \min_i \sum_j C(i|j)p(j|t)$$

$$p(t) = \Pr[x \in t]$$

$$\text{and } p(j|t) = \Pr[y = j|x \in t],$$

and $C(i|j)$ is the cost for misclassifying the actual class j as i. Note that $C(j|j) = 0$ and $C(i|j) > 0$, for $i \neq j$.

When the target is continuous (and the problem is a regression problem), the metric is instead the mean squared error

$$R(T) = \sum_{t \in \tilde{T}} R(t) = \frac{1}{N} \sum_{t \in \tilde{T}} \sum_{y_n \in t} (y_n - \hat{y}(t))^2$$

This class implements the optimal pruning algorithm 10.1, page 294 in Breiman, et. al (1984). The result of the algorithm is a sequence of sub-trees $T_{\max} \succ T_1 \succ T_2 \succ T_{M-1} \succ \{t_0\}$ obtained by pruning the fully generated tree, $T_{\max}$, until the sub-tree consists of the single root node, $\{t_0\}$. Corresponding to the sequence of sub-trees is the sequence of complexity values, $0 \leq \delta_{\min} < \delta_1 < \delta_2 < \ldots < \delta_{M-1} < \delta_M$ where M is the number of steps it takes in the algorithm to reach the root node. The sub-trees represent the optimally-pruned sub-trees for the sequence of complexity values. The minimum complexity $\delta_{\min}$ can be set via an optional argument.

V-Fold Cross-Validation

The `com.imsl.datamining.CrossValidation` (p. 1969) class can be used for model validation.

Prediction

Ensemble Methods

The `com.imsl.datamining.BootstrapAggregation` (p. 1962) class provides predictions through an ensemble of fitted trees, where the training is done on bootstrap samples.

The `GradientBoosting` (p. **??**) class provides predictions through an ensemble of trees trained on random subsets of the data and iteratively refined using the stochastic gradient boosting algorithm.

The `com.imsl.datamining.decisionTree.RandomTrees` (p. 2298) class provides predictions through an ensemble of fitted trees, where the training is done on bootstrap samples and random subsets of predictors.

Missing Values

Any observation or case with a missing response variable is eliminated from the analysis. If a predictor has a missing value, each algorithm skips that case when evaluating the given predictor. When making a prediction for a new case, if the split variable is missing, the prediction function applies *surrogate* split-variables and splitting rules in turn, if they are estimated with the decision tree. Otherwise, the prediction function returns the prediction from the most recent non-terminal node. In this implementation, only `com.imsl.datamining.decisionTree.ALACART` (p. 2269) estimates surrogate split variables when requested.

Constructor

DecisionTree

```
public DecisionTree(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a `DecisionTree` object for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

fitModel

```
public void fitModel() throws PredictiveModel.PredictiveModelException,
DecisionTree.PruningFailedToConvergeException,
PredictiveModel.StateChangeException, DecisionTree.PureNodeException,
PredictiveModel.SumOfProbabilitiesNotOneException,
DecisionTree.MaxTreeSizeExceededException
```

Description

Fits the decision tree. Implements the abstract method.

Exceptions

`PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

`PruningFailedToConvergeException` is thrown when pruning fails to converge.

`PredictiveModel.StateChangeException` is thrown when an input parameter changes that might affect the model estimates or predictions.

`PureNodeException` is thrown when attempting to split a node that is already pure.

`PredictiveModel.SumOfProbabilitiesNotOneException` is thrown when the sum of probabilities is not approximately one.

`MaxTreeSizeExceededException` is thrown when the maximum tree size has been exceeded.

getCostComplexityValues

```
public double[] getCostComplexityValues() throws
DecisionTree.PruningFailedToConvergeException,
PredictiveModel.StateChangeException
```

Description

Returns an array containing cost-complexity values.

Returns

a `double` array containing the cost-complexity values

The cost-complexity values are found via the optimal pruning algorithm of Breiman, et. al.

Exceptions

`PruningFailedToConvergeException` pruning has failed to converge.

`PredictiveModel.StateChangeException` an input parameter has changed that might affect the model estimates or predictions.

getDecisionTree

```
public Tree getDecisionTree() throws PredictiveModel.StateChangeException
```

Description

Returns a `Tree` object.

Returns

a `Tree` object containing the tree structure information

Exception

`PredictiveModel.StateChangeException` an input parameter has changed that might affect the model estimates or predictions.

getFittedMeanSquaredError

```
public double getFittedMeanSquaredError() throws
PredictiveModel.StateChangeException
```

Description

Returns the mean squared error on the training data.

Returns

a `double` equal to the mean squared error between the fitted value and the actual value of the response variable in the training data

Exception

`PredictiveModel.StateChangeException` an input parameter has changed that might affect the model estimates or predictions.

getMaxDepth

```
public int getMaxDepth()
```

Description

Returns the maximum depth a tree is allowed to have.

Returns

an `int`, the maximum depth a tree is allowed to have

getMaxNodes

```
public int getMaxNodes()
```

Description

Returns the maximum number of `com.imsl.datamining.decisionTree.TreeNode` (p. 2229) instances allowed in a tree.

Returns

an `int`, the maximum number of nodes allowed in a tree

getMeanSquaredPredictionError

```
public double getMeanSquaredPredictionError() throws
PredictiveModel.StateChangeException
```

Description

Returns the mean squared error.

Returns

a `double` equal to the mean squared error between the predicted value and the actual value of the response variable. The error is the in-sample fitted error if `predict` is first called with no arguments. Otherwise, the error is relative to the test data provided in the call to `predict`.

Exception

`PredictiveModel.StateChangeException` an input parameter has changed that might affect the model estimates or predictions.

getMinObsPerChildNode

```
public int getMinObsPerChildNode()
```

Description

Returns the minimum number of observations that are required for any child node before performing a split.

Returns

an `int`, the minimum number of observations that are required for any child node before performing a split

getMinObsPerNode

```
public int getMinObsPerNode()
```

Description

Returns the minimum number of observations that are required in a node before performing a split.

Returns

an `int` indicating the minimum number of observations that are required in a node before performing a split.

getNodeAssigments

```
public int[] getNodeAssigments(double[][] testData)
```

Description

Returns the terminal node assignments for each row of the test data.

Parameter

`testData` – a `double` matrix containing the test data

`testData` must have the same column structure and type as the training data.

Returns

an `int` array containing the (0-based) terminal node id's for each observation in `testData`

getNumberOfComplexityValues

```
public int getNumberOfComplexityValues()
```

Description

Returns the number of cost complexity values determined by the pruning algorithm.

Returns

an `int`, the number of cost complexity values

getNumberOfRandomFeatures

```
public int getNumberOfRandomFeatures()
```

Description

Returns the number of random features used in the splitting rules when `randomFeatureSelection=true`.

Returns

an `int`, the number of random features

isAutoPruningFlag

```
public boolean isAutoPruningFlag()
```

Description

Returns the current setting of the boolean to automatically prune the decision tree.

Returns

a `boolean`, the value of the auto-pruning flag. If `true`, the model is configured to automatically prune the decision tree. If the flag is `false`, no pruning is performed.

isRandomFeatureSelection

```
public boolean isRandomFeatureSelection()
```

Description

Returns the current setting of the boolean to perform random feature selection.

Returns

a `boolean`, the value of the flag. If `true`, the set of variables considered at each node is randomly selected. If the flag is `false`, all variables are considered at each node.

predict

```
public double[] predict() throws PredictiveModel.StateChangeException
```

Description

Predicts the training examples (in-sample predictions) using the most recently grown tree.

Returns

a `double` array containing fitted values of the response variable using the most recently grown decision tree. To populate fitted values, use the `predict` method without arguments.

Exception

`PredictiveModel.StateChangeException` is thrown when an input parameter changes that might affect the model estimates or predictions.

predict

`public double[] predict(double[][] testData) throws PredictiveModel.StateChangeException`

Description

Predicts new data using the most recently grown decision tree.

Parameter

`testData` – a `double` matrix containing test data for which predictions are to be made using the current tree. `testData` must have the same number of columns in the same arrangement as `xy`.

Returns

a `double` array containing predicted values

Exception

`PredictiveModel.StateChangeException` is thrown when an input parameter changes that might affect the model estimates or predictions.

predict

`public double[] predict(double[][] testData, double[] testDataWeights) throws PredictiveModel.StateChangeException`

Description

Predicts new weighted data using the most recently grown decision tree.

Parameters

`testData` – a `double` matrix containing test data for which predictions are to be made using the current tree. `testData` must have the same number of columns in the same arrangement as `xy`.

`testDataWeights` – a `double` array containing weights for each row of `testData`

Returns

a `double` array containing predicted values

Exception

`PredictiveModel.StateChangeException` is thrown when an input parameter changes that might affect the model estimates or predictions.

printDecisionTree

`public void printDecisionTree(boolean printMaxTree)`

Description

Prints the contents of the decision tree using distinct but general labels.

This method uses default values for the variable labels when printing (see printDecisionTree (String, String[], String[], String[], boolean) for these values.)

Parameter

`printMaxTree` – a `boolean` indicating that the maximal tree should be printed. When `true`, the maximal tree is printed. Otherwise, the pruned tree is printed.

printDecisionTree

```
public void printDecisionTree(String responseName, String[] predictorNames,
String[] classNames, String[] categoryNames, boolean printMaxTree)
```

Description

Prints the contents of the decision tree using labels.

Parameters

`responseName` – a `String` specifying a name for the response variable

If `null`, the default value is used.

Default: `responseName` = Y

`predictorNames` – a `String` array specifying names for the predictor variables

If `null`, the default value is used.

Default: `predictorNames` = X0, X1, ...

`classNames` – a `String` array specifying names for the class levels

If `null`, the default value is used.

Default: `classNames` = 0, 1, 2, ...

`categoryNames` – a `String` array specifying names for the categories of the predictor variables

If `null`, the default value is used.

Default: `categoryNames` = 0, 1, 2, ...

`printMaxTree` – a `boolean` indicating that the maximal tree should be printed. When `true`, the maximal tree is printed. Otherwise, the pruned tree is printed.

pruneTree

```
public void pruneTree(double gamma)
```

Description

Finds the minimum cost-complexity decision tree for the cost-complexity value, gamma.

The method implements the optimal pruning algorithm 10.1, page 294 in Breiman, et. al (1984). The result of the algorithm is a sequence of sub-trees $T_{\max} \succ T_1 \succ T_2 \succ T_{M-1} \succ \{t_0\}$ obtained by pruning the fully generated tree, $T_{\max}$, until the sub-tree consists of the single root node, $\{t_0\}$. Corresponding to the sequence of sub-trees is the sequence of complexity values, $0 \le \delta_{\min} < \delta_1 < \delta_2 < \ldots < \delta_{M-1} < \delta_M$ where M is the number of steps it takes in the algorithm to reach the root node. The sub-trees represent the optimally pruned sub-trees for the sequence of complexity values.

The effect of the pruning is stored in the tree's terminal node array. That is, when the algorithm determines that the tree should be pruned at a particular node, it sets that node to be a terminal node using the method `com.imsl.datamining.decisionTree.Tree.setTerminalNode` (p. **??**) .

No other changes are made to the tree structure so that the maximal tree can still be printed and reviewed. However, once a tree is pruned, all the predictions will use the pruned tree.

Parameter

`gamma` – a `double` giving the value of the cost-complexity parameter

selectSplitVariable

`abstract protected int selectSplitVariable(double[][] xy, double[] classCounts, double[] parentFreq, double[] splitValue, double[] splitCriterionValue, int[] splitPartition)`

Description

Abstract method for selecting the next split variable and split definition for the node.

Parameters

`xy` – a `double` matrix containing the data

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical

`parentFreq` – a `double` array used to indicate which subset of the observations belong in the current node

`splitValue` – a `double` array representing the resulting split point if the selected variable is quantitative

`splitCriterionValue` – a `double`, the value of the criterion used to determine the splitting variable

`splitPartition` – an `int` array indicating the resulting split partition if the selected variable is categorical

Returns

an `int` specifying the index of the split variable in `this.getPredictorIndexes()`

setAutoPruningFlag

`public void setAutoPruningFlag(boolean autoPruningFlag)`

Description

Sets the flag to automatically prune the tree during the fitting procedure.

The default value is `false`. Set to `true` before calling `com.imsl.datamining.decisionTree.DecisionTree.fitModel` (p. 2240) in order to prune the tree automatically. The pruning will use the cost-complexity value equal to `minCostComplexityValue`. See also `com.imsl.datamining.decisionTree.DecisionTree.pruneTree` (p. 2245) which prunes the tree using a given cost-complexity value.

Parameter

`autoPruningFlag` – a `boolean`, specifying the value of the flag. If `true`, the maximally grown tree should be automatically pruned in `com.imsl.datamining.decisionTree.DecisionTree.fitModel` (p. 2240)

Default: `autoPruningFlag=false`.

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
DecisionTree.PruningFailedToConvergeException,
PredictiveModel.StateChangeException,
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Sets the configuration of `PredictiveModel` to that of the input model.

Parameter

`pm` – a `PredictiveModel` object

Exceptions

`PruningFailedToConvergeException` is thrown when pruning fails to converge.

`PredictiveModel.StateChangeException` is thrown when an input parameter has changed that might affect the model estimates or predictions.

`com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException` is thrown when the sum of the probabilities does not equal 1.

setCostComplexityValues

```
public void setCostComplexityValues(double[] gammas)
```

Description

Sets the cost-complexity values. For the original tree, the values are generated in `com.imsl.datamining.decisionTree.DecisionTree.fitModel` (p. 2240) when `com.imsl.datamining.decisionTree.DecisionTree.isAutoPruningFlag` (p. 2243) returns `true`.

Parameter

`gammas` – `double` array containing cost-complexity values. This method is used when copying the configuration of one tree to another.
Default: `gammas` = `com.imsl.datamining.decisionTree.DecisionTree.setMinCostComplexityValue` (p. 2248) .

setMaxDepth

```
public void setMaxDepth(int nLevels)
```

Description

Sets the maximum tree depth allowed.

Parameter

`nLevels` – an `int` specifying the maximum depth that the `DecisionTree` is allowed to have. `nLevels` should be strictly positive.
Default: `nLevels` = 10.

setMaxNodes

```
public void setMaxNodes(int maxNodes)
```

Description

Sets the maximum number of nodes allowed in a tree.

Parameter

`maxNodes` – an `int` specifying the maximum number of nodes allowed in a tree

Default: `maxNodes` = 100.

setMinCostComplexityValue

```
public void setMinCostComplexityValue(double minCostComplexity)
```

Description

Sets the value of the minimum cost-complexity value.

Parameter

`minCostComplexity` – a `double` indicating the smallest value to use in cost-complexity pruning. The value must be in [0.0, 1.0].

Default: `minCostComplexity` = 0.

setMinObsPerChildNode

```
public void setMinObsPerChildNode(int nObs)
```

Description

Sets the minimum number of observations that a child node must have in order to split.

Parameter

`nObs` – an `int` specifying the minimum number of observations that a child node must have in order to split the current node. `nObs` must be strictly positive. `nObs` must also be greater than the minimum number of observations required before a node can split `com.imsl.datamining.decisionTree.DecisionTree.setMinObsPerNode` (p. 2248) .

Default: `nObs` = 7.

setMinObsPerNode

```
public void setMinObsPerNode(int nObs)
```

Description

Sets the minimum number of observations a node must have to allow a split.

Parameter

`nObs` – an `int` specifying the number of observations the current node must have before considering a split. `nObs` should be greater than 1 but less than or equal to the number of observations in `xy`.

Default: `nObs` = 21.

setNumberOfRandomFeatures

```
public void setNumberOfRandomFeatures(int numberOfRandomFeatures)
```

Description

Sets the number of predictors in the random subset to select from at each node.

Parameter

`numberOfRandomFeatures` – an `int`, the number of predictors in the random subset
Default: numberOfFeatures=$\sqrt(p)$ for classification problems, and $p/3$ for regression problems, where p is the number of predictors in the training data

setRandomFeatureSelection

`public void setRandomFeatureSelection(boolean selectRandomFeatures)`

Description

Sets the flag to select split variables from a random subset of the features.

Parameter

`selectRandomFeatures` – a `boolean`, indicating whether or not to select random features
Default: selectRandomFeatures`false`

Example: ALACART and C45

In this example, we use a small data set with response variable, Play, which indicates whether a golfer plays (1) or does not play (0) golf under weather conditions measured by Temperature, Humidity, Outlook (Sunny (0), Overcast (1), Rainy (2)), and Wind (True (0), False (1)). A decision tree is generated by `C45` and the `ALACART` class. The control parameters are adjusted because of the small data size and no cross-validation or pruning is performed. The maximal trees are printed out using `DecisionTree.printDecisionTree`. Notice that `C45` splits on Outlook, then Humidity and Wind, while `ALACART` splits on Outlook, then Temperature.

```
import com.imsl.datamining.decisionTree.*;

public class C45ALACART {

    public static void main(String[] args) throws Exception {

        int golfResponseIdx = 4;
        double[][] golfXY = {
            {0, 85, 85, 0, 0}, {0, 80, 90, 1, 0}, {1, 83, 78, 0, 1},
            {2, 70, 96, 0, 1}, {2, 68, 80, 0, 1}, {2, 65, 70, 1, 0},
            {1, 64, 65, 1, 1}, {0, 72, 95, 0, 0}, {0, 69, 70, 0, 1},
            {2, 75, 80, 0, 1}, {0, 75, 70, 1, 1}, {1, 72, 90, 1, 1},
            {1, 81, 75, 0, 1}, {2, 71, 80, 1, 0}
        };

        DecisionTree.VariableType[] golfVarType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        String[] names = {
            "Outlook", "Temperature", "Humidity", "Wind", "Play"
        };
```

```
        String[] classNames = {"Don't Play", "Play"};
        String[] varLevels = {"Sunny", "Overcast", "Rainy", "False", "True"};

        C45 dt = new C45(golfXY, golfResponseIdx, golfVarType);
        dt.setMinObsPerChildNode(2);
        dt.setMinObsPerNode(3);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\n\nDecision Tree using Method C4.5:");
        dt.printDecisionTree(null, names, classNames,
                varLevels, true);

        ALACART adt = new ALACART(golfXY, golfResponseIdx, golfVarType);
        adt.setMinObsPerChildNode(2);
        adt.setMinObsPerNode(3);
        adt.setMaxNodes(50);
        adt.fitModel();

        System.out.println("\n\nDecision Tree using Method ALACART:");
        adt.printDecisionTree(null, names, classNames,
                varLevels, true);
    }
}
```

Output

```
Decision Tree using Method C4.5:

Decision Tree:


Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  4  5
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play

Node 1: Cost = 0.143, N= 5, Level = 1, Child nodes:  2  3
   Rule:  Outlook  in: { Sunny }
    P(Y=0)= 0.600
    P(Y=1)= 0.400
    Predicted Y:  Don't Play

Node 2: Cost = 0.000, N= 2, Level = 2
      Rule:  Humidity       <= 77.500
        P(Y=0)= 0.000
        P(Y=1)= 1.000
        Predicted Y:  Play

Node 3: Cost = 0.000, N= 3, Level = 2
      Rule:  Humidity       > 77.500
        P(Y=0)= 1.000
        P(Y=1)= 0.000
```

```
        Predicted Y:  Don't Play

Node 4: Cost = 0.000, N= 4, Level = 1
   Rule:  Outlook  in: { Overcast }
    P(Y=0)= 0.000
    P(Y=1)= 1.000
    Predicted Y:  Play

Node 5: Cost = 0.143, N= 5, Level = 1, Child nodes:  6  7
   Rule:  Outlook  in: { Rainy }
    P(Y=0)= 0.400
    P(Y=1)= 0.600
    Predicted Y:  Play

Node 6: Cost = 0.000, N= 3, Level = 2
      Rule:  Wind  in: { False }
        P(Y=0)= 0.000
        P(Y=1)= 1.000
        Predicted Y:  Play

Node 7: Cost = 0.000, N= 2, Level = 2
      Rule:  Wind  in: { True }
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play


Decision Tree using Method ALACART:

Decision Tree:


Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  8
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play

Node 1: Cost = 0.357, N= 10, Level = 1, Child nodes:  2  7
   Rule:  Outlook  in: { Sunny  Rainy }
    P(Y=0)= 0.500
    P(Y=1)= 0.500
    Predicted Y:  Don't Play

Node 2: Cost = 0.214, N= 8, Level = 2, Child nodes:  3  6
      Rule:  Temperature       <= 77.500
        P(Y=0)= 0.375
        P(Y=1)= 0.625
        Predicted Y:  Play

Node 3: Cost = 0.214, N= 6, Level = 3, Child nodes:  4  5
         Rule:  Temperature          <= 73.500
            P(Y=0)= 0.500
            P(Y=1)= 0.500
            Predicted Y:  Don't Play

Node 4: Cost = 0.071, N= 4, Level = 4
```

```
            Rule:  Temperature             <= 70.500
                P(Y=0)= 0.250
                P(Y=1)= 0.750
                Predicted Y:  Play

Node 5: Cost = 0.000, N= 2, Level = 4
            Rule:  Temperature             > 70.500
                P(Y=0)= 1.000
                P(Y=1)= 0.000
                Predicted Y:  Don't Play

Node 6: Cost = 0.000, N= 2, Level = 3
         Rule:  Temperature          > 73.500
            P(Y=0)= 0.000
            P(Y=1)= 1.000
            Predicted Y:  Play

Node 7: Cost = 0.000, N= 2, Level = 2
      Rule:  Temperature       > 77.500
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play

Node 8: Cost = 0.000, N= 4, Level = 1
   Rule:  Outlook  in: { Overcast }
    P(Y=0)= 0.000
    P(Y=1)= 1.000
    Predicted Y:  Play
```

Example: C45 Simulated Categorical Data

This example uses the C45 method on simulated categorical data and demonstrates printing the tree structure with and without custom labels.

```
import com.imsl.datamining.decisionTree.*;

public class C45SimulatedCategoricalData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {2, 0, 2}, {1, 0, 0}, {2, 1, 3}, {0, 1, 0}, {1, 2, 0}, {2, 2, 3},
            {2, 2, 3}, {0, 1, 0}, {0, 0, 0}, {0, 1, 0}, {1, 2, 0}, {2, 0, 2},
            {0, 2, 0}, {2, 0, 1}, {0, 0, 0}, {2, 0, 1}, {1, 0, 0}, {0, 2, 0},
            {2, 0, 1}, {1, 2, 0}, {0, 2, 2}, {2, 1, 3}, {1, 1, 0}, {2, 2, 3},
            {1, 2, 0}, {2, 2, 3}, {2, 0, 1}, {2, 1, 3}, {1, 2, 0}, {1, 1, 0}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };
```

```
        String responseName = "Response";
        String[] names = {"Var1", "Var2"};
        String[] classNames = {"c1", "c2", "c3", "c4"};
        String[] varLabels = {"L1", "L2", "L3", "A", "B", "C"};

        C45 dt = new C45(xy, 2, varType);
        dt.setMinObsPerChildNode(5);
        dt.setMinObsPerNode(10);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\nGenerated labels:");
        dt.printDecisionTree(true);
        System.out.println("\nCustom labels:");
        dt.printDecisionTree(responseName, names,
                classNames, varLabels, false);
    }
}
```

Output

```
Generated labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2  3
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Y:   0

Node 1: Cost = 0.033, N= 8, Level = 1
   Rule: X0  in: { 0 }
    P(Y=0)= 0.875
    P(Y=1)= 0.000
    P(Y=2)= 0.125
    P(Y=3)= 0.000
    Predicted Y:   0

Node 2: Cost = 0.000, N= 9, Level = 1
   Rule: X0  in: { 1 }
    P(Y=0)= 1.000
    P(Y=1)= 0.000
    P(Y=2)= 0.000
    P(Y=3)= 0.000
    Predicted Y:   0

Node 3: Cost = 0.200, N= 13, Level = 1
   Rule: X0  in: { 2 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
    P(Y=2)= 0.154
```

```
    P(Y=3)= 0.538
    Predicted Y:   3

Custom labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2  3
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Response:  c1

Node 1: Cost = 0.033, N= 8, Level = 1
   Rule:  Var1  in: { L1 }
    P(Y=0)= 0.875
    P(Y=1)= 0.000
    P(Y=2)= 0.125
    P(Y=3)= 0.000
    Predicted Response:  c1

Node 2: Cost = 0.000, N= 9, Level = 1
   Rule:  Var1  in: { L2 }
    P(Y=0)= 1.000
    P(Y=1)= 0.000
    P(Y=2)= 0.000
    P(Y=3)= 0.000
    Predicted Response:  c1

Node 3: Cost = 0.200, N= 13, Level = 1
   Rule:  Var1  in: { L3 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Response:  c4
```

Example: CHAID Simulated Categorical Data

This example uses the CHAID method on simulated categorical data and demonstrates printing the tree structure with and without custom labels.

```
import com.imsl.datamining.decisionTree.*;

public class CHAIDSimulatedCategoricalData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {2, 0, 2}, {1, 0, 0}, {2, 1, 3}, {0, 1, 0}, {1, 2, 0}, {2, 2, 3},
            {2, 2, 3}, {0, 1, 0}, {0, 0, 0}, {0, 1, 0}, {1, 2, 0}, {2, 0, 2},
            {0, 2, 0}, {2, 0, 1}, {0, 0, 0}, {2, 0, 1}, {1, 0, 0}, {0, 2, 0},
```

```
            {2, 0, 1}, {1, 2, 0}, {0, 2, 2}, {2, 1, 3}, {1, 1, 0}, {2, 2, 3},
            {1, 2, 0}, {2, 2, 3}, {2, 0, 1}, {2, 1, 3}, {1, 2, 0}, {1, 1, 0}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        String responseName = "Response";
        String[] names = {"Var1", "Var2"};
        String[] classNames = {"c1", "c2", "c3", "c4"};
        String[] varLabels = {"L1", "L2", "L3", "A", "B", "C"};

        CHAID dt = new CHAID(xy, 2, varType);
        dt.setMinObsPerChildNode(5);
        dt.setMinObsPerNode(10);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\nGenerated labels:");
        dt.printDecisionTree(true);
        System.out.println("\nCustom labels:");
        dt.printDecisionTree(responseName, names,
                classNames, varLabels, false);
    }
}
```

Output

```
Generated labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Y:   0

Node 1: Cost = 0.033, N= 17, Level = 1
   Rule: X0  in: { 0  1 }
    P(Y=0)= 0.941
    P(Y=1)= 0.000
    P(Y=2)= 0.059
    P(Y=3)= 0.000
    Predicted Y:   0

Node 2: Cost = 0.200, N= 13, Level = 1, Child nodes:  3  4
   Rule: X0  in: { 2 }
    P(Y=0)= 0.000
```

```
    P(Y=1)= 0.308
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Y:   3

Node 3: Cost = 0.067, N= 6, Level = 2
      Rule: X1  in: { 0 }
        P(Y=0)= 0.000
        P(Y=1)= 0.667
        P(Y=2)= 0.333
        P(Y=3)= 0.000
        Predicted Y:   1

Node 4: Cost = 0.000, N= 7, Level = 2
      Rule: X1  in: { 1  2 }
        P(Y=0)= 0.000
        P(Y=1)= 0.000
        P(Y=2)= 0.000
        P(Y=3)= 1.000
        Predicted Y:   3

Custom labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Response:  c1

Node 1: Cost = 0.033, N= 17, Level = 1
   Rule:  Var1  in: { L1  L2 }
    P(Y=0)= 0.941
    P(Y=1)= 0.000
    P(Y=2)= 0.059
    P(Y=3)= 0.000
    Predicted Response:  c1

Node 2: Cost = 0.200, N= 13, Level = 1, Child nodes:  3  4
   Rule:  Var1  in: { L3 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Response:  c4

Node 3: Cost = 0.067, N= 6, Level = 2
      Rule:  Var2  in: { A }
        P(Y=0)= 0.000
        P(Y=1)= 0.667
        P(Y=2)= 0.333
        P(Y=3)= 0.000
        Predicted Response:  c2
```

```
Node 4: Cost = 0.000, N= 7, Level = 2
      Rule:  Var2  in: { B  C }
        P(Y=0)= 0.000
        P(Y=1)= 0.000
        P(Y=2)= 0.000
        P(Y=3)= 1.000
        Predicted Response:  c4
```

Example: QUEST Simulated Categorical Data

This example applies the QUEST method to a simulated data set with 50 cases and three predictors of mixed-type. Shown are the maximum-sized tree under default controls and the subtree resulting from pruning with a cost-complexity value of 0.0.

```
import com.imsl.datamining.decisionTree.*;

public class QUESTSimulatedCategoricalData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {2, 25.928690, 0, 0}, {1, 51.632450, 1, 1}, {1, 25.784321, 0, 2},
            {0, 39.379478, 0, 3}, {2, 24.650579, 0, 2}, {2, 45.200840, 0, 2},
            {2, 52.679600, 1, 3}, {1, 44.283421, 1, 3}, {2, 40.635231, 1, 3},
            {2, 51.760941, 0, 3}, {2, 26.303680, 0, 1}, {2, 20.702299, 1, 0},
            {2, 38.742729, 1, 3}, {2, 19.473330, 0, 0}, {1, 26.422110, 0, 0},
            {2, 37.059860, 1, 0}, {1, 51.670429, 1, 3}, {0, 42.401562, 0, 3},
            {2, 33.900269, 1, 2}, {1, 35.432819, 0, 0}, {1, 44.303692, 0, 1},
            {0, 46.723869, 0, 2}, {1, 46.992619, 0, 2}, {0, 36.059231, 0, 3},
            {2, 36.831970, 1, 1}, {1, 61.662571, 1, 2}, {0, 25.677139, 0, 3},
            {1, 39.085670, 1, 0}, {0, 48.843410, 1, 1}, {1, 39.343910, 0, 3},
            {2, 24.735220, 0, 2}, {1, 50.552509, 1, 3}, {0, 31.342630, 1, 3},
            {1, 27.157949, 1, 0}, {0, 31.726851, 0, 2}, {0, 25.004080, 0, 3},
            {1, 26.354570, 1, 3}, {2, 38.123428, 0, 1}, {0, 49.940300, 0, 2},
            {1, 42.457790, 1, 3}, {0, 38.809479, 1, 1}, {0, 43.227989, 1, 1},
            {0, 41.876240, 0, 3}, {2, 48.078201, 0, 2}, {0, 43.236729, 1, 0},
            {2, 39.412941, 0, 3}, {1, 23.933460, 0, 2}, {2, 42.841301, 1, 3},
            {2, 30.406691, 0, 1}, {0, 37.773891, 0, 2}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        QUEST dt = new QUEST(xy, 3, varType);
        dt.setPrintLevel(1);
        dt.fitModel();
        dt.pruneTree(0.0);

        System.out.println("\nMaximal tree: \n");
```

```
        dt.printDecisionTree(true);

        System.out.println("\nPruned subtree (cost-complexity = 0): \n");
        dt.printDecisionTree(false);
    }
}
```

Output

```
Growing the maximal tree using method QUEST:


Maximal tree:


Decision Tree:


Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes:  1  2
P(Y=0)= 0.180
P(Y=1)= 0.180
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y:   3

Node 1: Cost = 0.220, N= 17, Level = 1
   Rule: X1    <= 35.031
    P(Y=0)= 0.294
    P(Y=1)= 0.118
    P(Y=2)= 0.353
    P(Y=3)= 0.235
    Predicted Y:   2

Node 2: Cost = 0.360, N= 33, Level = 1, Child nodes:  3  4
   Rule: X1    > 35.031
    P(Y=0)= 0.121
    P(Y=1)= 0.212
    P(Y=2)= 0.212
    P(Y=3)= 0.455
    Predicted Y:   3

Node 3: Cost = 0.180, N= 19, Level = 2
      Rule: X1       <= 43.265
        P(Y=0)= 0.211
        P(Y=1)= 0.211
        P(Y=2)= 0.053
        P(Y=3)= 0.526
        Predicted Y:   3

Node 4: Cost = 0.160, N= 14, Level = 2
      Rule: X1       > 43.265
        P(Y=0)= 0.000
        P(Y=1)= 0.214
        P(Y=2)= 0.429
        P(Y=3)= 0.357
```

```
        Predicted Y:   2

Pruned subtree (cost-complexity = 0):


Decision Tree:


Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes:  1  2
P(Y=0)= 0.180
P(Y=1)= 0.180
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y:   3

Node 1: Cost = 0.220, N= 17, Level = 1
   Rule: X1    <= 35.031
    P(Y=0)= 0.294
    P(Y=1)= 0.118
    P(Y=2)= 0.353
    P(Y=3)= 0.235
    Predicted Y:   2

Node 2: Cost = 0.360, N= 33, Level = 1
   Rule: X1    > 35.031
    P(Y=0)= 0.121
    P(Y=1)= 0.212
    P(Y=2)= 0.212
    P(Y=3)= 0.455
    Predicted Y:   3
Pruned at Node id 2.
```

Example: QUEST Kyphosis Data

This example uses the dataset Kyphosis. The 81 cases represent 81 children who have undergone surgery to correct a type of spinal deformity known as Kyphosis. The response variable is the presence or absence of Kyphosis after the surgery. Three predictors are Age of the patient in months, Start, the number of the vertebra where the surgery started, and Number, the number of vertebra involved in the surgery. This example uses the method QUEST to produce a maximal tree. It also requests predictions for a test-data set consisting of 10 “new” cases.

```
import com.imsl.datamining.decisionTree.*;

public class QUESTKyphosisData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {0, 71, 3, 5}, {0, 158, 3, 14}, {1, 128, 4, 5}, {0, 2, 5, 1},
            {0, 1, 4, 15}, {0, 1, 2, 16}, {0, 61, 2, 17}, {0, 37, 3, 16},
            {0, 113, 2, 16}, {1, 59, 6, 12}, {1, 82, 5, 14}, {0, 148, 3, 16},
            {0, 18, 5, 2}, {0, 1, 4, 12}, {0, 168, 3, 18}, {0, 1, 3, 16},
            {0, 78, 6, 15}, {0, 175, 5, 13}, {0, 80, 5, 16},
```

```
    {0, 27, 4, 9}, {0, 22, 2, 16}, {1, 105, 6, 5}, {1, 96, 3, 12},
    {0, 131, 2, 3}, {1, 15, 7, 2}, {0, 9, 5, 13}, {0, 8, 3, 6},
    {0, 100, 3, 14}, {0, 4, 3, 16}, {0, 151, 2, 16}, {0, 31, 3, 16},
    {0, 125, 2, 11}, {0, 130, 5, 13}, {0, 112, 3, 16}, {0, 140, 5, 11},
    {0, 93, 3, 16}, {0, 1, 3, 9}, {1, 52, 5, 6}, {0, 20, 6, 9},
    {1, 91, 5, 12}, {1, 73, 5, 1}, {0, 35, 3, 13}, {0, 143, 9, 3},
    {0, 61, 4, 1}, {0, 97, 3, 16}, {1, 139, 3, 10}, {0, 136, 4, 15},
    {0, 131, 5, 13}, {1, 121, 3, 3}, {0, 177, 2, 14}, {0, 68, 5, 10},
    {0, 9, 2, 17}, {1, 139, 10, 6}, {0, 2, 2, 17}, {0, 140, 4, 15},
    {0, 72, 5, 15}, {0, 2, 3, 13}, {1, 120, 5, 8}, {0, 51, 7, 9},
    {0, 102, 3, 13}, {1, 130, 4, 1}, {1, 114, 7, 8}, {0, 81, 4, 1},
    {0, 118, 3, 16}, {0, 118, 4, 16}, {0, 17, 4, 10}, {0, 195, 2, 17},
    {0, 159, 4, 13}, {0, 18, 4, 11}, {0, 15, 5, 16}, {0, 158, 5, 14},
    {0, 127, 4, 12}, {0, 87, 4, 16}, {0, 206, 4, 10}, {0, 11, 3, 15},
    {0, 178, 4, 15}, {1, 157, 3, 13}, {0, 26, 7, 13}, {0, 120, 2, 13},
    {1, 42, 7, 6}, {0, 36, 4, 13}
};

double[][] xyTest = {
    {0, 71, 3, 5}, {1, 128, 4, 5}, {0, 1, 4, 15}, {0, 61, 6, 10},
    {0, 113, 2, 16}, {1, 82, 5, 14}, {0, 148, 3, 16}, {0, 1, 4, 12},
    {0, 1, 3, 16}, {0, 175, 5, 13}
};

DecisionTree.VariableType[] varType = {
    DecisionTree.VariableType.CATEGORICAL,
    DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
    DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
    DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS
};

String[] names = {"Age", "Number", "Start"};
String[] classNames = {"Absent", "Present"};
String responseName = "Kyphosis";

QUEST dt = new QUEST(xy, 0, varType);
dt.setMinObsPerChildNode(5);
dt.setMinObsPerNode(10);
dt.setMaxNodes(50);
dt.setPrintLevel(2);
dt.fitModel();

double[] predictions = dt.predict(xyTest);
double predErrSS = dt.getMeanSquaredPredictionError();

dt.printDecisionTree(responseName, names,
        classNames, null, true);

System.out.println("\nPredictions for test data:");
System.out.printf("%5s%8s%7s%10s\n", names[0], names[1], names[2],
        responseName);

for (int i = 0; i < xyTest.length; i++) {
    System.out.printf("%5.0f%8.0f%7.0f", xyTest[i][1], xyTest[i][2],
            xyTest[i][3]);
    int idx = (int) predictions[i];
```

```
            System.out.printf("%10s\n", classNames[idx]);
        }
        System.out.printf("\nMean squared prediction error: %f\n", predErrSS);
    }
}
```

Output

```
Growing the maximal tree using method QUEST:

Node 0has split variable 2
Node 1has split variable 0
Node 2 is a terminal node. It has 7.0 cases--too few cases to split.
Node 3 is a terminal node. It has 6.0 cases--too few cases to split.
Node 4has split variable 2
Node 5 is a terminal node. It has 6.0 cases--too few cases to split.
Node 6has split variable 2
Node 7has split variable 0
Node 8has split variable 0
Node 8 is a terminal node. The split is too thin having count 2.0.
Node 9has split variable 1
Node 10 is a terminal node. It has 6.0 cases--too few cases to split.
Node 11 is a terminal node, because it is pure.
Node 11 is a terminal node. It has 7.0 cases--too few cases to split.
Node 12is a terminal node. Could not find a splitting variable.

Decision Tree:


Node 0: Cost = 0.210, N= 81, Level = 0, Child nodes:  1  4
P(Y=0)= 0.790
P(Y=1)= 0.210
Predicted Kyphosis:  Absent

Node 1: Cost = 0.074, N= 13, Level = 1, Child nodes:  2  3
Rule:  Start <= 5.155
P(Y=0)= 0.538
P(Y=1)= 0.462
Predicted Kyphosis:  Absent

Node 2: Cost = 0.025, N= 7, Level = 2
Rule:  Age <= 84.030
P(Y=0)= 0.714
P(Y=1)= 0.286
Predicted Kyphosis:  Absent

Node 3: Cost = 0.025, N= 6, Level = 2
Rule:  Age > 84.030
P(Y=0)= 0.333
P(Y=1)= 0.667
Predicted Kyphosis:  Present

Node 4: Cost = 0.136, N= 68, Level = 1, Child nodes:  5  6
Rule:  Start > 5.155
P(Y=0)= 0.838
```

```
P(Y=1)= 0.162
Predicted Kyphosis:  Absent

Node 5: Cost = 0.012, N= 6, Level = 2
Rule:  Start <= 8.862
P(Y=0)= 0.167
P(Y=1)= 0.833
Predicted Kyphosis:  Present

Node 6: Cost = 0.074, N= 62, Level = 2, Child nodes:  7  12
Rule:  Start > 8.862
P(Y=0)= 0.903
P(Y=1)= 0.097
Predicted Kyphosis:  Absent

Node 7: Cost = 0.062, N= 28, Level = 3, Child nodes:  8  9
Rule:  Start <= 13.092
P(Y=0)= 0.821
P(Y=1)= 0.179
Predicted Kyphosis:  Absent

Node 8: Cost = 0.025, N= 15, Level = 4
Rule:  Age <= 91.722
P(Y=0)= 0.867
P(Y=1)= 0.133
Predicted Kyphosis:  Absent

Node 9: Cost = 0.037, N= 13, Level = 4, Child nodes:  10  11
Rule:  Age > 91.722
P(Y=0)= 0.769
P(Y=1)= 0.231
Predicted Kyphosis:  Absent

Node 10: Cost = 0.037, N= 6, Level = 5
Rule:  Number <= 3.450
P(Y=0)= 0.500
P(Y=1)= 0.500
Predicted Kyphosis:  Absent

Node 11: Cost = 0.000, N= 7, Level = 5
Rule:  Number > 3.450
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Kyphosis:  Absent

Node 12: Cost = 0.012, N= 34, Level = 3
Rule:  Start > 13.092
P(Y=0)= 0.971
P(Y=1)= 0.029
Predicted Kyphosis:  Absent

Predictions for test data:
  Age  Number  Start  Kyphosis
   71       3      5    Absent
  128       4      5   Present
    1       4     15    Absent
```

```
  61       6      10    Absent
 113       2      16    Absent
  82       5      14    Absent
 148       3      16    Absent
   1       4      12    Absent
   1       3      16    Absent
 175       5      13    Absent

Mean squared prediction error: 0.100000
```

DecisionTree.PureNodeException class

```
static public class
com.imsl.datamining.decisionTree.DecisionTree.PureNodeException extends
com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Exception thrown when attempting to split a node that is already pure (response variable is constant).

Constructor

DecisionTree.PureNodeException

`public DecisionTree.PureNodeException(String message)`

Description

Constructs a `PureNodeException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameter

`message` – a `String` that contains the key of an error message in the resource bundle

DecisionTree.PruningFailedToConvergeException class

```
static public class
com.imsl.datamining.decisionTree.DecisionTree.PruningFailedToConvergeException
extends com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Exception thrown when pruning fails to converge.

Constructor

DecisionTree.PruningFailedToConvergeException

```
public DecisionTree.PruningFailedToConvergeException(String message)
```

Description

Constructs a `PruningFailedToConvergeException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameter

`message` – a `String` that contains the key of an error message in the resource bundle

DecisionTree.MaxTreeSizeExceededException class

```
static public class
com.imsl.datamining.decisionTree.DecisionTree.MaxTreeSizeExceededException
extends com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Exception thrown when the maximum tree size has been exceeded.

Constructors

DecisionTree.MaxTreeSizeExceededException

```
public DecisionTree.MaxTreeSizeExceededException(String message)
```

Description

Constructs a `MaxTreeSizeExceededException` and issues the specified message.

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

DecisionTree.MaxTreeSizeExceededException

```
public DecisionTree.MaxTreeSizeExceededException(String key, Object[]
arguments)
```

Description

Constructs a `MaxTreeSizeExceededException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameters

`key` – a `String` that contains the key of an error message in the resource bundle

`arguments` – an `Object` array containing arguments used within the error message specified by the `key`

DecisionTreeInfoGain class

```
abstract public class com.imsl.datamining.decisionTree.DecisionTreeInfoGain
extends com.imsl.datamining.decisionTree.DecisionTree implements Serializable,
Cloneable
```

Abstract class that extends `com.imsl.datamining.decisionTree.DecisionTree` (p. 2237) for classes that use an information gain criteria.

Constructor

DecisionTreeInfoGain

```
public DecisionTreeInfoGain(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a `DecisionTree` object for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix with rows containing the observations on the predictor variables and one response variable

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

information

```
protected double information(int[] x, int[] y, double[] classCounts, double[]
weights, boolean xInfo)
```

Description

Returns the expected information of a variable `y` over a partition determined by the variable `x`.

Given a data subset S containing both variables x and y, let

$$S_1, S_2, .., S_k, \cup S_i = S, S_i \cap S_j = \emptyset$$

be a partition of S determined by the values in x. Then the expected information is

$$\sum_j Pr(S_j) I(S_j)$$

where $I(S_j)$ is either the Shannon entropy or the Gini index, according to `com.imsl.datamining.decisionTree.DecisionTreeInfoGain.GainCriteria` (p. 2268) . Note: if x is constant, the return value is the Shannon Entropy (or Gini index) of Y.

Parameters

`x` – an `int` array of length `xy.length` containing values of a predictor or an indicator vector defining the partition of the observations.

`y` – `int` array of length `xy.length` containing the values of the response variable.

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical.

`weights` – a `double` array used to indicate which subset of the observations belong in the current node.

`xInfo` – a `boolean` indicating that we are getting information about `x` using a simple frequency estimate.

Value	Method
`true`	simple frequency estimate
`false`	prior probabilities

Returns

a `double` indicating the information uncertainty.

selectSplitVariable

```
abstract protected int selectSplitVariable(double[][] xy, double[] classCounts,
double[] parentFreq, double[] splitValue, double[] splitCriterionValue, int[]
splitPartition)
```

Description

Abstract method for selecting the next split variable and split definition for the node.

Parameters

`xy` – a `double` matrix containing the data

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical

`parentFreq` – a `double` array used to indicate which subset of the observations belong in the current node

`splitValue` – a `double` array representing the resulting split point if the selected variable is quantitative

`splitCriterionValue` – a `double`, the value of the criterion used to determine the splitting variable

`splitPartition` – an `int` array indicating the resulting split partition if the selected variable is categorical

Returns

an `int` specifying the column index of the split variable in `this.getPredictorIndexes`

setGainCriteria

`public void setGainCriteria(DecisionTreeInfoGain.GainCriteria gainCriteria)`

Description

Specifies which criteria to use in gain calculations in order to determine the best split at each node.

Parameter

`gainCriteria` – a `com.imsl.datamining.decisionTree.DecisionTreeInfoGain.GainCriteria` (p. 2268) specifying which criteria to use in gain calculations in order to determine the best split at each node

Default: `gainCriteria` = `com.imsl.datamining.decisionTree.DecisionTreeInfoGain.GainCriteria.SHANNON_ENTROPY` (p. 2268)

setUseRatio

`public void setUseRatio(boolean ratio)`

Description

Sets the flag to use or not use the gain ratio instead of the gain to determine the best split.

Parameter

`ratio` – a `boolean` indicating if the gain ratio is to be used

`true` uses the gain ratio; `false` uses the gain.

Default: `useRatio=false`

useGainRatio

`public boolean useGainRatio()`

Description

Returns whether or not the gain ratio is to be used instead of the gain to determine the best split.

Returns

a `boolean` indicating if the gain ratio is to be used

`true`, uses the gain ratio; `false` uses the gain.

DecisionTreeInfoGain.GainCriteria class

```
static public final class
com.imsl.datamining.decisionTree.DecisionTreeInfoGain.GainCriteria extends
java.lang.Enum
```

Specifies which information gain criteria to use in determining the best split at each node.

Fields

DEVIANCE

```
static final public DecisionTreeInfoGain.GainCriteria DEVIANCE
```

A measure of the quality of fit.

For a categorical variable having C distinct values over a data set S, the Deviance measure is

$$\sum_{i=1}^{C} n_i \log(p_i)$$

where

$$p_i = Pr(Y = i)$$

and

$$n_i$$

is the number of cases with $Y = i$ on the node.

GINI_INDEX

```
static final public DecisionTreeInfoGain.GainCriteria GINI_INDEX
```

A measure of statistical dispersion.

For a categorical variable having C distinct values over a data set S, the Gini index is defined as

$$I(S) = \sum_{\substack{i,j=1 \\ i \neq j}}^{C} p(i|S) = 1 - \sum_{i=1}^{C} p^2(i|S)$$

where $p(i|S)$ denotes the probability that the variable is equal to the state i on the data set, S.

SHANNON_ENTROPY

```
static final public DecisionTreeInfoGain.GainCriteria SHANNON_ENTROPY
```

A measure of randomness or uncertainty.

For a categorical variable having C distinct values over a data set S, the Shannon Entropy is defined as

$$\sum_{i=1}^{C} p_i \log(p_i)$$

where

$$p_i = Pr(Y = i)$$

and where

$$p_i \log(p_i) := 0$$

if $p_i = 0$.

Methods

valueOf

```
static public DecisionTreeInfoGain.GainCriteria valueOf(String name)
```

values

```
static public DecisionTreeInfoGain.GainCriteria[] values()
```

ALACART class

```
public class com.imsl.datamining.decisionTree.ALACART extends
com.imsl.datamining.decisionTree.DecisionTreeInfoGain implements
com.imsl.datamining.decisionTree.DecisionTreeSurrogateMethod, Serializable,
Cloneable
```

Generates a decision tree using the CART$^{\text{TM}}$ method of Breiman, Friedman, Olshen and Stone (1984). CART$^{\text{TM}}$ stands for Classification and Regression Trees and applies to categorical or quantitative type variables.

Only binary splits are considered for categorical variables. That is, if X has values {A, B, C, D}, splits into only two subsets are considered, e.g., {A} and {B, C, D}, or {A, B} and {C, D}, are allowed, but a three-way split defined by {A}, {B} and {C,D} is not.

For classification problems, `ALACART` uses a similar criterion to information gain called *impurity*. The method searches for a split that reduces the node impurity the most. For a given set of data S at a node, the node impurity for a C-class categorical response is a function of the class probabilities.

$$I(S) = \phi(p(1|S), p(2|S), \ldots, p(C|S))$$

The measure function $\phi(\cdot)$ should be 0 for "pure" nodes, where all Y are in the same class, and maximum when Y is uniformly distributed across the classes.

As only binary splits of a subset S are considered (S_1, S_2 such that $S = S_1 \cup S_2$ and $S = S_1 \cap S_2 = \emptyset$), the reduction in impurity when splitting S into S_1, S_2 is

$$\Delta I = I(S) - q_1 I(S_1) - q_2 I(S_2)$$

where

$$q_j = Pr[S_j], j = 1, 2$$

is the node probability.

The gain criteria and the reduction in impurity ΔI are similar concepts and equivalent when I is entropy and when only binary splits are considered. Another popular measure for the impurity at a node is the *Gini* index, given by

$$I(S) = \sum_{\substack{i,j=1 \\ i \neq j}}^{C} p(i|S) = 1 - \sum_{i=1}^{C} p^2(i|S)$$

If Y is an ordered response or continuous, the problem is a regression problem. `ALACART` generates the tree using the same steps, except that node-level measures or loss-functions are the mean squared error (MSE) or mean absolute error (MAD) rather than node impurity measures.

Missing Values

Any observation or case with a missing response variable is eliminated from the analysis. If a predictor has a missing value, each algorithm skips that case when evaluating the given predictor. When making a prediction for a new case, if the split variable is missing, the prediction function applies *surrogate* split-variables and splitting rules in turn, if they are estimated with the decision tree. Otherwise, the prediction function returns the prediction from the most recent non-terminal node. In this implementation, only `ALACART` estimates surrogate split variables when requested.

Constructor

ALACART

```
public ALACART(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs an `ALACART` decision tree for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index in `xy` of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

addSurrogates

`public void addSurrogates(Tree tree, double[] surrogateInfo)`

Description

Adds the surrogate information to the tree.

Parameters

`tree` – a `Tree` containing the decision tree structure

`surrogateInfo` – a `double` array containing the surrogate split information

getNumberOfSurrogateSplits

`public int getNumberOfSurrogateSplits()`

Description

Returns the number of surrogate splits.

Returns

an `int`, the number of surrogate splits

getSurrogateInfo

`public double[] getSurrogateInfo()`

Description

Returns the surrogate split information.

Returns

a `double` array containing the surrogate split information

selectSplitVariable

`protected int selectSplitVariable(double[][] xy, double[] classCounts, double[] parentFreq, double[] splitValue, double[] splitCriterionValue, int[] splitPartition)`

Description

Selects the split variable for the present node using the CART^{TM} method.

Parameters

`xy` – a `double` matrix containing the data

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical

`parentFreq` – a `double` array used to determine the subset of the observations that belong to the current node

`splitValue` – a `double` array representing the resulting split point if the selected variable is quantitative

splitCriterionValue – a double, the value of the criterion used to determine the splitting variable

splitPartition – an int array indicating the resulting split partition if the selected variable is categorical

Returns

an int specifying the index of the split variable in this.getPredictorIndexes()

setNumberOfSurrogateSplits

public void setNumberOfSurrogateSplits(int nSplits)

Description

Sets the number of surrogate splits.

Parameter

nSplits – an int specifying the number of predictors to consider as surrogate splitting variables
Default: nSplits = 0

Example: ALACART and C45

In this example, we use a small data set with response variable, Play, which indicates whether a golfer plays (1) or does not play (0) golf under weather conditions measured by Temperature, Humidity, Outlook (Sunny (0), Overcast (1), Rainy (2)), and Wind (True (0), False (1)). A decision tree is generated by C45 and the ALACART class. The control parameters are adjusted because of the small data size and no cross-validation or pruning is performed. The maximal trees are printed out using DecisionTree.printDecisionTree. Notice that C45 splits on Outlook, then Humidity and Wind, while ALACART splits on Outlook, then Temperature.

```
import com.imsl.datamining.decisionTree.*;

public class C45ALACART {

    public static void main(String[] args) throws Exception {

        int golfResponseIdx = 4;
        double[][] golfXY = {
            {0, 85, 85, 0, 0}, {0, 80, 90, 1, 0}, {1, 83, 78, 0, 1},
            {2, 70, 96, 0, 1}, {2, 68, 80, 0, 1}, {2, 65, 70, 1, 0},
            {1, 64, 65, 1, 1}, {0, 72, 95, 0, 0}, {0, 69, 70, 0, 1},
            {2, 75, 80, 0, 1}, {0, 75, 70, 1, 1}, {1, 72, 90, 1, 1},
            {1, 81, 75, 0, 1}, {2, 71, 80, 1, 0}
        };

        DecisionTree.VariableType[] golfVarType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };
```

```
        String[] names = {
            "Outlook", "Temperature", "Humidity", "Wind", "Play"
        };
        String[] classNames = {"Don’t Play", "Play"};
        String[] varLevels = {"Sunny", "Overcast", "Rainy", "False", "True"};

        C45 dt = new C45(golfXY, golfResponseIdx, golfVarType);
        dt.setMinObsPerChildNode(2);
        dt.setMinObsPerNode(3);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\n\nDecision Tree using Method C4.5:");
        dt.printDecisionTree(null, names, classNames,
                varLevels, true);

        ALACART adt = new ALACART(golfXY, golfResponseIdx, golfVarType);
        adt.setMinObsPerChildNode(2);
        adt.setMinObsPerNode(3);
        adt.setMaxNodes(50);
        adt.fitModel();

        System.out.println("\n\nDecision Tree using Method ALACART:");
        adt.printDecisionTree(null, names, classNames,
                varLevels, true);
    }
}
```

Output

```
Decision Tree using Method C4.5:

Decision Tree:


Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  4  5
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play

Node 1: Cost = 0.143, N= 5, Level = 1, Child nodes:  2  3
   Rule:  Outlook  in: { Sunny }
    P(Y=0)= 0.600
    P(Y=1)= 0.400
    Predicted Y:  Don’t Play

Node 2: Cost = 0.000, N= 2, Level = 2
      Rule:  Humidity       <= 77.500
        P(Y=0)= 0.000
        P(Y=1)= 1.000
        Predicted Y:  Play
```

```
Node 3: Cost = 0.000, N= 3, Level = 2
      Rule:  Humidity       > 77.500
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play

Node 4: Cost = 0.000, N= 4, Level = 1
   Rule:  Outlook  in: { Overcast }
    P(Y=0)= 0.000
    P(Y=1)= 1.000
    Predicted Y:  Play

Node 5: Cost = 0.143, N= 5, Level = 1, Child nodes:  6  7
   Rule:  Outlook  in: { Rainy }
    P(Y=0)= 0.400
    P(Y=1)= 0.600
    Predicted Y:  Play

Node 6: Cost = 0.000, N= 3, Level = 2
      Rule:  Wind  in: { False }
        P(Y=0)= 0.000
        P(Y=1)= 1.000
        Predicted Y:  Play

Node 7: Cost = 0.000, N= 2, Level = 2
      Rule:  Wind  in: { True }
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play


Decision Tree using Method ALACART:

Decision Tree:


Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  8
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play

Node 1: Cost = 0.357, N= 10, Level = 1, Child nodes:  2  7
   Rule:  Outlook  in: { Sunny  Rainy }
    P(Y=0)= 0.500
    P(Y=1)= 0.500
    Predicted Y:  Don't Play

Node 2: Cost = 0.214, N= 8, Level = 2, Child nodes:  3  6
      Rule:  Temperature       <= 77.500
        P(Y=0)= 0.375
        P(Y=1)= 0.625
        Predicted Y:  Play

Node 3: Cost = 0.214, N= 6, Level = 3, Child nodes:  4  5
         Rule:  Temperature          <= 73.500
            P(Y=0)= 0.500
```

```
            P(Y=1)= 0.500
            Predicted Y:  Don't Play

Node 4: Cost = 0.071, N= 4, Level = 4
            Rule:  Temperature             <= 70.500
                P(Y=0)= 0.250
                P(Y=1)= 0.750
                Predicted Y:  Play

Node 5: Cost = 0.000, N= 2, Level = 4
            Rule:  Temperature             > 70.500
                P(Y=0)= 1.000
                P(Y=1)= 0.000
                Predicted Y:  Don't Play

Node 6: Cost = 0.000, N= 2, Level = 3
         Rule:  Temperature          > 73.500
            P(Y=0)= 0.000
            P(Y=1)= 1.000
            Predicted Y:  Play

Node 7: Cost = 0.000, N= 2, Level = 2
      Rule:  Temperature       > 77.500
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play

Node 8: Cost = 0.000, N= 4, Level = 1
   Rule:  Outlook  in: { Overcast }
    P(Y=0)= 0.000
    P(Y=1)= 1.000
    Predicted Y:  Play
```

C45 class

```
public class com.imsl.datamining.decisionTree.C45 extends
com.imsl.datamining.decisionTree.DecisionTreeInfoGain implements Serializable,
Cloneable
```

Generates a decision tree using the C4.5 algorithm for a categorical response variable and categorical or quantitative predictor variables. The C4.5 procedure (Quinlan, 1995) partitions the sample space using an information gain or a gain ratio as the splitting criterion. Specifically, the *entropy* or *uncertainty* in the response variable with C categories over the full training sample S is defined as

$$E(S) = -\sum_{i=1}^{C} p_i \log(p_i)$$

Where $p_i = \Pr[Y = i|S]$ is the probability that the response takes on category i on the dataset S. This measure is widely known as the Shannon Entropy. Splitting the dataset further may either increase or decrease the entropy in the response variable. For example, the entropy of Y over a partitioning of S by X, a variable with K categories, is given by

$$E(S,X) = -\sum_{k=1}^{K}\sum_{i=1}^{C_k} p(S_k)\, E(S_k)$$

If any split defined by the values of a categorical predictor decreases the entropy in Y, then it is said to yield *information gain*:

$$g(S,X) = E(S) - E(S,X)$$

The best splitting variable according to the information gain criterion is the variable yielding the largest information gain, calculated in this manner. A modified criterion is the *gain ratio*:

$$gR(S,X) = \frac{E(S) - E(S,X)}{E_X(S)}$$

where

$$E_x(S) = -\sum_{k=1}^{K} \nu_k \log(\nu_k)$$

with

$$\nu_k = \Pr[X = k|S]$$

Note that $E_X(S)$ is just the entropy of the variable X over S. The gain ratio is thought to be less biased toward predictors with many categories. C4.5 treats the continuous variable similarly, except that only binary splits of the form $X \le d$ and $X > d$ are considered, where d is a value in the range of X on S. The best split is determined by the split variable and split point that gives the largest criterion value. It is possible that no variable meets the threshold for further splitting at the current node, in which case growing stops and the node becomes a *terminal* node. Otherwise, the node is split creating two or more child nodes. Then, using the dataset partition defined by the splitting variable and split value, the very same procedure is repeated for each child node. Thus a collection of nodes and child nodes are generated, or, in other words, the tree is *grown*. The growth stops after one or more different conditions are met.

Constructor

C45

```
public C45(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a C45 object for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Method

selectSplitVariable

`protected int selectSplitVariable(double[][] xy, double[] classCounts, double[] parentFreq, double[] splitValue, double[] splitCriterionValue, int[] splitPartition)`

Description

Selects the split variable for the present node using the C45 method.

Parameters

`xy` – a `double` matrix containing the data

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical

`parentFreq` – a `double` array used to indicate which subset of the observations belong in the current node

`splitValue` – a `double` array representing the resulting split point if the selected variable is quantitative

`splitCriterionValue` – a `double`, the value of the criterion used to determine the splitting variable

`splitPartition` – an `int` array indicating the resulting split partition if the selected variable is categorical

Returns

an `int` specifying the column index of the split variable in `this.getPredictorIndexes()`

Example: ALACART and C45

In this example, we use a small data set with response variable, Play, which indicates whether a golfer plays (1) or does not play (0) golf under weather conditions measured by Temperature, Humidity, Outlook (Sunny (0), Overcast (1), Rainy (2)), and Wind (True (0), False (1)). A decision tree is generated by `C45` and the `ALACART` class. The control parameters are adjusted because of the small data size and no cross-validation or pruning is performed. The maximal trees are printed out using `DecisionTree.printDecisionTree`. Notice that `C45` splits on Outlook, then Humidity and Wind, while `ALACART` splits on Outlook, then Temperature.

```java
import com.imsl.datamining.decisionTree.*;

public class C45ALACART {

    public static void main(String[] args) throws Exception {

        int golfResponseIdx = 4;
        double[][] golfXY = {
            {0, 85, 85, 0, 0}, {0, 80, 90, 1, 0}, {1, 83, 78, 0, 1},
            {2, 70, 96, 0, 1}, {2, 68, 80, 0, 1}, {2, 65, 70, 1, 0},
            {1, 64, 65, 1, 1}, {0, 72, 95, 0, 0}, {0, 69, 70, 0, 1},
            {2, 75, 80, 0, 1}, {0, 75, 70, 1, 1}, {1, 72, 90, 1, 1},
            {1, 81, 75, 0, 1}, {2, 71, 80, 1, 0}
        };

        DecisionTree.VariableType[] golfVarType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        String[] names = {
            "Outlook", "Temperature", "Humidity", "Wind", "Play"
        };
        String[] classNames = {"Don't Play", "Play"};
        String[] varLevels = {"Sunny", "Overcast", "Rainy", "False", "True"};

        C45 dt = new C45(golfXY, golfResponseIdx, golfVarType);
        dt.setMinObsPerChildNode(2);
        dt.setMinObsPerNode(3);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\n\nDecision Tree using Method C4.5:");
        dt.printDecisionTree(null, names, classNames,
                varLevels, true);

        ALACART adt = new ALACART(golfXY, golfResponseIdx, golfVarType);
        adt.setMinObsPerChildNode(2);
        adt.setMinObsPerNode(3);
        adt.setMaxNodes(50);
        adt.fitModel();

        System.out.println("\n\nDecision Tree using Method ALACART:");
        adt.printDecisionTree(null, names, classNames,
                varLevels, true);
    }
}
```

Output

```
Decision Tree using Method C4.5:

Decision Tree:


Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  4  5
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play

Node 1: Cost = 0.143, N= 5, Level = 1, Child nodes:  2  3
   Rule:  Outlook  in: { Sunny }
    P(Y=0)= 0.600
    P(Y=1)= 0.400
    Predicted Y:  Don't Play

Node 2: Cost = 0.000, N= 2, Level = 2
      Rule:  Humidity       <= 77.500
        P(Y=0)= 0.000
        P(Y=1)= 1.000
        Predicted Y:  Play

Node 3: Cost = 0.000, N= 3, Level = 2
      Rule:  Humidity       > 77.500
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play

Node 4: Cost = 0.000, N= 4, Level = 1
   Rule:  Outlook  in: { Overcast }
    P(Y=0)= 0.000
    P(Y=1)= 1.000
    Predicted Y:  Play

Node 5: Cost = 0.143, N= 5, Level = 1, Child nodes:  6  7
   Rule:  Outlook  in: { Rainy }
    P(Y=0)= 0.400
    P(Y=1)= 0.600
    Predicted Y:  Play

Node 6: Cost = 0.000, N= 3, Level = 2
      Rule:  Wind  in: { False }
        P(Y=0)= 0.000
        P(Y=1)= 1.000
        Predicted Y:  Play

Node 7: Cost = 0.000, N= 2, Level = 2
      Rule:  Wind  in: { True }
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play


Decision Tree using Method ALACART:

Decision Tree:
```

```
Node 0: Cost = 0.357, N= 14, Level = 0, Child nodes:  1  8
P(Y=0)= 0.357
P(Y=1)= 0.643
Predicted Y:  Play

Node 1: Cost = 0.357, N= 10, Level = 1, Child nodes:  2  7
   Rule:  Outlook  in: { Sunny  Rainy }
    P(Y=0)= 0.500
    P(Y=1)= 0.500
    Predicted Y:  Don't Play

Node 2: Cost = 0.214, N= 8, Level = 2, Child nodes:  3  6
      Rule:  Temperature       <= 77.500
        P(Y=0)= 0.375
        P(Y=1)= 0.625
        Predicted Y:  Play

Node 3: Cost = 0.214, N= 6, Level = 3, Child nodes:  4  5
         Rule:  Temperature          <= 73.500
            P(Y=0)= 0.500
            P(Y=1)= 0.500
            Predicted Y:  Don't Play

Node 4: Cost = 0.071, N= 4, Level = 4
            Rule:  Temperature             <= 70.500
                P(Y=0)= 0.250
                P(Y=1)= 0.750
                Predicted Y:  Play

Node 5: Cost = 0.000, N= 2, Level = 4
            Rule:  Temperature             > 70.500
                P(Y=0)= 1.000
                P(Y=1)= 0.000
                Predicted Y:  Don't Play

Node 6: Cost = 0.000, N= 2, Level = 3
         Rule:  Temperature          > 73.500
            P(Y=0)= 0.000
            P(Y=1)= 1.000
            Predicted Y:  Play

Node 7: Cost = 0.000, N= 2, Level = 2
      Rule:  Temperature       > 77.500
        P(Y=0)= 1.000
        P(Y=1)= 0.000
        Predicted Y:  Don't Play

Node 8: Cost = 0.000, N= 4, Level = 1
   Rule:  Outlook  in: { Overcast }
    P(Y=0)= 0.000
    P(Y=1)= 1.000
    Predicted Y:  Play
```

Example: C45 Simulated Categorical Data

This example uses the C45 method on simulated categorical data and demonstrates printing the tree structure with and without custom labels.

```
import com.imsl.datamining.decisionTree.*;

public class C45SimulatedCategoricalData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {2, 0, 2}, {1, 0, 0}, {2, 1, 3}, {0, 1, 0}, {1, 2, 0}, {2, 2, 3},
            {2, 2, 3}, {0, 1, 0}, {0, 0, 0}, {0, 1, 0}, {1, 2, 0}, {2, 0, 2},
            {0, 2, 0}, {2, 0, 1}, {0, 0, 0}, {2, 0, 1}, {1, 0, 0}, {0, 2, 0},
            {2, 0, 1}, {1, 2, 0}, {0, 2, 2}, {2, 1, 3}, {1, 1, 0}, {2, 2, 3},
            {1, 2, 0}, {2, 2, 3}, {2, 0, 1}, {2, 1, 3}, {1, 2, 0}, {1, 1, 0}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        String responseName = "Response";
        String[] names = {"Var1", "Var2"};
        String[] classNames = {"c1", "c2", "c3", "c4"};
        String[] varLabels = {"L1", "L2", "L3", "A", "B", "C"};

        C45 dt = new C45(xy, 2, varType);
        dt.setMinObsPerChildNode(5);
        dt.setMinObsPerNode(10);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\nGenerated labels:");
        dt.printDecisionTree(true);
        System.out.println("\nCustom labels:");
        dt.printDecisionTree(responseName, names,
                classNames, varLabels, false);
    }
}
```

Output

```
Generated labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2  3
P(Y=0)= 0.533
P(Y=1)= 0.133
```

```
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Y:   0

Node 1: Cost = 0.033, N= 8, Level = 1
   Rule: X0  in: { 0 }
    P(Y=0)= 0.875
    P(Y=1)= 0.000
    P(Y=2)= 0.125
    P(Y=3)= 0.000
    Predicted Y:   0

Node 2: Cost = 0.000, N= 9, Level = 1
   Rule: X0  in: { 1 }
    P(Y=0)= 1.000
    P(Y=1)= 0.000
    P(Y=2)= 0.000
    P(Y=3)= 0.000
    Predicted Y:   0

Node 3: Cost = 0.200, N= 13, Level = 1
   Rule: X0  in: { 2 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Y:   3

Custom labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2  3
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Response:  c1

Node 1: Cost = 0.033, N= 8, Level = 1
   Rule:  Var1  in: { L1 }
    P(Y=0)= 0.875
    P(Y=1)= 0.000
    P(Y=2)= 0.125
    P(Y=3)= 0.000
    Predicted Response:  c1

Node 2: Cost = 0.000, N= 9, Level = 1
   Rule:  Var1  in: { L2 }
    P(Y=0)= 1.000
    P(Y=1)= 0.000
    P(Y=2)= 0.000
    P(Y=3)= 0.000
    Predicted Response:  c1
```

```
Node 3: Cost = 0.200, N= 13, Level = 1
   Rule:  Var1  in: { L3 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Response:  c4
```

CHAID class

```
public class com.imsl.datamining.decisionTree.CHAID extends
com.imsl.datamining.decisionTree.DecisionTree implements Serializable,
Cloneable
```

Generates a decision tree using CHAID for categorical or discrete ordered predictor variables. Due to Kass (1980), CHAID is an acronym for chi-square automatic interaction detection. At each node, `CHAID` looks for the best splitting variable using the following steps: given a predictor variable *X*, perform a 2-way chi-squared test of association between each possible pair of categories of *X* with the categories of *Y*. The least significant result is noted and, if a threshold is met, the two categories of *X* are merged.

Next, treating this merged category as a single category, CHAID repeats the series of tests to determine if there is further merging possible. If a merged category consists of three or more of the original categories of *X*, `CHAID` calls for a step to test whether the merged categories should be split. This is done by forming all binary partitions of the merged category and testing each one against *Y* in a 2-way test of association. If the most significant result meets a threshold, then the merged category is split accordingly. As long as the threshold in this step is smaller than the threshold in the merge step, the splitting step and the merge step will not cycle back and forth.

Once each predictor is processed in this manner, the predictor with the most significant qualifying 2-way test with *Y* is selected as the splitting variable, and its last state of merged categories defines the split at the given node. If none of the tests qualify (by having an adjusted p-value smaller than a threshold), then the node is not split. This growing procedure continues until one or more stopping conditions are met.

Constructor

CHAID

```
public CHAID(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Constructs a `CHAID` object for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix containing the training data and associated response values

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

getMergeCategoriesSigLevel

```
public double getMergeCategoriesSigLevel()
```

Description

Returns the significance level for merging categories.

Returns

a `double`, the significance level for merging categories

getSplitMergedCategoriesSigLevel

```
public double getSplitMergedCategoriesSigLevel()
```

Description

Returns the significance level for splitting previously merged categories.

Returns

a `double`, the significance level for splitting merged categories

getSplitVariableSignificanceLevel

```
public double getSplitVariableSignificanceLevel()
```

Description

Returns the significance level for split variable selection.

Returns

a `double`, the significance level for split variable selection

selectSplitVariable

```
protected int selectSplitVariable(double[][] xy, double[] classCounts, double[]
parentFreq, double[] splitValue, double[] splitCriterionValue, int[]
splitPartition)
```

Description

Selects the split variable for the current node using CHAID (chi-square automatic interaction detection).

Parameters

`xy` – a `double` matrix containing the data

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical

`parentFreq` – a `double` array used to indicate which subset of the observations belong in the current node

`splitValue` – a `double` array representing the resulting split point if the selected variable is quantitative

`splitCriterionValue` – a `double`, the value of the criterion used to determine the splitting variable

`splitPartition` – an `int` array indicating the resulting split partition if the selected variable is categorical

Returns

an `int` specifying the column index of the split variable in `this.getPredictorIndexes()`

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
DecisionTree.PruningFailedToConvergeException,
PredictiveModel.StateChangeException,
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Sets the configuration of `PredictiveModel` to that of the input model.

Parameter

`pm` – a `PredictiveModel` object which is to have its attributes duplicated in this instance

Exceptions

`PruningFailedToConvergeException` is thrown when pruning fails to converge.

`PredictiveModel.StateChangeException` is thrown when an input parameter has changed that might affect the model estimates or predictions.

`com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException` is thrown when the sum of the probabilities does not equal 1.

setMergeCategoriesSignificanceLevel

```
public void setMergeCategoriesSignificanceLevel(double mergeAlpha)
```

Description

Sets the significance level for merging categories.

Parameter

`mergeAlpha` – a `double`, specifying the significance level for merging categories

`mergeAlpha` must be between 0.0 and 1.0. In addition, if `splitMergeAlpha` is set to enable splitting of previously merged categories, then `mergeAlpha` ≤ `splitMergeAlpha`.

Default: `mergeAlpha` = 0.05.

setSplitMergedCategoriesSigLevel

```
public void setSplitMergedCategoriesSigLevel(double splitMergedAlpha)
```

Description

Sets the significance level for splitting previously merged categories.

Parameter

`splitMergedAlpha` – a `double` specifying the significance level for splitting merged categories

`splitMergeAlpha` must be greater than or equal to `com.imsl.datamining.decisionTree.CHAID.getMergeCategoriesSigLevel` (p. 2284) unless disabled using `splitMergeAlpha=-1.` Default: `splitMergeAlpha` = -1.0 disables splitting of merged categories.

setSplitVariableSignificanceLevel

```
public void setSplitVariableSignificanceLevel(double
splitVariableSelectionAlpha)
```

Description

Sets the significance level for split variable selection.

Parameter

`splitVariableSelectionAlpha` – a `double` specifying the significance level for split variable selection

`splitVariableSelectionAlpha` must be between 0.0 and 1.0.

Default: `splitVariableSelectionAlpha` = 0.05.

Example: CHAID Simulated Categorical Data

This example uses the CHAID method on simulated categorical data and demonstrates printing the tree structure with and without custom labels.

```
import com.imsl.datamining.decisionTree.*;

public class CHAIDSimulatedCategoricalData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {2, 0, 2}, {1, 0, 0}, {2, 1, 3}, {0, 1, 0}, {1, 2, 0}, {2, 2, 3},
            {2, 2, 3}, {0, 1, 0}, {0, 0, 0}, {0, 1, 0}, {1, 2, 0}, {2, 0, 2},
            {0, 2, 0}, {2, 0, 1}, {0, 0, 0}, {2, 0, 1}, {1, 0, 0}, {0, 2, 0},
            {2, 0, 1}, {1, 2, 0}, {0, 2, 2}, {2, 1, 3}, {1, 1, 0}, {2, 2, 3},
```

```
            {1, 2, 0}, {2, 2, 3}, {2, 0, 1}, {2, 1, 3}, {1, 2, 0}, {1, 1, 0}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        String responseName = "Response";
        String[] names = {"Var1", "Var2"};
        String[] classNames = {"c1", "c2", "c3", "c4"};
        String[] varLabels = {"L1", "L2", "L3", "A", "B", "C"};

        CHAID dt = new CHAID(xy, 2, varType);
        dt.setMinObsPerChildNode(5);
        dt.setMinObsPerNode(10);
        dt.setMaxNodes(50);
        dt.fitModel();

        System.out.println("\nGenerated labels:");
        dt.printDecisionTree(true);
        System.out.println("\nCustom labels:");
        dt.printDecisionTree(responseName, names,
                classNames, varLabels, false);
    }
}
```

Output

```
Generated labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Y:   0

Node 1: Cost = 0.033, N= 17, Level = 1
   Rule: X0  in: { 0  1 }
    P(Y=0)= 0.941
    P(Y=1)= 0.000
    P(Y=2)= 0.059
    P(Y=3)= 0.000
    Predicted Y:   0

Node 2: Cost = 0.200, N= 13, Level = 1, Child nodes:  3  4
   Rule: X0  in: { 2 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
```

```
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Y:   3

Node 3: Cost = 0.067, N= 6, Level = 2
      Rule: X1  in: { 0 }
        P(Y=0)= 0.000
        P(Y=1)= 0.667
        P(Y=2)= 0.333
        P(Y=3)= 0.000
        Predicted Y:   1

Node 4: Cost = 0.000, N= 7, Level = 2
      Rule: X1  in: { 1  2 }
        P(Y=0)= 0.000
        P(Y=1)= 0.000
        P(Y=2)= 0.000
        P(Y=3)= 1.000
        Predicted Y:   3

Custom labels:

Decision Tree:


Node 0: Cost = 0.467, N= 30, Level = 0, Child nodes:  1  2
P(Y=0)= 0.533
P(Y=1)= 0.133
P(Y=2)= 0.100
P(Y=3)= 0.233
Predicted Response:  c1

Node 1: Cost = 0.033, N= 17, Level = 1
   Rule:  Var1  in: { L1  L2 }
    P(Y=0)= 0.941
    P(Y=1)= 0.000
    P(Y=2)= 0.059
    P(Y=3)= 0.000
    Predicted Response:  c1

Node 2: Cost = 0.200, N= 13, Level = 1, Child nodes:  3  4
   Rule:  Var1  in: { L3 }
    P(Y=0)= 0.000
    P(Y=1)= 0.308
    P(Y=2)= 0.154
    P(Y=3)= 0.538
    Predicted Response:  c4

Node 3: Cost = 0.067, N= 6, Level = 2
      Rule:  Var2  in: { A }
        P(Y=0)= 0.000
        P(Y=1)= 0.667
        P(Y=2)= 0.333
        P(Y=3)= 0.000
        Predicted Response:  c2
```

```
Node 4: Cost = 0.000, N= 7, Level = 2
      Rule:  Var2  in: { B  C }
        P(Y=0)= 0.000
        P(Y=1)= 0.000
        P(Y=2)= 0.000
        P(Y=3)= 1.000
        Predicted Response:  c4
```

QUEST class

```
public class com.imsl.datamining.decisionTree.QUEST extends
com.imsl.datamining.decisionTree.DecisionTree implements Serializable,
Cloneable
```

Generates a decision tree using the QUEST algorithm for a categorical response variable and categorical or quantitative predictor variables. The procedure (Loh and Shih, 1997) is as follows: For each categorical predictor, QUEST performs a multi-way chi-square test of association between the predictor and Y. For every continuous predictor, QUEST performs an ANOVA test to see if the means of the predictor vary among the groups of Y. Among these tests, the variable with the most significant result is selected as a potential splitting variable, say, X_j. If the p-value (adjusted for multiple tests) is less than the specified splitting threshold, then X_j is the splitting variable for the current node. If not, QUEST performs for each continuous variable X a Levene's test of homogeneity to see if the variance of X varies within the different groups of Y. Among these tests, we again find the predictor with the most significant result, say X_i. If its p-value (adjusted for multiple tests) is less than the splitting threshold, X_i is the splitting variable. Otherwise, the node is not split.

Assuming a splitting variable is found, the next step is to determine how the variable should be split. If the selected variable X_j is continuous, a split point d is determined by quadratic discriminant analysis (QDA) of X_j into two populations determined by a binary partition of the response Y. The goal of this step is to group the classes of Y into two subsets or super classes, A and B. If there are only two classes in the response Y, the super classes are obvious. Otherwise, calculate the means and variances of X_j in each of the classes of Y. If the means are all equal, put the largest-sized class into group A and combine the rest to form group B. If they are not all equal, use a k-means clustering method ($k = 2$) on the class means to determine A and B.

X_j in A and in B is assumed to be normally distributed with estimated means $\bar{x}_{j|A}$, $\bar{x}_{j|B}$, and variances $S^2{}_{j|A}$, $S^2{}_{j|B}$, respectively. The quadratic discriminant is the partition $X_j \leq d$ and $X_j > d$ such that $\Pr(X_j, A) = \Pr(X_j, B)$. The discriminant rule assigns an observation to A if $x_{ij} \leq d$ and to B if $x_{ij} > d$. For d to maximally discriminate, the probabilities must be equal.

If the selected variable X_j is categorical, it is first transformed using the method outlined in Loh and Shih (1997) and then QDA is performed as above. The transformation is related to the discriminant coordinate (CRIMCOORD) approach due to Gnanadesikan (1977).

Constructor

QUEST

```
public QUEST(double[][] xy, int responseColumnIndex,
PredictiveModel.VariableType[] varType)
```

Description

Instantiates a `QUEST` object for a single response variable and multiple predictor variables.

Parameters

`xy` – a `double` matrix with rows containing the observations on the predictor variables and one response variable

`responseColumnIndex` – an `int` specifying the column index of the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

getSplitVariableSelectionCriterion

```
public double getSplitVariableSelectionCriterion()
```

Description

Returns the significance level for split variable selection.

Returns

a `double`, the significance criterion for split variable selection

selectSplitVariable

```
protected int selectSplitVariable(double[][] xy, double[] classCounts, double[]
parentFreq, double[] splitValue, double[] splitCriterionValue, int[]
splitPartition)
```

Description

Selects the split variable for the present node using the QUEST method.

Parameters

`xy` – a `double` matrix containing the data

`classCounts` – a `double` array containing the counts for each class of the response variable, when it is categorical

`parentFreq` – a `double` array used to determine which subset of the observations belong in the current node

`splitValue` – a `double` array representing the resulting split point if the selected variable is quantitative

splitCriterionValue – a double, the value of the criterion used to determine the splitting variable

splitPartition – an int array indicating the resulting split partition if the selected variable is categorical

Returns

an int specifying the column index of the split variable in xy

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
DecisionTree.PruningFailedToConvergeException,
PredictiveModel.StateChangeException,
PredictiveModel.SumOfProbabilitiesNotOneException
```

Description

Sets the configuration of PredictiveModel to that of the input model.

Parameter

pm – a PredictiveModel object which is to have its attributes duplicated in this instance

Exceptions

PruningFailedToConvergeException is thrown when pruning fails to converge.

PredictiveModel.StateChangeException is thrown when an input parameter has changed that might affect the model estimates or predictions.

com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException is thrown when the sum of the probabilities does not equal 1.

setSplitVariableSelectionCriterion

```
public void setSplitVariableSelectionCriterion(double criterion)
```

Description

Sets the significance level for split variable selection.

Parameter

criterion – a double specifying the criterion for split variable selection. criterion must be between 0.0 and 1.0.

Default: criterion = 0.05

Example: QUEST Simulated Categorical Data

This example applies the QUEST method to a simulated data set with 50 cases and three predictors of mixed-type. Shown are the maximum-sized tree under default controls and the subtree resulting from pruning with a cost-complexity value of 0.0.

```
import com.imsl.datamining.decisionTree.*;

public class QUESTSimulatedCategoricalData {
```

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

        double[][] xy = {
            {2, 25.928690, 0, 0}, {1, 51.632450, 1, 1}, {1, 25.784321, 0, 2},
            {0, 39.379478, 0, 3}, {2, 24.650579, 0, 2}, {2, 45.200840, 0, 2},
            {2, 52.679600, 1, 3}, {1, 44.283421, 1, 3}, {2, 40.635231, 1, 3},
            {2, 51.760941, 0, 3}, {2, 26.303680, 0, 1}, {2, 20.702299, 1, 0},
            {2, 38.742729, 1, 3}, {2, 19.473330, 0, 0}, {1, 26.422110, 0, 0},
            {2, 37.059860, 1, 0}, {1, 51.670429, 1, 3}, {0, 42.401562, 0, 3},
            {2, 33.900269, 1, 2}, {1, 35.432819, 0, 0}, {1, 44.303692, 0, 1},
            {0, 46.723869, 0, 2}, {1, 46.992619, 0, 2}, {0, 36.059231, 0, 3},
            {2, 36.831970, 1, 1}, {1, 61.662571, 1, 2}, {0, 25.677139, 0, 3},
            {1, 39.085670, 1, 0}, {0, 48.843410, 1, 1}, {1, 39.343910, 0, 3},
            {2, 24.735220, 0, 2}, {1, 50.552509, 1, 3}, {0, 31.342630, 1, 3},
            {1, 27.157949, 1, 0}, {0, 31.726851, 0, 2}, {0, 25.004080, 0, 3},
            {1, 26.354570, 1, 3}, {2, 38.123428, 0, 1}, {0, 49.940300, 0, 2},
            {1, 42.457790, 1, 3}, {0, 38.809479, 1, 1}, {0, 43.227989, 1, 1},
            {0, 41.876240, 0, 3}, {2, 48.078201, 0, 2}, {0, 43.236729, 1, 0},
            {2, 39.412941, 0, 3}, {1, 23.933460, 0, 2}, {2, 42.841301, 1, 3},
            {2, 30.406691, 0, 1}, {0, 37.773891, 0, 2}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        QUEST dt = new QUEST(xy, 3, varType);
        dt.setPrintLevel(1);
        dt.fitModel();
        dt.pruneTree(0.0);

        System.out.println("\nMaximal tree: \n");
        dt.printDecisionTree(true);

        System.out.println("\nPruned subtree (cost-complexity = 0): \n");
        dt.printDecisionTree(false);
    }
}
```

Output

```
Growing the maximal tree using method QUEST:


Maximal tree:


Decision Tree:


Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes:  1  2
```

```
P(Y=0)= 0.180
P(Y=1)= 0.180
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y:   3

Node 1: Cost = 0.220, N= 17, Level = 1
   Rule: X1    <= 35.031
    P(Y=0)= 0.294
    P(Y=1)= 0.118
    P(Y=2)= 0.353
    P(Y=3)= 0.235
    Predicted Y:   2

Node 2: Cost = 0.360, N= 33, Level = 1, Child nodes:  3  4
   Rule: X1    > 35.031
    P(Y=0)= 0.121
    P(Y=1)= 0.212
    P(Y=2)= 0.212
    P(Y=3)= 0.455
    Predicted Y:   3

Node 3: Cost = 0.180, N= 19, Level = 2
      Rule: X1       <= 43.265
        P(Y=0)= 0.211
        P(Y=1)= 0.211
        P(Y=2)= 0.053
        P(Y=3)= 0.526
        Predicted Y:   3

Node 4: Cost = 0.160, N= 14, Level = 2
      Rule: X1       > 43.265
        P(Y=0)= 0.000
        P(Y=1)= 0.214
        P(Y=2)= 0.429
        P(Y=3)= 0.357
        Predicted Y:   2

Pruned subtree (cost-complexity = 0):


Decision Tree:


Node 0: Cost = 0.620, N= 50, Level = 0, Child nodes:  1  2
P(Y=0)= 0.180
P(Y=1)= 0.180
P(Y=2)= 0.260
P(Y=3)= 0.380
Predicted Y:   3

Node 1: Cost = 0.220, N= 17, Level = 1
   Rule: X1    <= 35.031
    P(Y=0)= 0.294
    P(Y=1)= 0.118
    P(Y=2)= 0.353
```

```
    P(Y=3)= 0.235
    Predicted Y:   2

Node 2: Cost = 0.360, N= 33, Level = 1
   Rule: X1    > 35.031
    P(Y=0)= 0.121
    P(Y=1)= 0.212
    P(Y=2)= 0.212
    P(Y=3)= 0.455
    Predicted Y:   3
Pruned at Node id 2.
```

Example: QUEST Kyphosis Data

This example uses the dataset Kyphosis. The 81 cases represent 81 children who have undergone surgery to correct a type of spinal deformity known as Kyphosis. The response variable is the presence or absence of Kyphosis after the surgery. Three predictors are Age of the patient in months, Start, the number of the vertebra where the surgery started, and Number, the number of vertebra involved in the surgery. This example uses the method QUEST to produce a maximal tree. It also requests predictions for a test-data set consisting of 10 “new” cases.

```
import com.imsl.datamining.decisionTree.*;

public class QUESTKyphosisData {

    public static void main(String[] args) throws Exception {

        double[][] xy = {
            {0, 71, 3, 5}, {0, 158, 3, 14}, {1, 128, 4, 5}, {0, 2, 5, 1},
            {0, 1, 4, 15}, {0, 1, 2, 16}, {0, 61, 2, 17}, {0, 37, 3, 16},
            {0, 113, 2, 16}, {1, 59, 6, 12}, {1, 82, 5, 14}, {0, 148, 3, 16},
            {0, 18, 5, 2}, {0, 1, 4, 12}, {0, 168, 3, 18}, {0, 1, 3, 16},
            {0, 78, 6, 15}, {0, 175, 5, 13}, {0, 80, 5, 16},
            {0, 27, 4, 9}, {0, 22, 2, 16}, {1, 105, 6, 5}, {1, 96, 3, 12},
            {0, 131, 2, 3}, {1, 15, 7, 2}, {0, 9, 5, 13}, {0, 8, 3, 6},
            {0, 100, 3, 14}, {0, 4, 3, 16}, {0, 151, 2, 16}, {0, 31, 3, 16},
            {0, 125, 2, 11}, {0, 130, 5, 13}, {0, 112, 3, 16}, {0, 140, 5, 11},
            {0, 93, 3, 16}, {0, 1, 3, 9}, {1, 52, 5, 6}, {0, 20, 6, 9},
            {1, 91, 5, 12}, {1, 73, 5, 1}, {0, 35, 3, 13}, {0, 143, 9, 3},
            {0, 61, 4, 1}, {0, 97, 3, 16}, {1, 139, 3, 10}, {0, 136, 4, 15},
            {0, 131, 5, 13}, {1, 121, 3, 3}, {0, 177, 2, 14}, {0, 68, 5, 10},
            {0, 9, 2, 17}, {1, 139, 10, 6}, {0, 2, 2, 17}, {0, 140, 4, 15},
            {0, 72, 5, 15}, {0, 2, 3, 13}, {1, 120, 5, 8}, {0, 51, 7, 9},
            {0, 102, 3, 13}, {1, 130, 4, 1}, {1, 114, 7, 8}, {0, 81, 4, 1},
            {0, 118, 3, 16}, {0, 118, 4, 16}, {0, 17, 4, 10}, {0, 195, 2, 17},
            {0, 159, 4, 13}, {0, 18, 4, 11}, {0, 15, 5, 16}, {0, 158, 5, 14},
            {0, 127, 4, 12}, {0, 87, 4, 16}, {0, 206, 4, 10}, {0, 11, 3, 15},
            {0, 178, 4, 15}, {1, 157, 3, 13}, {0, 26, 7, 13}, {0, 120, 2, 13},
            {1, 42, 7, 6}, {0, 36, 4, 13}
        };

        double[][] xyTest = {
            {0, 71, 3, 5}, {1, 128, 4, 5}, {0, 1, 4, 15}, {0, 61, 6, 10},
```

```
            {0, 113, 2, 16}, {1, 82, 5, 14}, {0, 148, 3, 16}, {0, 1, 4, 12},
            {0, 1, 3, 16}, {0, 175, 5, 13}
        };

        DecisionTree.VariableType[] varType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS
        };

        String[] names = {"Age", "Number", "Start"};
        String[] classNames = {"Absent", "Present"};
        String responseName = "Kyphosis";

        QUEST dt = new QUEST(xy, 0, varType);
        dt.setMinObsPerChildNode(5);
        dt.setMinObsPerNode(10);
        dt.setMaxNodes(50);
        dt.setPrintLevel(2);
        dt.fitModel();

        double[] predictions = dt.predict(xyTest);
        double predErrSS = dt.getMeanSquaredPredictionError();

        dt.printDecisionTree(responseName, names,
                classNames, null, true);

        System.out.println("\nPredictions for test data:");
        System.out.printf("%5s%8s%7s%10s\n", names[0], names[1], names[2],
                responseName);

        for (int i = 0; i < xyTest.length; i++) {
            System.out.printf("%5.0f%8.0f%7.0f", xyTest[i][1], xyTest[i][2],
                    xyTest[i][3]);
            int idx = (int) predictions[i];
            System.out.printf("%10s\n", classNames[idx]);
        }
        System.out.printf("\nMean squared prediction error: %f\n", predErrSS);
    }
}
```

Output

```
Growing the maximal tree using method QUEST:

Node 0has split variable 2
Node 1has split variable 0
Node 2 is a terminal node. It has 7.0 cases--too few cases to split.
Node 3 is a terminal node. It has 6.0 cases--too few cases to split.
Node 4has split variable 2
Node 5 is a terminal node. It has 6.0 cases--too few cases to split.
Node 6has split variable 2
Node 7has split variable 0
Node 8has split variable 0
```

```
Node 8 is a terminal node. The split is too thin having count 2.0.
Node 9has split variable 1
Node 10 is a terminal node. It has 6.0 cases--too few cases to split.
Node 11 is a terminal node, because it is pure.
Node 11 is a terminal node. It has 7.0 cases--too few cases to split.
Node 12is a terminal node. Could not find a splitting variable.

Decision Tree:


Node 0: Cost = 0.210, N= 81, Level = 0, Child nodes:  1  4
P(Y=0)= 0.790
P(Y=1)= 0.210
Predicted Kyphosis:  Absent

Node 1: Cost = 0.074, N= 13, Level = 1, Child nodes:  2  3
Rule:  Start <= 5.155
P(Y=0)= 0.538
P(Y=1)= 0.462
Predicted Kyphosis:  Absent

Node 2: Cost = 0.025, N= 7, Level = 2
Rule:  Age <= 84.030
P(Y=0)= 0.714
P(Y=1)= 0.286
Predicted Kyphosis:  Absent

Node 3: Cost = 0.025, N= 6, Level = 2
Rule:  Age > 84.030
P(Y=0)= 0.333
P(Y=1)= 0.667
Predicted Kyphosis:  Present

Node 4: Cost = 0.136, N= 68, Level = 1, Child nodes:  5  6
Rule:  Start > 5.155
P(Y=0)= 0.838
P(Y=1)= 0.162
Predicted Kyphosis:  Absent

Node 5: Cost = 0.012, N= 6, Level = 2
Rule:  Start <= 8.862
P(Y=0)= 0.167
P(Y=1)= 0.833
Predicted Kyphosis:  Present

Node 6: Cost = 0.074, N= 62, Level = 2, Child nodes:  7  12
Rule:  Start > 8.862
P(Y=0)= 0.903
P(Y=1)= 0.097
Predicted Kyphosis:  Absent

Node 7: Cost = 0.062, N= 28, Level = 3, Child nodes:  8  9
Rule:  Start <= 13.092
P(Y=0)= 0.821
P(Y=1)= 0.179
Predicted Kyphosis:  Absent
```

```
Node 8: Cost = 0.025, N= 15, Level = 4
Rule:  Age <= 91.722
P(Y=0)= 0.867
P(Y=1)= 0.133
Predicted Kyphosis:  Absent

Node 9: Cost = 0.037, N= 13, Level = 4, Child nodes:  10  11
Rule:  Age > 91.722
P(Y=0)= 0.769
P(Y=1)= 0.231
Predicted Kyphosis:  Absent

Node 10: Cost = 0.037, N= 6, Level = 5
Rule:  Number <= 3.450
P(Y=0)= 0.500
P(Y=1)= 0.500
Predicted Kyphosis:  Absent

Node 11: Cost = 0.000, N= 7, Level = 5
Rule:  Number > 3.450
P(Y=0)= 1.000
P(Y=1)= 0.000
Predicted Kyphosis:  Absent

Node 12: Cost = 0.012, N= 34, Level = 3
Rule:  Start > 13.092
P(Y=0)= 0.971
P(Y=1)= 0.029
Predicted Kyphosis:  Absent

Predictions for test data:
  Age  Number  Start  Kyphosis
   71       3      5    Absent
  128       4      5   Present
    1       4     15    Absent
   61       6     10    Absent
  113       2     16    Absent
   82       5     14    Absent
  148       3     16    Absent
    1       4     12    Absent
    1       3     16    Absent
  175       5     13    Absent

Mean squared prediction error: 0.100000
```

RandomTrees class

`public class com.imsl.datamining.decisionTree.RandomTrees extends com.imsl.datamining.PredictiveModel implements Serializable, Cloneable`

Generates predictions using a random forest of decision trees.

A random forest is an ensemble of decision trees. Like bootstrap aggregation, a tree is fit to each of *M* bootstrap samples from the training data. Each tree is then used to generate predictions. For a regression problem (continuous response variable), the *M* predictions are combined into a single predicted value by averaging. For classification (categorical response variable), majority vote is used. A random forest also randomizes the predictors. That is, in every tree, the splitting variable at every node is selected from a random subset of the predictors. Randomization of the predictors reduces correlation among individual trees. The random forest was invented by Leo Breiman in 2001 (Breiman, 2001). Random Forests™ is the trademark term for this approach. Also see Hastie, Tibshirani, and Friedman, 2008, for further discussion.

Constructors

RandomTrees

`public RandomTrees(DecisionTree dt)`

Description

Constructs a `RandomTrees` random forest of the input decision tree.

Parameter

`dt` – a `DecisionTree` object

RandomTrees

`public RandomTrees(double[][] xy, int responseColumnIndex, PredictiveModel.VariableType[] varType)`

Description

Constructs a `RandomTrees` random forest of `ALACART` decision trees.

Parameters

`xy` – a `double` matrix containing the training data

`responseColumnIndex` – an `int`, the column index for the response variable

`varType` – a `com.imsl.datamining.PredictiveModel.VariableType` (p. 1960) array containing the type of each variable

Methods

fitModel

```
public void fitModel() throws PredictiveModel.PredictiveModelException
```

Description

Fits the random forest to the training data.

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

getNumberOfRandomFeatures

```
public int getNumberOfRandomFeatures()
```

Description

Returns the number of random features used in the splitting rules.

Returns

an `int`, the number of random features

getNumberOfTrees

```
public int getNumberOfTrees()
```

Description

Returns the number of trees.

Returns

an `int`, the number of trees

getOutOfBagPredictionError

```
public double getOutOfBagPredictionError()
```

Description

Returns the out-of-bag prediction error.

Returns

a `double`, the out-of-bag prediction error

getOutOfBagPredictions

```
public double[] getOutOfBagPredictions()
```

Description

Returns the out-of-bag predicted values for the examples in the training data.

Returns

a `double` array containing the out-of-bag predictions

getVariableImportance

```
public double[] getVariableImportance()
```

Description

Returns the variable importance measure based on the out-of-bag prediction error.

Variable importance for a predictor is obtained by randomly permuting the out-of-bag values of the predictor and calculating the difference in predictive accuracy, before and after the permutation. The measure is averaged over all the trees.

Returns

a `double` array containing variable importance for each predictor

isCalculateVariableImportance

```
public boolean isCalculateVariableImportance()
```

Description

Returns the current setting of the boolean to calculate variable importance.

Returns

a `boolean`, the current setting of the flag

predict

```
public double[] predict() throws PredictiveModel.PredictiveModelException
```

Description

Returns the predicted values generated by the random forest on the training data.

Returns

a `double` array containing the fitted values

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

predict

```
public double[] predict(double[][] testData) throws
PredictiveModel.PredictiveModelException
```

Description

Returns the predicted values on the input test data.

Parameter

`testData` – a `double` matrix containing test data

Note: `testData` must have the same number of columns as `xy` and the columns must be in the same arrangement as in `xy`.

Returns

a `double` array containing the predicted values

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

predict

```
public double[] predict(double[][] testData, double[] testDataWeights) throws
PredictiveModel.PredictiveModelException
```

Description

Returns the predicted values on the input test data and the test data weights.

Parameters

`testData` – a `double` matrix containing test data

`testDataWeights` – a `double` array containing weight values for each row of `testData`

Note: `testData` must have the same number of columns as `xy` and the columns must be in the same arrangement as in `xy`.

Returns

a `double` array containing the predicted values

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

setCalculateVariableImportance

```
public void setCalculateVariableImportance(boolean calculate)
```

Description

Sets the boolean to calculate variable importance.

When `true`, a permutation type variable importance measure is calculated during bootstrap aggregation.

Parameter

`calculate` – a `boolean` indicating whether or not to calculate variable importance

Default: `calculate` = false

setConfiguration

```
protected void setConfiguration(PredictiveModel pm) throws
PredictiveModel.PredictiveModelException
```

Description

Sets the configuration of `RandomTrees` to that of the input model.

Parameter

`pm` – a `RandomTrees` object

Exception

`com.imsl.datamining.PredictiveModel.PredictiveModelException` is thrown when an exception occurs in the com.imsl.datamining.PredictiveModel. Superclass exceptions should be considered such as com.imsl.datamining.PredictiveModel.StateChangeException and com.imsl.datamining.PredictiveModel.SumOfProbabilitiesNotOneException.

setNumberOfRandomFeatures

```
public void setNumberOfRandomFeatures(int numberOfRandomFeatures)
```

Description

Sets the number of random features used in the splitting rules.

Parameter

`numberOfRandomFeatures` – an `int`, the number of predictors in the random subset

Default: `numberOfRandomFeatures`=$\sqrt{p}$ for classification problems, $\frac{p}{3}$ for regression problems, where p is the number of predictors in the training data.

setNumberOfThreads

```
public void setNumberOfThreads(int numberOfThreads)
```

Description

Sets the maximum number of `java.lang.Thread` instances that may be used for parallel processing.

Parameter

`numberOfThreads` – an `int` specifying the maximum number of `java.lang.Thread` instances that may be used for parallel processing.

The actual number of threads used in parallel processing will be the lesser of `numberOfThreads` and `numberOfTrees`, the number of trees in the random forest. This assessment is made to optimize use of resources.

Default: `numberOfThreads` = 1.

setNumberOfTrees

```
public void setNumberOfTrees(int numberOfTrees)
```

Description

Sets the number of trees to generate in the random forest.

The number of trees is equivalent to the number of bootstrap samples.

Parameter

`numberOfTrees` – an `int`, the number of trees to generate

Default: numberOfTrees=50

Example 1: RandomTrees

This example builds a random forest with ALACART decision trees. A single tree and the random forest are fit to the Kyphosis data and predictions for a test-data set consisting of 10 "new" cases are generated.

The Kyphosis data 81 cases represent 81 children who have undergone surgery to correct a type of spinal deformity known as Kyphosis. The response variable is the presence or absence of Kyphosis after the surgery. The three predictors are:

- Age of the patient in months
- Start, the number of the vertebra where the surgery started
- Number, the number of vertebra involved in the surgery

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

public class RandomTreesEx1 {

    public static void main(String[] args) throws Exception {
        DecisionTree.VariableType[] kyphosisVarType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS
        };
        int kyphosisResponseIdx = 0;
        double[][] kyphosisXY = {
            {0, 71, 3, 5}, {0, 158, 3, 14}, {1, 128, 4, 5}, {0, 2, 5, 1},
            {0, 1, 4, 15}, {0, 1, 2, 16}, {0, 61, 2, 17}, {0, 37, 3, 16},
            {0, 113, 2, 16}, {1, 59, 6, 12}, {1, 82, 5, 14}, {0, 148, 3, 16},
            {0, 18, 5, 2}, {0, 1, 4, 12}, {0, 168, 3, 18}, {0, 1, 3, 16},
            {0, 78, 6, 15}, {0, 175, 5, 13}, {0, 80, 5, 16}, {0, 27, 4, 9},
            {0, 22, 2, 16}, {1, 105, 6, 5}, {1, 96, 3, 12}, {0, 131, 2, 3},
            {1, 15, 7, 2}, {0, 9, 5, 13}, {0, 8, 3, 6}, {0, 100, 3, 14},
            {0, 4, 3, 16}, {0, 151, 2, 16}, {0, 31, 3, 16}, {0, 125, 2, 11},
            {0, 130, 5, 13}, {0, 112, 3, 16}, {0, 140, 5, 11}, {0, 93, 3, 16},
            {0, 1, 3, 9}, {1, 52, 5, 6}, {0, 20, 6, 9}, {1, 91, 5, 12},
            {1, 73, 5, 1}, {0, 35, 3, 13}, {0, 143, 9, 3}, {0, 61, 4, 1},
            {0, 97, 3, 16}, {1, 139, 3, 10}, {0, 136, 4, 15}, {0, 131, 5, 13},
            {1, 121, 3, 3}, {0, 177, 2, 14}, {0, 68, 5, 10}, {0, 9, 2, 17},
            {1, 139, 10, 6}, {0, 2, 2, 17}, {0, 140, 4, 15}, {0, 72, 5, 15},
```

```
            {0, 2, 3, 13}, {1, 120, 5, 8}, {0, 51, 7, 9}, {0, 102, 3, 13},
            {1, 130, 4, 1}, {1, 114, 7, 8}, {0, 81, 4, 1}, {0, 118, 3, 16},
            {0, 118, 4, 16}, {0, 17, 4, 10}, {0, 195, 2, 17}, {0, 159, 4, 13},
            {0, 18, 4, 11}, {0, 15, 5, 16}, {0, 158, 5, 14}, {0, 127, 4, 12},
            {0, 87, 4, 16}, {0, 206, 4, 10}, {0, 11, 3, 15}, {0, 178, 4, 15},
            {1, 157, 3, 13}, {0, 26, 7, 13}, {0, 120, 2, 13}, {1, 42, 7, 6},
            {0, 36, 4, 13}
        };

        double[][] kyphosisXYTest = {
            {0, 71, 3, 5}, {1, 128, 4, 5}, {0, 1, 4, 15}, {0, 61, 6, 10},
            {0, 113, 2, 16}, {1, 82, 5, 14}, {0, 148, 3, 16}, {0, 1, 4, 12},
            {0, 1, 3, 16}, {0, 175, 5, 13}
        };
        ALACART dt
                = new ALACART(kyphosisXY, kyphosisResponseIdx, kyphosisVarType);

        dt.fitModel();
        double[] singlePredictions = dt.predict(kyphosisXYTest);

        RandomTrees rf = new RandomTrees(dt);
        rf.setRandomObject(new Random(123457));
        rf.setNumberOfRandomFeatures(2);

        double[] rfPredictions = rf.predict(kyphosisXYTest);

        new PrintMatrix("Kyphosis test data single tree predictions"
                + " on the test data:").print(singlePredictions);

        new PrintMatrix("Kyphosis test data random forest predictions"
                + " on the test data:").print(rfPredictions);
    }
}
```

Output

```
Kyphosis test data single tree predictions on the test data:
   0
0  0
1  0
2  0
3  1
4  0
5  0
6  0
7  0
8  0
9  0

Kyphosis test data random forest predictions on the test data:
   0
0  0
1  1
2  0
3  0
```

```
4  0
5  0
6  0
7  0
8  0
9  0
```

Example 2: RandomTrees

This example builds a random forest with ALACART decision trees. Classification errors for the Iris data are shown using the out-of-bag predictions. An out-of-bag prediction for an observation is generated by models fitted to bootstrap samples that do not include the observation.

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

public class RandomTreesEx2 {

    public static void main(String[] args) throws Exception {
        double[][] irisFisherData = {
            {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, .1}, {1.0, 5.0, 3.2, 1.2, .2},
            {1.0, 5.5, 3.5, 1.3, .2}, {1.0, 4.9, 3.1, 1.5, .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 irisResponseIdx = 0;
    DecisionTree.VariableType[] irisVarType = {
        DecisionTree.VariableType.CATEGORICAL,
        DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
        DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
        DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
        DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS
    };
    int n = irisFisherData.length, m = irisFisherData[0].length;
    double[][] irisXY = new double[n][m];
    double[] irisY = new double[n];
```

```
        for (int i = 0; i < n; i++) {
            System.arraycopy(irisFisherData[i], 0, irisXY[i], 0, m);
            irisXY[i][0]--;
            irisY[i] = irisXY[i][0];
        }

        RandomTrees rf
                = new RandomTrees(irisXY, irisResponseIdx, irisVarType);

        rf.setRandomObject(new Random(123457));
        rf.fitModel();

        double outOfBagError = rf.getOutOfBagPredictionError();
        double[] outOfBagPredictions = rf.getOutOfBagPredictions();
        int[][] rfClassErrors = rf.getClassErrors(irisY, outOfBagPredictions);

        System.out.println("RandomTrees ALACART out of bag error rate: "
                + outOfBagError + "\n");

        new PrintMatrix("RandomTrees ALACART out of bag errors:").
                print(rfClassErrors);
    }
}
```

Output

```
RandomTrees ALACART out of bag error rate: 0.04666666666666667

RandomTrees ALACART out of bag errors:
   0   1
0  0   50
1  4   50
2  3   50
3  7  150
```

Example 3: RandomTrees

This example builds a random forest with C45 decision trees on simulated categorical data. A variable importance measure based on the change in predictive accuracy is highest for variable 0.

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

public class RandomTreesEx3 {

    public static void main(String[] args) throws Exception {
        double[][] simOXY = {
            {2, 25.92869, 0, 0}, {1, 51.63245, 1, 1}, {1, 25.78432, 0, 2},
            {0, 39.37948, 0, 3}, {2, 24.65058, 0, 2}, {2, 45.20084, 0, 2},
            {2, 52.67960, 1, 3}, {1, 44.28342, 1, 3}, {2, 40.63523, 1, 3},
```

```
            {2, 51.76094, 0, 3}, {2, 26.30368, 0, 1}, {2, 20.70230, 1, 0},
            {2, 38.74273, 1, 3}, {2, 19.47333, 0, 0}, {1, 26.42211, 0, 0},
            {2, 37.05986, 1, 0}, {1, 51.67043, 1, 3}, {0, 42.40156, 0, 3},
            {2, 33.90027, 1, 2}, {1, 35.43282, 0, 0}, {1, 44.30369, 0, 1},
            {0, 46.72387, 0, 2}, {1, 46.99262, 0, 2}, {0, 36.05923, 0, 3},
            {2, 36.83197, 1, 1}, {1, 61.66257, 1, 2}, {0, 25.67714, 0, 3},
            {1, 39.08567, 1, 0}, {0, 48.84341, 1, 1}, {1, 39.34391, 0, 3},
            {2, 24.73522, 0, 2}, {1, 50.55251, 1, 3}, {0, 31.34263, 1, 3},
            {1, 27.15795, 1, 0}, {0, 31.72685, 0, 2}, {0, 25.00408, 0, 3},
            {1, 26.35457, 1, 3}, {2, 38.12343, 0, 1}, {0, 49.94030, 0, 2},
            {1, 42.45779, 1, 3}, {0, 38.80948, 1, 1}, {0, 43.22799, 1, 1},
            {0, 41.87624, 0, 3}, {2, 48.07820, 0, 2}, {0, 43.23673, 1, 0},
            {2, 39.41294, 0, 3}, {1, 23.93346, 0, 2}, {2, 42.84130, 1, 3},
            {2, 30.40669, 0, 1}, {0, 37.77389, 0, 2}
        };

        DecisionTree.VariableType[] sim0VarType = {
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.QUANTITATIVE_CONTINUOUS,
            DecisionTree.VariableType.CATEGORICAL,
            DecisionTree.VariableType.CATEGORICAL
        };

        int sim0ResponseIdx = 0, n = sim0XY.length;

        double[] knownY = new double[n];
        for (int i = 0; i < n; i++) {
            knownY[i] = sim0XY[i][sim0ResponseIdx];
        }

        C45 dt = new C45(sim0XY, sim0ResponseIdx, sim0VarType);
        RandomTrees rf = new RandomTrees(dt);
        rf.setRandomObject(new Random(123457));
        rf.setCalculateVariableImportance(true);
        rf.fitModel();

        double[] outOfBagPredictions = rf.getOutOfBagPredictions();
        int[][] classErrors = rf.getClassErrors(knownY, outOfBagPredictions);
        double[] variableImportance = rf.getVariableImportance();

        new PrintMatrix("C45 Random Forest class errors:").
                print(classErrors);

        new PrintMatrix("C45 Random Forest variable importance:").
                print(variableImportance);
    }
}
```

Output

```
C45 Random Forest class errors:
   0   1
0  13  15
1  16  16
2  13  19
```

```
3  42  50

C45 Random Forest variable importance:
     0
0  -0.018
1  -0.002
2  -0.007
```

RandomTrees.ReflectiveOperationException class

```
static public class
com.imsl.datamining.decisionTree.RandomTrees.ReflectiveOperationException
extends com.imsl.datamining.PredictiveModel.PredictiveModelException
```

Class that wraps exceptions thrown by reflective operations in core reflection.

Constructors

RandomTrees.ReflectiveOperationException

`public RandomTrees.ReflectiveOperationException(String message)`

Description

Constructs a `ReflectiveOperationException` and issues the specified message.

Parameter

`message` – a `String` that contains a message to be issued when the exception occurs

RandomTrees.ReflectiveOperationException

`public RandomTrees.ReflectiveOperationException(String key, Object[] arguments)`

Description

Constructs a `ReflectiveOperationException` with the specified detail message. The error message string is in a resource bundle, ErrorMessages.

Parameters

`key` – a `String` that contains the key of an error message in the resource bundle

`arguments` – an `Object` array containing arguments used within the error message specified by the `key`

Chapter 33: Maximum Likelihood Estimation

Types

class MaximumLikelihoodEstimation 2312
class ProbabilityDistribution 2321
interface PDFGradientInterface 2324
interface PDFHessianInterface 2324
interface ClosedFormMaximumLikelihoodInterface 2325
interface MethodOfMomentsInterface 2326
class BetaPD 2326
class ContinuousUniformPD 2329
class ExponentialPD 2333
class GammaPD 2336
class NormalPD 2339

Usage Notes

The classes and methods in this chapter are for modeling and estimating univariate probability distributions.

Maximum Likelihood Estimation

Suppose we have the random sample

$$\{x_i, i = 1, 2, \ldots, N\}$$

from a probability distribution having a density function $f(x;\theta)$ which depends on a vector of unknown parameters, θ. The likelihood function given the sample is the product of probability densities evaluated at the sample points

$$L(\theta, \{x_i, i = 1, 2, \ldots, N\}) = \prod_{i=1,\ldots,N} f(x_i;\theta)$$

The estimator

$$\hat{\theta} = \mathrm{argmax}_{\theta} L(\theta, \{x_i\})$$

is the maximum likelihood estimator (MLE) for θ. The problem is usually expressed in terms of the log-likelihood:

$$\hat{\theta} = \text{argmax}_\theta \log(L(\theta, \{x_i\}))$$

$$= \text{argmax}_\theta \sum_i^N \log(f(x_i; \theta))$$

Or, equivalently, the problem is often expressed as a minimization problem:

$$\hat{\theta} = \text{argmin}_\theta - \sum_i^N \log(f(x_i; \theta))$$

The likelihood problem is a constrained non-linear optimization problem, where the constraints are determined by the domain of θ. Numerical optimization is usually successful in solving the likelihood problem for densities having first and second partial derivatives with respect to θ. Furthermore, under some general regularity conditions, the maximum likelihood estimator is consistent and asymptotically normally distributed with mean equal to the true value of the parameter θ_0 and variance-covariance matrix equal to the inverse *Fisher's Information* matrix evaluated at the true value of the parameter:

$$Var(\hat{\theta}) = I(\theta_0)^{-1} = -E_{\theta_0}\left[\frac{\partial^2 \log L}{\partial \theta^2}\right]^{-1}$$

The variance is approximated by the negative inverse hessian of the log-likelihood evaluated at the maximum likelihood estimate.

$$Var(\hat{\theta}) \approx -\left[\frac{\partial^2 \log L}{\partial \theta^2}\right]_{\hat{\theta}}^{-1}$$

See Kendall and Stuart (1979) for further details on the theory of the maximum likelihood.

MaximumLikelihoodEstimation class

```
public class com.imsl.stat.distributions.MaximumLikelihoodEstimation implements
Serializable, Cloneable
```

Maximum likelihood parameter estimation

Constructor

MaximumLikelihoodEstimation

```
public MaximumLikelihoodEstimation(double[] x, ProbabilityDistribution pd,
double[] guess)
```

Description

Constructor for maximum likelihood estimation

Parameters

x – a double array containing the sample observations

pd – an instance of ProbabilityDistribution

guess – a double array or a comma-separated list of doubles giving the starting values for the parameters
Note: The argument guess is a variable length argument list (varargs).

Methods

compute

```
public void compute() throws MinConNLP.ConstraintEvaluationException,
MinConNLP.ObjectiveEvaluationException, MinConNLP.WorkingSetSingularException,
MinConNLP.QPInfeasibleException,
MinConNLP.PenaltyFunctionPointInfeasibleException,
MinConNLP.LimitingAccuracyException, MinConNLP.TooManyIterationsException,
MinConNLP.BadInitialGuessException, MinConNLP.IllConditionedException,
MinConNLP.SingularException, MinConNLP.LinearlyDependentGradientsException,
MinConNLP.NoAcceptableStepsizeException,
MinConNLP.TerminationCriteriaNotSatisfiedException
```

Description

Computes the maximum likelihood estimates.

Exceptions

com.imsl.math.MinConNLP.ConstraintEvaluationException Constraint evaluation returns an error with current point.

com.imsl.math.MinConNLP.ObjectiveEvaluationException Objective evaluation returns an error with current point.

com.imsl.math.MinConNLP.WorkingSetSingularException Working set is singular in dual extended QP.

com.imsl.math.MinConNLP.QPInfeasibleException QP problem seemingly infeasible.

com.imsl.math.MinConNLP.PenaltyFunctionPointInfeasibleException Penalty function point infeasible.

com.imsl.math.MinConNLP.LimitingAccuracyException Limiting accuracy reached for a singular problem.

com.imsl.math.MinConNLP.TooManyIterationsException Maximum number of iterations exceeded.

com.imsl.math.MinConNLP.BadInitialGuessException Penalty function point infeasible for original problem. Try new initial guess.

`com.imsl.math.MinConNLP.IllConditionedException` Problem is singular or ill-conditioned.

`com.imsl.math.MinConNLP.SingularException` Problem is singular.

`com.imsl.math.MinConNLP.LinearlyDependentGradientsException` Working set gradients are linearly dependent.

`com.imsl.math.MinConNLP.NoAcceptableStepsizeException` No acceptable stepsize in [SIGMA,SIGLA].

`com.imsl.math.MinConNLP.TerminationCriteriaNotSatisfiedException` Termination criteria are not satisfied.

getEstimates

```
public double[] getEstimates()
```

Description

Returns the parameter estimates.

Returns

a `double` array containing the parameter estimates

getHessian

```
public double[][] getHessian()
```

Description

Returns the Hessian of the log-likelihood function evaluated at the current parameter estimates.

Returns

a `double` matrix containing the Hessian matrix

getLogLikelihood

```
public double getLogLikelihood(double[] x, double[] params)
```

Description

Returns the log-likelihood.

Note that this method is for convenience and does not use any of the member data or parameters. The user supplies the arguments.

Parameters

`x` – a `double` array containing sample data

`params` – a `double` array or comma separated list of doubles containing the parameter values

Returns

a `double`, the log-likelihood evaluated at the given data and parameter values

getMinusLogLikelihood

```
public double getMinusLogLikelihood()
```

Description

Returns minus the log-likelihood evaluated at the parameter estimates.

This method returns the value of the objective function, the minimum of minus the log-likelihood.

Returns

a `double`, minus the log-likelihood

getStandardErrors

```
public double[] getStandardErrors() throws SingularMatrixException
```

Description

Returns the approximate standard errors of the maximum likelihood estimates.

Returns

a `double` array containing the standard errors

Exception

`SingularMatrixException` The matrix is singular.

getVarCov

```
public double[][] getVarCov() throws SingularMatrixException
```

Description

Returns the approximate variance-covariance matrix of the maximum likelihood estimates. The approximation is the negative inverse Hessian of the log-likelihood.

Returns

a `double` matrix containing the approximate variance-covariance matrix

Exception

`SingularMatrixException` The matrix is singular.

setClosedForm

```
public void setClosedForm(boolean cf)
```

Description

Sets the flag indicating whether or not the closed form solution should be used.

Parameter

`cf` – a `boolean`. When true, the `com.imsl.stat.distributions.MaximumLikelihoodEstimation.compute` (p. 2313) method returns the closed form solution if the `ProbabilityDistribution` is a `ClosedFormMaximumLikelihoodInterface` object. The option is ignored if that is not the case.

Default: `cf` = `false`.

setExact

```
public void setExact(boolean exact)
```

Description

Sets the flag indicating whether or not the PDF supplies the exact gradient and Hessian.

Parameter

`exact` – a `boolean`. When `true`, the function expects the `pdf` to supply the analytic gradient and Hessian calculation.

Default: `exact` = `false`.

setGuess

```
public void setGuess(double[] guess)
```

Description

Sets the guess or starting values of the parameters.

Parameter

`guess` – a `double` array of the same length as the parameters containing proper starting values for the optimization. `guess` may also be a comma separated list of doubles giving the parameter values.

Note: The argument `guess` is a variable length argument list (varargs).

setSample

```
public void setSample(double[] x)
```

Description

Sets the sample data to use in the estimation procedure.

Parameter

`x` – a `double` array containing sample observations of the random variable

Example 1: Maximum Likelihood Estimation

This example obtains maximum likelihood estimates for the parameters of a beta distribution.

```
import com.imsl.math.*;
import com.imsl.stat.distributions.*;
import java.text.DecimalFormat;

public class MaximumLikelihoodEstimationEx1 {

    public static void main(String[] args) throws Exception {
        double[] x = {
            0.12396e0, 0.58037e0, 0.28837e0, 0.38195e0,
            0.44387e0, 0.17680e0, 0.22661e0, 0.55939e0,
            0.41646e0, 0.33781e0, 0.63768e0, 0.81501e0,
            0.55952e0, 0.50559e0, 0.39315e0, 0.67901e0,
            0.66032e0, 0.52279e0, 0.53164e0, 0.19165e0,
            0.83432e0, 0.16578e0, 0.30614e0, 0.57622e0,
            0.24793e0, 0.55212e0, 0.32037e0, 0.62443e0,
            0.54663e0, 0.56399e0, 0.61369e0, 0.62853e0,
            0.47957e0, 0.57208e0, 0.44813e0, 0.69026e0,
```

```
            0.73784e0, 0.28413e0, 0.18926e0, 0.58319e0,
            0.28740e0, 0.61739e0, 0.32052e0, 0.29761e0,
            0.68279e0, 0.75275e0, 0.48364e0, 0.56711e0,
            0.72584e0, 0.32971e0, 0.86950e0, 0.79991e0,
            0.52609e0, 0.46006e0, 0.52828e0, 0.27842e0,
            0.58179e0, 0.37934e0, 0.29819e0, 0.53552e0,
            0.46476e0, 0.22105e0, 0.36031e0, 0.49400e0,
            0.41216e0, 0.34985e0, 0.26751e0, 0.57636e0,
            0.54788e0, 0.48447e0, 0.75758e0, 0.60260e0,
            0.56603e0, 0.57345e0, 0.35777e0, 0.79578e0,
            0.46025e0, 0.57042e0, 0.31177e0, 0.56309e0,
            0.52548e0, 0.69891e0, 0.27270e0, 0.38819e0,
            0.73049e0, 0.54734e0, 0.50803e0, 0.48510e0,
            0.87042e0, 0.52287e0, 0.67998e0, 0.40186e0,
            0.48845e0, 0.64425e0, 0.65228e0, 0.45132e0,
            0.23021e0, 0.39449e0, 0.70417e0, 0.31853e0
        };

        BetaPD beta = new BetaPD();

        MaximumLikelihoodEstimation mle
                = new MaximumLikelihoodEstimation(x, beta, 1, 1);

        mle.compute();

        double[] parameterEstimates = mle.getEstimates();
        double[][] hessian = mle.getHessian();
        double[] stdErrs = mle.getStandardErrors();
        double mLogLikelihood = mle.getMinusLogLikelihood();

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new DecimalFormat("0.00000000"));

        PrintMatrix pm = new PrintMatrix("MLEs");
        pm.print(pmf, parameterEstimates);

        pm.setTitle("hessian");
        pm.print(pmf, hessian);

        pm.setTitle("standard errors");
        pm.print(pmf, stdErrs);

        System.out.printf("Minus log likelihood: %5.2f\n", mLogLikelihood);
    }
}
```

Output

```
     MLEs
       0
0  3.67112042
1  3.73274034

            hessian
        0             1
```

```
0  -16.82252107   14.45938489
1   14.45938489  -16.23511103

standard errors
       0
0  0.50349626
1  0.51252395

Minus log likelihood: -35.22
```

Example 2: Maximum Likelihood Estimation

This example obtains maximum likelihood estimates for the parameters of a gamma distribution.

```
import com.imsl.math.*;
import com.imsl.stat.distributions.*;
import java.text.DecimalFormat;

public class MaximumLikelihoodEstimationEx2 {

    public static void main(String[] args) throws Exception {
        double x[] = {
            1.015998e0, 0.2677489e0, 2.198958e0, 1.784697e0, 0.691063e0,
            0.992409e0, 0.5466309e0, 1.154601e0, 2.613015e0, 1.219826e0,
            1.158957e0, 3.933280e0, 0.2462264e0, 3.946624e0, 4.184852e0,
            1.148567e0, 1.798003e0, 1.437205e0, 2.383511e0, 2.030147e0,
            0.7575e0, 0.903873e0, 0.5005532e0, 0.296348e0, 1.564283e0,
            2.787072e0, 0.3472906e0, 2.228984e0, 2.328759e0, 4.634677e0,
            1.41974e0, 0.5815979e0, 2.654966e0, 1.719702e0, 1.668744e0,
            2.203117e0, 0.7410172e0, 0.5147782e0, 0.4841398e0, 2.745041e0,
            1.455526e0, 0.7109387e0, 0.967477e0, 0.876192e0, 0.4505406e0,
            0.3986173e0, 0.4707685e0, 0.301807e0, 0.624848e0, 4.325418e0,
            1.410972e0, 2.630923e0, 0.4841873e0, 0.1835388e0, 0.2373447e0,
            0.5298828e0, 1.323294e0, 1.672188e0, 0.009495691e0, 1.832066e0,
            1.210357e0, 1.133037e0, 1.23147e0, 0.379497e0, 2.001283e0,
            0.5448404e0, 0.1165402e0, 1.834957e0, 1.530335e0, 2.550148e0,
            1.240445e0, 1.058339e0, 1.613384e0, 1.708352e0, 1.134821e0,
            0.6249367e0, 0.8273115e0, 0.9793533e0, 0.05378025e0,
            0.6828388e0, 0.9259066e0, 0.8168141e0, 3.910878e0, 1.88685e0,
            2.836617e0, 1.644921e0, 0.863347e0, 2.128436e0, 2.710456e0,
            4.747736e0, 0.823015e0, 1.279859e0, 2.991689e0, 2.483813e0,
            0.0610894e0, 0.5572344e0, 0.07148165e0, 0.3847188e0,
            1.322698e0, 0.3358647e0
        };

        GammaPD gamma = new GammaPD();

        MaximumLikelihoodEstimation mle
                = new MaximumLikelihoodEstimation(x, gamma, 3, 1);

        mle.compute();

        double[] parameterEstimates = mle.getEstimates();
        double[][] hessian = mle.getHessian();
```

```
        double[] stdErrs = mle.getStandardErrors();
        double mLogLikelihood = mle.getMinusLogLikelihood();

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new DecimalFormat("0.00000000"));

        PrintMatrix pm = new PrintMatrix("MLEs");
        pm.print(pmf, parameterEstimates);

        pm.setTitle("hessian");
        pm.print(pmf, hessian);

        pm.setTitle("standard errors");
        pm.print(pmf, stdErrs);

        System.out.printf("Minus log likelihood: %5.2f\n", mLogLikelihood);
    }
}
```

Output

```
     MLEs
       0
0  1.47875577
1  0.96006357

             hessian
         0              1
0   -95.27326615  -104.15984884
1  -104.15984884  -160.43415880

standard errors
       0
0  0.19017869
1  0.14655443

Minus log likelihood: 130.88
```

Example 3: Maximum Likelihood Estimation

The MLEs for the normal distribution can be derived analytically and are well known to be the sample mean and the sample standard deviation (biased version). This example compares the numerical solution to the analytical solution.

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

public class MaximumLikelihoodEstimationEx3 {

    public static void main(String[] args) throws Exception {
        int n = 100;
        double[] sample = new double[n];
```

```
        double mean = 25;
        double stdev = 1.5;
        Random rand = new Random(123457);

        for (int i = 0; i < n; i++) {
            sample[i] = stdev * rand.nextNormal() + mean;
        }

        NormalPD norm = new NormalPD();
        double[] exactMLEs = norm.getMLEs(sample);

        MaximumLikelihoodEstimation mle
                = new MaximumLikelihoodEstimation(sample, norm, 2, 1);

        mle.compute();

        double[] parameterEstimates = mle.getEstimates();
        double[][] hessian = mle.getHessian();
        double[] stdErrs = mle.getStandardErrors();
        double mLogLikelihood = mle.getMinusLogLikelihood();

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new java.text.DecimalFormat("0.0000000"));

        PrintMatrix pm = new PrintMatrix("Analytic MLEs");
        pm.print(pmf, exactMLEs);

        pm.setTitle("Numerical MLEs");
        pm.print(pmf, parameterEstimates);

        pm.setTitle("hessian");
        pm.print(pmf, hessian);

        pm.setTitle("standard errors");
        pm.print(pmf, stdErrs);

        System.out.printf("Minus log likelihood: %5.2f\n", mLogLikelihood);
    }
}
```

Output

```
 Analytic MLEs
       0
0  25.0062530
1   1.6472313

Numerical MLEs
       0
0  25.0062531
1   1.6472315

           hessian
        0           1
0  -36.8545110    0.0000055
```

```
1    0.0000055  -73.7091163

standard errors
       0
0  0.1647232
1  0.1164768

Minus log likelihood: 191.80
```

ProbabilityDistribution class

```
abstract public class com.imsl.stat.distributions.ProbabilityDistribution
implements Serializable, Cloneable
```

The ProbabilityDistribution abstract class defines members and methods common to univariate probability distributions and useful in parameter estimation.

Constructor

ProbabilityDistribution

```
protected ProbabilityDistribution(int numberOfParameters)
```

Description

Constructor for the probability distribution

Parameter

`numberOfParameters` – an `int` specifying the number of parameters
Every subclass (specific probability distribution) must set the number of parameters.

Methods

getNumberOfParameters

```
public int getNumberOfParameters()
```

Description

Returns the number of parameters of the probability distribution.

Returns

an `int`, the number of parameters

getPDFGradientApproximation

```
public double[] getPDFGradientApproximation(double x, double[] params)
```

Description

Returns the approximate gradient of the probability density function, `pdf`.

Parameters

`x` – a `double` value

`params` – a `double` array or a comma-separated list of doubles giving the values for the parameters
Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` array containing the gradient approximation given the parameter values and evaluated at *X*=`x`

getPDFHessianApproximation

```
public double[][] getPDFHessianApproximation(double x, double[] params)
```

Description

Returns the approximate hessian of the probability density function, `pdf`.

Parameters

`x` – a `double` value

`params` – a `double` array or a comma-separated list of doubles giving the values for the parameters
Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` matrix equal to the second partial derivatives of the probability density function with respect to the parameters evaluated at *X*=`x`

getParameterLowerBounds

```
abstract public double[] getParameterLowerBounds()
```

Description

Returns the lower bounds of the parameters.

Each `ProbabilityDistribution` subclass must override this method.

Returns

a `double` array containing the lower bounds of the parameters

getParameterUpperBounds

```
abstract public double[] getParameterUpperBounds()
```

Description

Returns the upper bounds of the parameters.

Each `ProbabilityDistribution` subclass must override this method.

Returns

a `double` array containing the upper bounds of the parameters

getRangeOfX

```
public double[] getRangeOfX()
```

Description

Returns the proper range of the random variable having the current probability distribution.

Returns

a `double` array containing the lower bound (at index 0) and upper bound (at index 1) for the random variable `X`

pdf

```
abstract public double pdf(double x, double[] params)
```

Description

Returns the value of the probability density function.

Each `ProbabilityDistribution` subclass must override this method.

Parameters

`x` – a `double` value

`params` – a `double` array or a comma-separated list of doubles giving the values for the parameters Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` value equal to the probability density function given the parameters evaluated at X=`x`

setRangeOfX

```
public void setRangeOfX(double[] range)
```

Description

Sets the proper range of the random variable having the current probability distribution.

Parameter

`range` – a `double` array containing the lower bound (at index 0) and the upper bound (at index 1) for the random variable

Default: `range[0]=Double.NEGATIVE_INFINITY` and `range[1]=Double.POSITIVE_INFINITY`.

PDFGradientInterface interface

`public interface com.imsl.stat.distributions.PDFGradientInterface`

A public interface for probability distributions that provide a method to calculate the gradient of the density function

Method

getPDFGradient

`public double[] getPDFGradient(double x, double[] params)`

Description

Returns the gradient of the probability density function.

Parameters

`x` – a `double` value

`params` – a `double` array or a comma-separated list of doubles giving the values of the parameters Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` array equal to the first partial derivatives of the probability density function with respect to the parameters evaluated at *X*=`x`

PDFHessianInterface interface

`public interface com.imsl.stat.distributions.PDFHessianInterface implements com.imsl.stat.distributions.PDFGradientInterface`

A public interface for probability distributions that provide methods to calculate the gradient and hessian of the density function

Method

getPDFHessian

`public double[][] getPDFHessian(double x, double[] params)`

Description

Returns the hessian of the probability density function.

Parameters

x – a double value in the range of the random variable

params – a double array or a comma-separated list of doubles giving the values of the parameters
Note: The argument params is a variable length argument list (varargs).

Returns

a double matrix equal to the second partial derivatives of the probability density function with respect to the parameters evaluated at X=x

ClosedFormMaximumLikelihoodInterface interface

```
public interface
com.imsl.stat.distributions.ClosedFormMaximumLikelihoodInterface
```

A public interface for probability distributions that provide a method for a closed form solution of the maximum likelihood function

Methods

getClosedFormMLE

```
public double[] getClosedFormMLE(double[] x)
```

Description

Returns the maximum likelihood estimates (MLEs).

Parameter

x – a double array containing the data

Returns

a double array containing distribution parameters

getClosedFormMlStandardError

```
public double[] getClosedFormMlStandardError(double[] x)
```

Description

Returns the standard error based on the closed form solution of the maximum liklihood for the sample data.

Parameter

`x` – a `double` array containing the data

Returns

a `double` array containing the standard errors

MethodOfMomentsInterface interface

```
public interface com.imsl.stat.distributions.MethodOfMomentsInterface
```

A public interface for probability distributions that provide a method for calculating the method of moments for the density function

Method

getMethodOfMomentsEstimates

```
public double[] getMethodOfMomentsEstimates(double[] x)
```

Description

Returns the method-of-moments estimates given the sample data.

Parameter

`x` – a `double` array containing the data

Returns

a `double` array containing method-of-moments estimates for the parameters of the distribution

BetaPD class

```
public class com.imsl.stat.distributions.BetaPD extends
com.imsl.stat.distributions.ProbabilityDistribution implements Serializable,
Cloneable
```

The beta probability distribution

Constructor

BetaPD

```
public BetaPD()
```

Description

Constructor for the beta probability distribution

Methods

getParameterLowerBounds

```
public double[] getParameterLowerBounds()
```

Description

Returns the lower bounds for the two shape parameters of the beta distribution.

Returns

a `double` array containing the lower bounds

getParameterUpperBounds

```
public double[] getParameterUpperBounds()
```

Description

Returns the upper bounds for the two shape parameters of the beta distribution.

Returns

a `double` array containing the upper bounds

pdf

```
public double pdf(double x, double[] params)
```

Description

Returns the value of the beta probability density function.

The probability density function of the beta distribution is

$$f(x;a,b) = x^{a-1}(1-x)^{b-1}\frac{1}{B(a,b)}$$

where a>0 and b>0 are shape parameters and the beta function is

$$B(a,b) = \frac{\Gamma(a,b)}{\Gamma(a)\Gamma(b)}$$

Parameters

x – a double value in the range of X

params – a double array containing values of the parameters, with params[0]=*"a"* and params[1]=*"b"*. The argument can also be a comma-delimited list *"a"*, *"b"* or params[0],params[1].

Note: The argument params is a variable length argument list (varargs).

Returns

a double value equal to the probability density at x given the parameter values

Example: beta probability distribution

This example evaluates the beta density, gradient, and hessian for a small sample of data.

```
import com.imsl.math.*;
import com.imsl.stat.distributions.BetaPD;
import java.text.DecimalFormat;

public class BetaPDEx1 {

    public static void main(String[] args) throws Exception {

        double[] x = {
            0.12396e0, 0.58037e0, 0.28837e0, 0.38195e0,
            0.44387e0, 0.17680e0, 0.22661e0, 0.55939e0
        };

        double[] values = new double[x.length];
        double a = 1.2;
        double b = 2.0;

        BetaPD beta = new BetaPD();

        for (int i = 0; i < x.length; i++) {
            values[i] = beta.pdf(x[i], a, b);
        }

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new DecimalFormat("0.00000000"));

        PrintMatrix pm = new PrintMatrix("Beta(1.2,2.0) pdf values at x=");
        pm.print(x);

        pm.setTitle("f(x;a,b)=");
        pm.print(values);

        double[] gradient = beta.getPDFGradientApproximation(x[0], a, b);

        pm.setTitle("Gradient f(x;a,b) at x[0]");
        pm.print(gradient);

        double[][] hessian = beta.getPDFHessianApproximation(x[0], a, b);
```

```
        pm.setTitle("Hessian of f(x;a,b) at x[0]");
        pm.print(pmf, hessian);
    }
}
```

Output

```
Beta(1.2,2.0) pdf values at x=
     0
0  0.124
1  0.58
2  0.288
3  0.382
4  0.444
5  0.177
6  0.227
7  0.559

f(x;a,b)=
     0
0  1.523
1  0.994
2  1.465
3  1.346
4  1.248
5  1.537
6  1.517
7  1.036

Gradient f(x;a,b) at x[0]
     0
0  -1.219
1   0.676

 Hessian of f(x;a,b) at x[0]
        0            1
0  -0.39787363   0.01734585
1   0.01734585  -0.12450620
```

ContinuousUniformPD class

```
public class com.imsl.stat.distributions.ContinuousUniformPD extends
com.imsl.stat.distributions.ProbabilityDistribution implements Serializable,
Cloneable, com.imsl.stat.distributions.PDFHessianInterface,
com.imsl.stat.distributions.ClosedFormMaximumLikelihoodInterface,
com.imsl.stat.distributions.MethodOfMomentsInterface
```

The continuous uniform probability distribution.

Constructor

ContinuousUniformPD

`public ContinuousUniformPD()`

Description

Constructs a continuous uniform probability distribution.

Methods

getClosedFormMLE

`public double[] getClosedFormMLE(double[] x)`

Description

Returns the closed form determination of the maximum likelihood values for the continuous uniform distribution. These are the maximum likelihood estimates for the sample minimum and maximum of the continuous uniform distribution, given the data.

Parameter

`x` – a `double` array containing the data

Returns

a `double` array containing distribution parameters

getClosedFormMlStandardError

`public double[] getClosedFormMlStandardError(double[] x)`

Description

Returns the standard error based on the closed form solution of the maximum likelihood for the sample data.

Parameter

`x` – a `double` array containing the data

Returns

a `double` array containing the standard errors

getPDFGradient

`public double[] getPDFGradient(double x, double[] params)`

Description

Returns the analytic gradient of the continuous uniform PDF evaluated at a point, `x`.

Parameters

`x` – a `double` value. The function is undefined when `x` = *a* or `x` = *b*.

`params` – a `double` array containing values of the parameters. `params[0]` (*a*) must be less than `param[1]` (*b*).

Returns

a `double` array containing the first partial derivative of the parameters

getPDFHessian

```
public double[][] getPDFHessian(double x, double[] params)
```

Description

Returns the analytic Hessian matrix of the continuous uniform pdf evaluated at a point, `x`.

Parameters

`x` – a `double` value

`params` – a `double` array containing values of the parameters

Returns

a `double` matrix containing the second partial derivatives of the parameters

getParameterLowerBounds

```
public double[] getParameterLowerBounds()
```

Description

Returns the lower bounds of the continuous uniform distribution.

Returns

a `double` array of length 2 containing the lower bounds. *a* and *b* can be any real number.

getParameterUpperBounds

```
public double[] getParameterUpperBounds()
```

Description

Returns the upper bounds of the continuous uniform distribution.

Returns

a `double` array of length 2 containing the upper bounds. *a* and *b* can be any real number.

pdf

```
public double pdf(double x, double[] params)
```

Description

Returns the value of the continuous uniform probability density function.

The probability density function of the continuous uniform distribution is

$$f(x|a,b) = \begin{cases} \frac{1}{b-a}, & \text{for} \quad a \le x \le b \\ 0, & \text{for} \quad x < a \text{ or } x > b \end{cases}$$

Parameters

x – a double indicating where the pdf is to be evaluated

params – a double specifying the probability of success. params[0] (a) must be less than param[1] (b) or a probability of 0.0 is returned.

Returns

a double value equal to the probability density at x given the parameter value

Example: Continuous Uniform Probability Distribution

This example obtains the Continuous Uniform probability density, gradient and Hessian values at a point.

```
import com.imsl.stat.distributions.ContinuousUniformPD;

public class ContinuousUniformPDEx1 {

    public static void main(String[] args) {

        double x = 0.5;
        ContinuousUniformPD cupd = new ContinuousUniformPD();

        double pdf = cupd.pdf(x, 0.0, 1.0);
        double[] gradient = cupd.getPDFGradient(x, 0.0, 1.0);
        double[][] hessian = cupd.getPDFHessian(x, 0.0, 1.0);

        System.out.printf("Probability density at %.3f with a=0.0 and b=1.0 ="
                + " %.3f\n\n", x, pdf);
        System.out.printf("Probability density gradient at %.3f with a=0.0 and"
                + " b=1.0 = {%.3f, %.3f}\n\n", x, gradient[0], gradient[1]);
        System.out.printf("Probability density Hessian at %.3f with a=0.0\nand"
                + " b=1.0 = {{%.3f, %.3f}, {%.3f, %.3f}}\n\n", x, hessian[0][0],
                hessian[0][1], hessian[1][0], hessian[1][1]);
    }
}
```

Output

```
Probability density at 0.500 with a=0.0 and b=1.0 = 1.000

Probability density gradient at 0.500 with a=0.0 and b=1.0 = {1.000, -1.000}

Probability density Hessian at 0.500 with a=0.0
```

```
and b=1.0 = {{2.000, -2.000}, {-2.000, 2.000}}
```

ExponentialPD class

```
public class com.imsl.stat.distributions.ExponentialPD extends
com.imsl.stat.distributions.ProbabilityDistribution implements Serializable,
Cloneable, com.imsl.stat.distributions.PDFHessianInterface,
com.imsl.stat.distributions.ClosedFormMaximumLikelihoodInterface,
com.imsl.stat.distributions.MethodOfMomentsInterface
```

The exponential probability distribution.

Constructor

ExponentialPD

```
public ExponentialPD()
```

Description

Constructs an exponential probability distribution.

Methods

getClosedFormMLE

```
public double[] getClosedFormMLE(double[] x)
```

Description

Returns the closed form determination of the maximum likelihood values for the exponential distribution.

Parameter

x – a double array containing the data

Returns

a double array containing maximum likelihood values

getClosedFormMlStandardError

```
public double[] getClosedFormMlStandardError(double[] x)
```

Description

Returns the standard error based on the closed form solution of the maximum likelihood for the sample data.

Parameter

`x` – a `double` array containing the data

Returns

a `double` array containing the standard errors

getPDFGradient

`public double[] getPDFGradient(double x, double[] params)`

Description

Returns the analytic gradient of the exponential pdf evaluated at a point, `x`.

Parameters

`x` – a `double` value. `x` must be non negative.

`params` – a `double` array containing values of the parameters. `params` (b) is strictly positive.

Returns

a `double` array containing the first partial derivative of the parameters

getPDFHessian

`public double[][] getPDFHessian(double x, double[] params)`

Description

Returns the analytic Hessian matrix of the exponential pdf evaluated at a point, `x`.

Parameters

`x` – a `double` value. `x` must be non negative.

`params` – a `double` array containing values of the parameters. `params` (b) is strictly positive.

Returns

a `double` matrix containing the second partial derivatives of the parameters

getParameterLowerBounds

`public double[] getParameterLowerBounds()`

Description

Returns the lower bounds for the scale parameter of the exponential distribution.

Returns

a `double` array of length 1 containing the lower bound. b is strictly positive.

getParameterUpperBounds

`public double[] getParameterUpperBounds()`

Description

Returns the upper bounds for the scale parameter of the exponential distribution.

Returns

a `double` array of length 1 containing the upper bound. b is strictly positive.

pdf

`public double pdf(double x, double[] params)`

Description

Returns the value of the exponential probability density function.

The probability density function of the exponential distribution is

$$f(x|b) = \Gamma(x|1,b) = \frac{1}{b}e^{-\frac{x}{b}}$$

where b is a scale parameter.

Parameters

`x` – a `double` indicating where the pdf is to be evaluated. `x` must be non negative.

`params` – a `double` specifying the scale parameter. `params` (b) is strictly positive.

Returns

a `double` value equal to the probability density at `x` given the parameter value

Example: Exponential Probability Distribution

This example obtains the Exponential probability density, gradient and Hessian values at a point.

```
import com.imsl.stat.distributions.ExponentialPD;

public class ExponentialPDEx1 {

    public static void main(String[] args) {

        double x = 2.0 / (5.0 + 1.0);
        double scale = 3.0;
        ExponentialPD epd = new ExponentialPD();

        double pdf = epd.pdf(x, scale);
        double[] gradient = epd.getPDFGradient(x, scale);
        double[][] hessian = epd.getPDFHessian(x, scale);

        System.out.printf("Probability density at %.3f with scale value of 3.0"
                + " = %.3f\n\n", x, pdf);
        System.out.printf("Probability density gradient at %.3f with scale"
                + " value of 3.0 = {%.3f}\n\n", x, gradient[0]);
        System.out.printf("Probability density Hessian at %.3f with scale"
                + " value of 3.0 = {{%.3f}}\n\n", x, hessian[0][0]);
    }
}
```

Output

```
Probability density at 0.333 with scale value of 3.0 = 1.104

Probability density gradient at 0.333 with scale value of 3.0 = {0.000}

Probability density Hessian at 0.333 with scale value of 3.0 = {{-0.123}}
```

GammaPD class

```
public class com.imsl.stat.distributions.GammaPD extends
com.imsl.stat.distributions.ProbabilityDistribution implements Serializable,
Cloneable
```

The gamma probability distribution

Constructor

GammaPD

```
public GammaPD()
```

Description

Constructor for the gamma probability distribution

Methods

getParameterLowerBounds

```
public double[] getParameterLowerBounds()
```

Description

Returns the lower bounds for the shape and scale parameters of the gamma distribution.

Returns

a `double` array containing the lower bounds

getParameterUpperBounds

```
public double[] getParameterUpperBounds()
```

Description

Returns the upper bounds for the shape and scale parameters of the gamma distribution.

Returns

a `double` array containing the upper bounds

pdf

`public double pdf(double x, double[] params)`

Description

Returns the value of the gamma probability density function.

The probability density function of the gamma distribution is

$$f(x;a,b) = x^{a-1}\frac{1}{b^a\Gamma(a)}e^{-x/b}$$

where a is the shape parameter and b is the scale parameter.

Parameters

`x` – a `double` value in the range of `X`

`params` – a `double` array containing values of the parameters, with `params[0]`=“a” and `params[1]`=“b”. The argument can also be a comma-delimited list “a”, “b” or `params[0]`,`params[1]`.

Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` value equal to the probability density at `x` given the parameter values

Example: gamma probability distribution

This example evaluates the gamma density, gradient, and hessian for a small sample of data.

```
import com.imsl.math.*;
import com.imsl.stat.distributions.*;
import java.text.DecimalFormat;

public class GammaPDEx1 {

    public static void main(String[] args) throws Exception {

        double x[] = {
            1.015998e0, 0.2677489e0, 2.198958e0, 1.784697e0, 0.691063e0,
            0.992409e0, 0.5466309e0, 1.154601e0, 2.613015e0, 1.219826e0
        };

        double[] values = new double[x.length];
        double a = 3.0;
        double b = 2.0;

        GammaPD gamma = new GammaPD();
```

```
        for (int i = 0; i < x.length; i++) {
            values[i] = gamma.pdf(x[i], a, b);
        }

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new DecimalFormat("0.00000000"));

        PrintMatrix pm = new PrintMatrix("Gamma(3.0,2.0) pdf values at x=");
        pm.print(x);

        pm.setTitle("f(x;a,b)=");
        pm.print(values);

        double[] gradient = gamma.getPDFGradientApproximation(x[0], a, b);

        pm.setTitle("Gradient f(x;a,b) at x[0]");
        pm.print(gradient);

        double[][] hessian = gamma.getPDFHessianApproximation(x[0], a, b);

        pm.setTitle("Hessian of f(x;a,b) at x[0]");
        pm.print(pmf, hessian);
    }
}
```

Output

```
Gamma(3.0,2.0) pdf values at x=
     0
0  1.016
1  0.268
2  2.199
3  1.785
4  0.691
5  0.992
6  0.547
7  1.155
8  2.613
9  1.22

f(x;a,b)=
     0
0  0.039
1  0.004
2  0.101
3  0.082
4  0.021
5  0.037
6  0.014
7  0.047
8  0.116
9  0.051

Gradient f(x;a,b) at x[0]
```

```
     0
0  -0.062
1  -0.048

Hessian of f(x;a,b) at x[0]
       0           1
0  0.08405321  0.05798301
1  0.05798301  0.07952132
```

NormalPD class

`public class com.imsl.stat.distributions.NormalPD extends com.imsl.stat.distributions.ProbabilityDistribution implements com.imsl.stat.distributions.PDFHessianInterface, Serializable, Cloneable`

The normal (Gaussian) probability distribution

Constructor

NormalPD

`public NormalPD()`

Description

Constructor for the normal probability distribution

Methods

getMLEs

`public double[] getMLEs(double[] x)`

Description

Returns the mean and standard deviation of the sample data.

These are the maximum likelihood estimates for the mean and standard deviation of the Normal distribution, given the data.

Parameter

`x` – a `double` array containing the data

Returns

a `double` array containing the mean and standard deviation

getPDFGradient

`public double[] getPDFGradient(double x, double[] params)`

Description

Returns the analytic gradient of the normal pdf.

Parameters

`x` – a `double` value

`params` – a `double` array containing values of the parameters, `params[0]`="μ" and `params[1]`="σ". The argument can also be a comma-delimited list "μ", "σ" or `params[0]`, `params[1]`.
Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` array containing the partial derivatives of the pdf with respect to the parameters evaluated at `x` and the given parameter values

getPDFHessian

`public double[][] getPDFHessian(double x, double[] params)`

Description

Returns the analytic hessian matrix of the normal pdf evaluated at a point, `x`.

Parameters

`x` – a `double` value

`params` – a `double` array containing values of the parameters, `params[0]`="μ" and `params[1]`="σ". The argument can also be a comma-delimited list "μ", "σ" or `params[0]`, `params[1]`.
Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` matrix containing the second partial derivatives of the pdf with respect to the parameters evaluated at `x` and the given parameter values

getParameterLowerBounds

`public double[] getParameterLowerBounds()`

Description

Returns the lower bounds for the mean μ and standard deviation σ.

Returns

a `double` array containing the lower bounds

getParameterUpperBounds

`public double[] getParameterUpperBounds()`

Description

Returns the upper bounds for the mean μ and standard deviation σ.

Returns

a `double` array containing the upper bounds

pdf

`public double pdf(double x, double[] params)`

Description

Returns the value of the normal probability density function.

The probability density function for a normal distribution is given by

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ and $\sigma > 0$ are the mean and standard deviation of the random variable.

Parameters

`x` – a `double` value

`params` – a `double` array containing values of the parameters, `params[0]`="μ" and `params[1]`="σ". The argument can also be a comma-delimited list "μ", "σ" or `params[0]`, `params[1]`.

Note: The argument `params` is a variable length argument list (varargs).

Returns

a `double` value equal to the probability density at `x` given the parameter values

Example Normal Probability Distribution

This example evaluates the normal density, approximate gradient and hessian, and analytic gradient and hessian for a small sample of data.

```
import com.imsl.math.*;
import com.imsl.stat.distributions.*;
import java.text.DecimalFormat;

public class NormalPDEx1 {

    public static void main(String[] args) throws Exception {
        double x[] = {
            24.67074744935116, 23.452437146719504, 23.508058666655124,
            24.085805009837582, 26.387706562892493, 26.244756646592872
        };

        double[] values = new double[x.length];
        double mean = 25.0;
        double stdDev = 1.0;
```

```
        NormalPD normal = new NormalPD();

        for (int i = 0; i < x.length; i++) {
            values[i] = normal.pdf(x[i], mean, stdDev);
        }

        PrintMatrixFormat pmf = new PrintMatrixFormat();
        pmf.setNumberFormat(new DecimalFormat("0.00000000"));

        PrintMatrix pm = new PrintMatrix("Normal(25.0,1.0) pdf values at x=");
        pm.print(x);

        pm.setTitle("f(x;a,b)=");
        pm.print(values);

        double[] gradientApprox
                = normal.getPDFGradientApproximation(x[0], mean, stdDev);

        pm.setTitle("Gradient approximation f(x;a,b) at x[0]");
        pm.print(gradientApprox);

        double[] gradient = normal.getPDFGradient(x[0], mean, stdDev);

        pm.setTitle("Gradient f(x;a,b) at x[0]");
        pm.print(gradient);

        double[][] hessianApprox
                = normal.getPDFHessianApproximation(x[0], mean, stdDev);

        pm.setTitle("Hessian approximation of f(x;a,b) at x[0]");
        pm.print(pmf, hessianApprox);

        double[][] hessian = normal.getPDFHessian(x[0], mean, stdDev);

        pm.setTitle("Hessian of f(x;a,b) at x[0]");
        pm.print(pmf, hessian);
    }
}
```

Output

```
Normal(25.0,1.0) pdf values at x=
     0
0  24.671
1  23.452
2  23.508
3  24.086
4  26.388
5  26.245

f(x;a,b)=
     0
0  0.378
1  0.12
2  0.131
```

```
3  0.263
4  0.152
5  0.184

Gradient approximation f(x;a,b) at x[0]
     0
0  -0.124
1  -0.337

Gradient f(x;a,b) at x[0]
     0
0  -0.124
1  -0.337

Hessian approximation of f(x;a,b) at x[0]
        0           1
0  -0.33692738  0.35977953
1   0.35977953  0.55539644

Hessian of f(x;a,b) at x[0]
        0           1
0  -0.33692735  0.35977916
1   0.35977916  0.55539649
```

Chapter 34: Error Handling

Types

class Messages 2345
class Version 2347
class IMSLFormatter 2347
class Warning 2348
class WarningObject 2351
exception IMSLException 2352
exception IMSLRuntimeException 2353
exception IMSLUnexpectedErrorException 2354
exception LicenseManagerException 2354

Messages class

```
public class com.imsl.Messages
```

Retrieve and format message strings.

Constructor

Messages

```
public Messages()
```

Methods

formatMessage

```
static public String formatMessage(String bundleName, 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.

Returns

the formatted message

formatMessage

```
static public String formatMessage(String bundleName, String key, 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.

Returns

the formatted message

throwIllegalArgumentException

```
static public void throwIllegalArgumentException(String packageName, String
key, Object[] args)
```

Description

Throws an IllegalArgumentException with a formatted String argument.

Parameters

`packageName` – is the 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.

throwIllegalStateException

```
static public void throwIllegalStateException(String packageName, String key,
Object[] args)
```

Description

Throws an IllegalStateException with a formatted String argument.

Parameters

`packageName` – is the 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.

Version class

```
public class com.imsl.Version
```

Print the version information.

Constructor

Version

```
public Version()
```

Method

main

```
static public void main(String[] args) throws ParseException
```

Description

Print the version information about the environment and this library.

IMSLFormatter class

```
public class com.imsl.IMSLFormatter extends java.util.logging.Formatter
```

Simple formatter for classes that implement logging.

Constructor

IMSLFormatter

```
public IMSLFormatter()
```

Method

format

```
public String format(LogRecord record)
```

Description

Format the given log record and return the formatted string.

The LogRecord's message field must contain a reference to one of the resource files in the JMSL Library.

Parameter

`record` – the log record to be formatted.

Returns

the formatted log record.

Warning class

```
public final class com.imsl.Warning
```

Handle warning messages. This class maintains a single, private, WarningObject that actually displays the warning messages.

Constructor

Warning

```
public Warning()
```

Methods

getWarning

```
static public WarningObject getWarning()
```

Description

Gets the WarningObject.

Returns

The current warning object.

print

```
static public void print(Object source, String bundleName, String key, 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

```
static public void setOut(PrintStream out)
```

Description

Reassigns the output stream. The default warning stream is `java.lang.System.err` .

Parameter

`out` – is the new warning output stream. It may be null, in which case warnings are not printed.

setWarning

```
static public void setWarning(WarningObject warningObject)
```

Description

Sets a new WarningObject. Replacing the WarningObject allows warning errors to be handled in a more custom fashion.

Parameter

`warningObject` – is the new WarningObject. It may be null, in which case error messages will be ignored.

Example: Warning and WarningObject classes

This examples shows how to re-driect warning messages, instead of using the default `System.err`. In this example, a warning message is captured, using methods in Warning and WarningObject classes, and re-printed later.

```
import com.imsl.*;
import com.imsl.stat.*;
import java.io.*;

public class WarningEx1 {

    public static void main(String args[]) throws Exception {
        // Capture warning message.
        WarningObject w = Warning.getWarning();

        ByteArrayOutputStream os = new ByteArrayOutputStream();
        PrintStream ps = new PrintStream(os);

        WarningObject newWarning = new WarningObject();
        newWarning.setOut(ps);
        Warning.setWarning(newWarning);

        // 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());

        // Print warning message.
        System.out.println(os.toString());

        // Restore Warning.
        Warning.setWarning(w);
        ps.close();
```

```
        os.close();
    }
}
```

Output

```
The Chi-squared statistic is 4.79629666357389
The P-value is 0.44124295720552553
The Degrees of freedom are 5.0
com.imsl.stat.ChiSquaredTest: An expected value is less than five.
```

WarningObject class

`public class com.imsl.WarningObject`

Handle warning messages.

Field

out

`protected PrintStream out`

The warning stream. Its default value is System.err.

Constructor

WarningObject

`public WarningObject()`

Methods

print

`public void print(Object source, String bundleName, String key, 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. It `source` is a String then it is assumed to be name of the class raising 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 void setOut(PrintStream out)`

Description

Reassigns the output stream. The default warning stream is `java.lang.System.err` .

Parameter

`out` – is the new warning output stream. It may be null, in which case warnings are not printed.

IMSLException class

`abstract public class com.imsl.IMSLException extends java.lang.Exception`

Signals that a mathematical exception has occurred.

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(String s)`

Description

Constructs an IMSLException with the specified detail message. A detail message is a String that describes this particular exception.

Parameter

`s` – the detail message

IMSLException

```
public IMSLException(String packageName, String key, 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.

IMSLRuntimeException class

```
abstract public class com.imsl.IMSLRuntimeException extends
java.lang.RuntimeException
```

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.

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(String s)
```

Description

Constructs an IMSLRuntimeException with the specified detail message. A detail message is a String that describes this particular exception.

Parameter

`s` – the detail message

IMSLRuntimeException

```
public IMSLRuntimeException(String packageName, String key, 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.

IMSLUnexpectedErrorException class

```
public class com.imsl.IMSLUnexpectedErrorException extends
com.imsl.IMSLRuntimeException
```

Signals that an unexpected error has occurred. Please contact your IMSL technical support representative to report this problem. Since IMSLUnexpectedErrorException is a subclass of RuntimeException, this exception does not have to be caught.

Constructor

IMSLUnexpectedErrorException

```
public IMSLUnexpectedErrorException()
```

Description

Constructs an IMSLUnexpectedErrorException.

LicenseManagerException class

```
public class com.imsl.LicenseManagerException extends
com.imsl.IMSLRuntimeException
```

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. Default is `imsl_eval.dat`.
com.imsl.license.popup	`''true''` or `''false''`	If `''true''`, use a dialog box to show any license manager errors. If `''false''`, errors only result in this exception being thrown. Default is to use a popup.

Methods

getErrorNumber

```
public int getErrorNumber()
```

Description

Returns the error number for this exception.

getFeature

```
public String getFeature()
```

Description

Returns the name of the feature that could not be licensed.

getLicensePath

```
public String getLicensePath()
```

Description

Returns the license file path for this exception.

getLocalizedMessage

```
public String getLocalizedMessage()
```

Description

Returns the localized error message for this exception.

Chapter 35: References

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.

Afifi and Azen

Afifi, A.A. and S.P. Azen (1979), *Statistical Analysis: A Computer Oriented Approach*, 2d ed., Academic Press, New York.

Agrawal and Srikant

Agrawal, R. and Srikant, R. (1994), Fast algorithms for mining association rules, *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29- September 1, 1994.

Agresti, Wackerly, and Boyette

Agresti, Alan, Dennis Wackerly, and James M. Boyette (1979), Exact conditional tests for cross-classifications: Approximation of attained significance levels, *Psychometrika*, **44**, 75-83.

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.

Akaike

Akaike, H., (1978), *Covariance Matrix Computation of the State Variable of a Stationary Gaussian Process*, Ann. Inst. Statist. Math. 30 , Part B, 499-504.

Akaike et al

Akaike, H. , Kitagawa, G., Arahata, E., Tada, F., (1979), Computer Science Monographs No. 13, The Institute of Statistical Mathematics, Tokyo.

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.

Altman and Gondzio

Altman, Anna, and Jacek Gondzio (1998), *Regularized Symmetric Indefinite Systems in Interior Point Methods for Linear and Quadratic Optimization*, Logilab Technical Report 1998.6, Logilab, HEC Geneva, Section of Management Studies, Geneva.

Anderberg

Anderberg, Michael R. (1973), *Cluster Analysis for Applications*, Academic Press, New York.

Anderson

Anderson, T.W. (1971), *The Statistical Analysis of Time Series*, John Wiley & Sons, New York.
Anderson, T. W. (1994) *The Statistical Analysis of Time Series*, John Wiley & Sons, New York.

Anderson and Bancroft

Anderson, R.L. and T.A. Bancroft (1952), *Statistical Theory in Research*, McGraw-Hill Book Company, 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.

Barrodale and Roberts

Barrodale, I., and F.D.K. Roberts (1973), An improved algorithm for discrete L1 approximation, *SIAM Journal on Numerical Analysis*, **10**, 839 848.

Barrodale, I., and F.D.K. Roberts (1974), Solution of an overdetermined system of equations in the l1 norm, *Communications of the ACM*, **17**, 319 320.

Barrodale, I., and C. Phillips (1975), Algorithm 495. Solution of an overdetermined system of linear equations in the Chebyshev norm, *ACM Transactions on Mathematical Software*, **1**, 264 270.

Bartlett, M. S.

Bartlett, M.S. (1935), Contingency table interactions, *Journal of the Royal Statistics Society Supplement*, 2, 248 252.

Bartlett, M. S. (1937), Some examples of statistical methods of research in agriculture and applied biology, *Supplement to the Journal of the Royal Statistical Society*, 4, 137-183.

Bartlett, M. (1937), The statistical conception of mental factors, *British Journal of Psychology*, 28, 97-104.

Bartlett, M.S. (1946), On the theoretical specification and sampling properties of autocorrelated time series, *Supplement to the Journal of the Royal Statistical Society*, 8, 27-41.

Bartlett, M.S. (1978), *Stochastic Processes*, 3rd. ed., Cambridge University Press, Cambridge.

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.

Bendel and Mickey

Bendel, Robert B., and M. Ray Mickey (1978), Population correlation matrices for sampling experiments, *Communications in Statistics*, B7, 163 182.

Beckers

Beckers, S. (1980), The Constant Elasticity of Variance Model and its Implications for Option Pricing, *The Journal of Finance*, **(35)** No.3, 661-673.

Bentley and Sedgewick

Bentley, Jon L. and Robert Sedgewick, *Fast Algorithms for Sorting and Searching Strings*, Eighth Symposium on Discrete Algorithms, New Orleans, January, 1997.

Berry and Linoff

Berry, M. J. A. and Linoff, G. (1997) *Data Mining Techniques*, John Wiley & Sons, Inc.

Best and Fisher

Best, D.J., and N.I. Fisher (1979), Efficient simulation of the von Mises distribution, *Applied Statistics*, 28, 152 157.

Bishop

Bishop, C. M. (1995) *Neural Networks for Pattern Recognition*, Oxford University Press.

Bishop et al

Bishop, Yvonne M.M., Stephen E. Feinberg, and Paul W. Holland (1975), *Discrete Multivariate Analysis: Theory and Practice*, MIT Press, Cambridge, Mass.

Bjorck and Golub

Bjorck, Ake, and Gene H. Golub (1973), Numerical Methods for Computing Angles Between Subspaces, *Mathematics of Computation,*, **27**, 579 594.

Blom

Blom, Gunnar (1958), *Statistical Estimates and Transformed Beta-Variables*, John Wiley & Sons, New York.

Blom and Zegeling

Blom, JG, and Zegeling, PA (1994), A Moving-grid Interface for Systems of One-dimensional Time-dependent Partial Differential Equations, *ACM Transactions on Mathematical Software*, Vol 20, No.2, 194-214.

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.

Box and Pierce

Box, G.E.P., and David A. Pierce (1970), Distribution of residual autocorrelations in autoregressive-integrated moving average time series models, *Journal of the American Statistical Association*, 65, 1509-1526.

Boyette

Boyette, James M. (1979), Random RC tables with given row and column totals, Applied Statistics, 28, 329 332.

Bradley

Bradley, J.V. (1968), Distribution-Free Statistical Tests, Prentice-Hall, New Jersey.

Breiman, Leo

Breiman, Leo (2001), Random Forests, *Machine Learning*, October 2001, 45, 1, 5-32.

Breiman, et al.

Breiman, L., Friedman, J. H., Olshen, R. A. and Stone, C. J. (1984) *Classification and Regression Trees*, Chapman & Hall.

Brenan, Campbell, and Petzold

Brenan, K.E., S.L. Campbell, L.R. Petzold (1989), *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elseview Science Publ. Co.

Brent

Brent, Richard P. (1973), *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Breslow

Breslow, N.E. (1974), Covariance analysis of censored survival data, *Biometrics*, 30, 89 99.

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.

Brown

Brown, Morton E. (1983), MCDP4F, two-way and multiway frequency tables-measures of association and the log-linear model (complete and incomplete tables), *in BMDP Statistical Software, 1983 Printing with Additions*, (edited by W.J. Dixon), University of California Press, Berkeley.

Brown and Benedetti

Brown, Morton B. and Jacqualine K. Benedetti (1977), Sampling behavior and tests for correlation in two-way contingency tables, *Journal of the American Statistical Association,*, 42, 309 315.

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.

Chang and Lin

Chang, Chih-Chung and Chih-Jen Lin (2011), LIBSVM : a library for support vector machines, *ACM Transactions on Intelligent Systems and Technology* , 2, 27:1–27:27. Software available at http://www.csie.ntu.edu.tw/c̃jlin/libsvm

Chen and Liu

Chen, C. and Liu, L., Joint Estimation of Model Parameters and Outlier Effects in Time Series, *Journal of the American Statistical Association*, Vol. 88, No.421, March 1993.

Cheng

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317-322.

Chiang

Chiang, Chin Long (1968), *Introduction to Stochastic Processes in Statistics*, John Wiley & Sons, New York.

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.

Conover

Conover, W.J. (1980), *Practical Nonparametric Statistics*, 2d ed., John Wiley & Sons, New York.

Cook and Weisberg

Cook, R. Dennis and Sanford Weisberg (1982), *Residuals and Influence in Regression*, Chapman and Hall, New York.

Cortes and Vapnik

Cortes, C., and V. Vapnik (1995). Support-vector networks, *Machine Learning*, 20 (3): 273.

Courant and Hilbert

Courant, R., and D. Hilbert (1962), *Methods of Mathematical Physics*, Volume II, John Wiley & Sons, New York, NY.

Cox

Cox, David R. (1970), The Analysis of Binary Data, Methuen, London.

Cox, D.R. (1972), Regression models and life tables (with discussion), *Journal of the Royal Statistical Society*, Series B, Methodology, **34**, 187-220.

Cox and Oakes

Cox, D.R., and D. Oakes (1984), *Analysis of Survival Data*, Chapman and Hall, London.

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, P. F., and Rabinowitz, P. (1984), *Methods of Numerical Integration*, Academic Press, Orlando, Florida.

Davis et al.

Davis, T. A., Gilbert, J.R., Larimore, S.I., and Ng, E.G. (2004), *A Column Approximate Minimum Degree Ordering Algorithm*, ACM Transactions on Mathematical Software, **30** (3), 353-376.

de Boor

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, New York.

Deming

Deming, W.E., (1982), *Quality, Productivity and Competitive Position*, Cambridge, MA: Massachusetts Institute of Technology, Center for Advanced Engineering Study.

Demmel et al.

Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S. and Liu, J.W.H. (1999), *A Supernodal Approach to Sparse Partial Pivoting*, SIAM J. Matrix Analysis and Applications, vol. 20 (3), **720-755**.

Demmel, J.W., Gilbert, J.R., Li, X.S. (1999), *SuperLU User's Guide*, Lawrence Berkeley National Laboratory, University of California, Berkeley, CA, Xerox Corporation.

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.

Doornik

Doornik, J.A. (2005), , University of Oxford.

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.

Elant-Johnson et al.

Elandt-Johnson, Regina C., and Norman L. Johnson (1980), Survival Models and Data Analysis, John Wiley and Sons, New York, 172-173.

Elman

Elman, J. L. (1990) *Finding Structure in Time, Cognitive Science*, **14**, 179-211.

Eisenstat et al.

Eisenstat, S.C., Schultz, M.H., and Sherman, A.H. (1981), *Algorithms and Data Structures for Sparse Symmetric Gaussian Elimination,*, SIAM J. Sci. Statist. Comput., vol. 2 **2**, 225-237.

Emmett

Emmett, W.G. (1949), Factor analysis by Lawless method of maximum likelihood, *British Journal of Psychology, Statistical Section*, **2**, 90-97.

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.

Friedman

Friedman, Jerome H. (1999), Stochastic Gradient Boosting, *Technical Report*, Stanford University.

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.

Gear and Petzold

Gear, C.W. and Petzold, Linda R. (1984), ODE methods for the solution of differential/algebraic equations. *SIAM Journal of Numerical Analysis*, **21**, #4, 716.

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.

Gondzio (1994)

Gondzio, Jacek (1994), *Multiple Centrality Corrections in a Primal-Dual Method for Linear Programming*, Logilab Technical Report 1994.20, Logilab, HEC Geneva, Section of Management Studies, Geneva.

Gondzio (1995)

Gondzio, Jacek (1995), HOPDM - *Modular Solver for LP Problems*, User's Guide to version 2.12, WP-95-50, International Institute for Applied Systems Analysis, Laxenburg, Austria.

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.

Gross and Clark

Gross, Alan J., and Virginia A. Clark (1975), *Survival Distributions: Reliability Applications in the Biomedical Sciences*, John Wiley & Sons, New York.

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.

Harman

Harman, Harry H. (1976), *Modern Factor Analysis*, 3d ed. revised, University of Chicago Press, Chicago.

Hart et al.

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.

Hastie et al.

Hastie, Trevor, Robert Tibshirani, Jerome Friedman (2009), *The Elements of Statistical Learning*, 2nd ed., Springer, New York.

Hayter

Hayter, Anthony J. (1984), A proof of the conjecture that the Tukey-Kramer multiple comparisons procedure is conservative, *Annals of Statistics*, **12**, 61-75.

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.

Jöhnk

Jöhnk, 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.

Jöreskog

Jöreskog, 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.

Juran and Godfrey

Juran, J.M., and Godfrey, A.B. (1988), *Juran's Quality Handbook*, 5th ed, New York, McGraw-Hill.

Kachitvichyanukul

Kachitvichyanukul, Voratas (1982), *Computer generation of Poisson, binomial, and hypergeometric random variates*, Ph.D. dissertation, Purdue University, West Lafayette, Indiana.

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.

Kalbfleisch and Prentice

Kalbfleisch, John D., and Ross L. Prentice (1980), *The Statistical Analysis of Failure Time Data*, **1**, 13-14, John Wiley & Sons, New York.

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**.

Kim and Jennrich

Kim, P.J., and R.I. Jennrich (1973), Tables of the exact sampling distribution of the two sample Kolmogorov-Smirnov criterion *Dmn*, in *Selected Tables in Mathematical Statistics*, Volume 1, (edited by H. L. Harter and D.B. Owen), American Mathematical Society, Providence, Rhode Island.

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.

Kochanek and Bartels

Kochanek, Doris H. U., and Bartels, Richard H (1984), *Interpolating Splines with Local Tension, Continuity, and Bias Control*, ACM SIGGRAPH , vol. 18, no. 3, pp. 33-41

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.

Krogh

Krogh, Fred T. (2005), *An Algorithm for Linear Programming*, http://mathalacarte.com/fkrogh/pub/lp.pdf, Tujunga, CA.

Lachenbruch

Lachenbruch, Peter A. (1975), *Discriminant Analysis*, Hafner Press, London.

Lawless

Lawless, J.F. (1982), *Statistical Models and Methods for Lifetime Data*, John Wiley and Sons, New York.

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.

Lawson and Hanson

Lawson, C. L., and Hanson, R. J., (1974), Solving Least Squares Problems, Prentice Hall.

Lawson and Hanson

Lawson, C. L., and Hanson, R. J., (1995), Solving Least Squares Problems, *Classics in Applied Mathematics*, *SIAM Journal on Applied Mathematics* 15.

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.

Leavenworth

Leavenworth, B. (1960), Algorithm 25: Real zeros of an arbitrary function, *Communications of the ACM*, **3**, 602.

Lee

Lee, Elisa T. (1980), *Statistical Methods for Survival Data Analysis*, Lifetime Learning Publications, Belmont, Calif.

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.

Lentini and Pereyra

Pereyra, Victor (1978), PASVA3: An adaptive finite-difference FORTRAN program for first order nonlinear boundary value problems, in *Lecture Notes in Computer Science*, **76**, Springer-Verlag, Berlin, 67 88.

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.

Li

Li, X. S. (2005), *An Overview of SuperLU: Algorithms, Implementation, and User Interface*, ACM Transactions on Mathematical Software, vol. 31 **(3)**, 302-325.

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.

Ljung and Box

Ljung, G.M., and Box,G.E.P. (1978), On a measure of lack of fit in time series models, *Biometrika*, 65, 297-303.

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 Schütze

Manning, C. D. and Schütze, 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.

Miller

Miller, Rupert G., Jr. (1980), *Simultaneous Statistical Inference*,**8**, 101, 254-255. 2d ed., Springer-Verlag, New York.

Milliken and Johnson

Milliken, George A. and Dallas E. Johnson (1984), *Analysis of Messy Data*, Volume 1: Designed Experiments, **31**, Van Nostrand Reinhold, New York.

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}.

Montgomery

Montgomery, D.C. (2001) *Introduction to Statistical Quality Control,*, 4th ed,. Wiley, New York.

Moré et al.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL-80-74, Argonne, Illinois.

Müller

Müller, 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.

NIST Engineering Statistics Handbook

NIST Engineering Statistics Handbook, *http://www.itl.nist.gov/div898/handbook/pmc/section3/pmc3.htm*

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.

Pennington and Berzins

Pennington, S. V., Berzins, M., (1994), Software for First-order Partial Differential Equations. 63-99.

Petro

Petro, R. (1970), Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, *Communications of the ACM*, **13<’**, 624.

Petzold

Petzold, L.R. (1982), A description of DASSL: A differential/ algebraic system solver, Proceedings of the IMACS World Congress, Montreal, Canada.

Piessens et al.

Piessens, R., E. deDoncker-Kapenga, C.W. Überhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, New York.

Platt

Platt, John C. (1998), Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines, *Microsoft Research Technical Report* TSR-TR-98-14. April 21,1998.

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.

Ralston

Ralston, Anthony (1965), *A First Course in Numerical Analysis*, McGraw-Hill, NY.

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

Savage, I. Richard (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.

Shewart

Shewart, W.A., (1931), *Economic Control of Quality of Manufactured Product*, D. Van Nostrand Company.

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.

Smirnov

Smirnov, N.V. (1939), Estimate of deviation between empirical distribution functions in two independent samples (in Russian), *Bulletin of Moscow University*, **2**, 3–16.

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.

Spellucci, Peter

Spellucci, P. (1998), An SQP method for general nonlinear programs using only equality constrained subproblems, *Math. Prog.*, **82**, 413-448, Physica Verlag, Heidelberg, Germany

Spellucci, P. (1998), A new technique for inconsistent problems in the SQP method. *Math. Meth. of Oper*. Res.,**47**, 355-500, Physica Verlag, Heidelberg, Germany.

Spurrier and Isham

Spurrier, John D. and Steven P. Isham (1985), Exact simultaneous confidence intervals for pairwise comparisons of three normal means, *Journal of the American Statistical Association*, 80, 438-442.

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.

Stoline

Stoline, Michael R. (1981), The status of multiple comparisons: simultaneous estimation of all pairwise comparisons in one-way ANOVA designs, *The American Statistician*, 35, 134-141.

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.

Taguchi

Taguchi, G. (1986), Introduction to Quality Engineering,*Asian Productivity Organization*, UNIPUB, White Plains, NY.

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.

Verwer et al

Verwer, J. G., Blom, J. G., Furzeland, R. M., and Zegeling, P. A. (1989), A moving-grid method for one-dimensional PDEs Based on the Method of Lines, *Adaptive Methods for Partial Differential Equations*, Eds., J. E. Flaherty, P. J. Paslow, M. S. Shephard, and J. D. Vasiilakis, SIAM Publications, Philadelphia, PA (USA) pp. 160-175.

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.

Western Electric

Western Electric (1956) *Statistical Quality Control Handbook*, Western Electric Corporation, Indianapolis, IN.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.

Wilmott et al

Wilmott, P., Howison, and S., Dewynne, J., (1996), *The Mathematics of Financial Derivatives (A Student Introduction)*, Cambridge Univ. Press, New York, NY. 317 pages.

Witten and Frank

Witten, I. H. and Frank, E. (2000) Data Mining: *Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann Publishers.

Woodfield

Woodfield, T. J (1990) Data Mining: Some notes on the Ljung-Box Portmanteau Statistic, *American Statistical Association 1990 Proceedings of the Statistical Computing Section*, 155-160.

Wu

Wu, S-I (1995) Mirroring Our Thought Processes, *IEEE Potentials*, **14**, 36-41.

Index

AbstractChartNode, 1567
AbstractFlatFile, 1373
    FlatFileSQLException, 1422
    FlatFileSQLFeatureNotSupportedException, 1422
Activation, 2121
ALACART, 2269
AmbientLight, 1857
ANCOVA, 791
Annotation, 1610
ANOVA, 771
ANOVAFactorial, 781
Apriori, 1926
ARAutoUnivariate, 890
    Formatter, 915
    TriangularMatrixSingularException, 914
ARMA, 928
    IllConditionedException, 953
    IncreaseErrRelException, 948
    MatrixSingularException, 950
    NewInitialGuessException, 949
    TooManyCallsException, 948
    TooManyFcnEvalException, 951
    TooManyITNException, 951
    TooManyJacobianEvalException, 952
ARMAEstimateMissing, 954
ARMAMaxLikelihood, 964
    NonInvertibleException, 979
    NonStationaryException, 980
ARMAOutlierIdentification, 981
ARSeasonalFit, 915
AssociationRule, 1925
AutoARIMA, 1000
    NoAcceptableModelFoundException, 1023
AutoCorrelation, 880
    NonPosVariancesException, 889
Axis, 1613
Axis1D, 1617
Axis3D, 1865
AxisBox, 1863
AxisLabel, 1622, 1868
AxisLine, 1623, 1869
AxisR, 1628
AxisRLabel, 1630
AxisRLine, 1631
AxisRMajorTick, 1631
AxisTheta, 1632
AxisTitle, 1623, 1869
AxisUnit, 1624
AxisXY, 1615
AxisXYZ, 1861

Background, 1608, 1852
Bar, 1720
BarItem, 1726
BarSet, 1727
BasisPart, 1472
Bessel, 528
BetaPD, 2326
BinaryClassification, 2144
Bond, 1473
BootstrapAggregation, 1962
BoundedLeastSquares, 441
    FalseConvergenceException, 451
    Function, 449
    Jacobian, 450
BoundedVariableLeastSquares, 451
    TooManyIterException, 455
BoxPlot, 1685
    Statistics, 1692
BsInterpolate, 180
BsLeastSquares, 182
BSpline, 176

BufferedPaint, 1856

C45, 2275
Candlestick, 1714
CandlestickItem, 1716
Canvas3DChart, 1852
  Paint, 1855
CategoricalGenLinModel, 818
  ClassificationVariableException, 841
  ClassificationVariableLimitException, 841
  ClassificationVariableValueException, 842
  DeleteObservationsException, 842
  RankDeficientException, 843
CChart, 1809
Cdf, 1239
CdfFunction, 1307
CHAID, 2283
Chart, 1562
Chart3D, 1837
ChartFunction, 1645
ChartLights, 1857
ChartNode, 1586
ChartNode3D, 1842
ChartServlet, 1678
ChartSpline, 1645
ChartTitle, 1609
ChartXML, 1766
ChiSquaredTest, 857
  DidNotConvergeException, 864
  NoObservationsException, 863
  NotCDFException, 863
Cholesky, 80
  NotSPDException, 84
ClosedFormMaximumLikelihoodInterface, 2325
ClusterHierarchical, 1146
ClusterKMeans, 1117
  ClusterNoPointsException, 1132
  NoConvergenceException, 1131
  NonnegativeFreqException, 1132
  NonnegativeWeightException, 1133
ClusterKNN, 1134
ColorFunction, 1893
Colormap, 1764
ColormapLegend, 1893
Complex, 549
ComplexFFT, 322
ComplexLU, 60
ComplexMatrix, 14
ComplexSparseCholesky, 92
  NotSPDException, 98
  NumericFactor, 99
  SymbolicFactor, 99
ComplexSparseMatrix, 30
  SparseArray, 39
ComplexSuperLU, 65
ConjugateGradient, 128
  Function, 140
  NoConvergenceException, 138
  NotDefiniteAMatrixException, 137
  NotDefiniteJacobiPreconditionerException, 139
  NotDefinitePreconditionMatrixException, 137
  Preconditioner, 140
  SingularPreconditionMatrixException, 136
ContingencyTable, 805
ContinuousUniformPD, 2329
Contour, 1695
  Legend, 1702
ControlLimit, 1777
Covariances, 607
  DiffObsDeletedException, 616
  MoreObsDelThanEnteredException, 615
  NonnegativeFreqException, 613
  NonnegativeWeightException, 614
  TooManyObsDeletedException, 615
CrossCorrelation, 1024
  NonPosVariancesException, 1035
CrossValidation, 1969
CsAkima, 159
CsInterpolate, 166
CsPeriodic, 168
CsShape, 170
  TooManyIterationsException, 171
CsSmooth, 172
CsSmoothC2, 174
CsTCB, 160
CuSum, 1819
CuSumStatus, 1822

Data, 1634, 1881
  CustomMarkerFactory, 1892
DataNode, 2045
DayCountBasis, 1527

DecisionTree, 2237
MaxTreeSizeExceededException, 2264
PruningFailedToConvergeException, 2263
PureNodeException, 2263
DecisionTreeInfoGain, 2265
GainCriteria, 2268
Dendrogram, 1733
DenseLP, 406
AllConstraintsNotSatisfiedException, 418
BoundsInconsistentException, 414
CyclingOccurringException, 420
MultipleSolutionsException, 417
NoAcceptablePivotException, 415
ProblemUnboundedException, 416
ProblemVacuousException, 417
SomeConstraintsDiscardedException, 419
WrongConstraintTypeException, 414
Difference, 1035
DirectionalLight, 1858
DiscriminantAnalysis, 1174
CovarianceSingularException, 1198
EmptyGroupException, 1197
SumOfWeightsNegException, 1196
Dissimilarities, 1139
NoPositiveVarianceException, 1146
ScaleFactorZeroException, 1145
ZeroNormException, 1145
Distribution, 1311
Draw, 1653
DrawMap, 1679
DrawPick, 1667

Eigen, 144
DidNotConvergeException, 147
EmpiricalQuantiles, 668
ScaleFactorZeroException, 671
EpochTrainer, 2138
EpsilonAlgorithm, 580
ErrorBar, 1703
EWMA, 1815
ExponentialPD, 2333

FactorAnalysis, 1155
BadVarianceException, 1172
EigenvalueException, 1173
NonPositiveEigenvalueException, 1173
NotPositiveSemiDefiniteException, 1170
NotSemiDefiniteException, 1170
RankException, 1169
SingularException, 1171
FaureSequence, 1349
FeedForwardNetwork, 2099
FeynmanKac, 251
Boundaries, 306
BoundaryInconsistentException, 315
ConstraintsInconsistentException, 314
CorrectorConvergenceException, 311
ErrorTestException, 310
ForcingTerm, 308
InitialConstraintsException, 314
InitialData, 307
IterationMatrixSingularException, 312
PdeCoefficients, 304
TcurrentTstopInconsistentException, 313
TEqualsToutException, 313
TimeIntervalTooSmallException, 312
ToleranceTooSmallException, 309
TooManyIterationsException, 310
FFT, 318
FillPaint, 1650
Finance, 1530
FlatFile, 1423
Parser, 1455

GammaDistribution, 1315
GammaPD, 2336
GARCH, 1040
ConstrInconsistentException, 1049
EqConstrInconsistentException, 1049
NoVectorXException, 1048
TooManyIterationsException, 1047
VarsDeterminedException, 1046
GenMinRes, 108
Formatter, 128
Function, 125
Norm, 126
Preconditioner, 125
TooManyIterationsException, 127
VectorProducts, 126
GradientBoosting, 1976
LossFunctionType, 2000
Grid, 1612
GridPolar, 1633

Heatmap, 1745
    Legend, 1754
HiddenLayer, 2115
HighLowClose, 1708
HoltWintersExponentialSmoothing, 1079
Hyperbolic, 544
HyperRectangleQuadrature, 225
    Function, 227

IEEE, 542
IMSLException, 2352
IMSLFormatter, 2347
IMSLRuntimeException, 2353
IMSLUnexpectedErrorException, 2354
InputLayer, 2114
InputNode, 2118
InvCdf, 1296
InverseCdf, 1308
    DidNotConvergeException, 1310
Itemsets, 1923

JFrameChart, 1664
JFrameChart3D, 1840
JMath, 534
JPanelChart, 1665
JspBean, 1675

KalmanFilter, 1050
KaplanMeierECDF, 1199
KaplanMeierEstimates, 1203
Kernel, 2037
KohonenSOM, 1934
KohonenSOMTrainer, 1942
KolmogorovOneSample, 869
KolmogorovTwoSample, 872

LackOfFit, 1076
Layer, 2113
LeastSquaresTrainer, 2134
Legend, 1609
LicenseManagerException, 2354
LifeTables, 1231
LinearKernel, 2039
LinearRegression, 708
    CaseStatistics, 717
    CoefficientTTests, 716
Link, 2123
LogNormalDistribution, 1317
LU, 40

MajorTick, 1625, 1870
Matrix, 9
MaximumLikelihoodEstimation, 2312
MersenneTwister, 1353
MersenneTwister64, 1357
Messages, 2345
MethodOfMomentsInterface, 2326
MinConGenLin, 429
    ConstraintsInconsistentException, 438
    ConstraintsNotSatisfiedException, 439
    EqualityConstraintsException, 440
    Function, 437
    Gradient, 437
    VarBoundsInconsistentException, 439
MinConNLP, 461
    BadInitialGuessException, 486
    ConstraintEvaluationException, 479
    Formatter, 489
    Function, 478
    Gradient, 478
    IllConditionedException, 486
    LimitingAccuracyException, 484
    LinearlyDependentGradientsException, 488
    NoAcceptableStepsizeException, 481
    ObjectiveEvaluationException, 480
    PenaltyFunctionPointInfeasibleException, 483
    QPInfeasibleException, 482
    SingularException, 487
    TerminationCriteriaNotSatisfiedException, 489
    TooManyIterationsException, 484
    TooMuchTimeException, 485
    WorkingSetSingularException, 481
MinorTick, 1625
MinUncon, 351
    Derivative, 357
    Function, 356
MinUnconMultiVar, 357
    ApproximateMinimumException, 366
    FalseConvergenceException, 367
    Function, 365
    Gradient, 365
    Hessian, 366

MaxIterationsException, 368
UnboundedBelowException, 369
MPSReader, 1457
Element, 1469
InvalidMPSFileException, 1468
Row, 1468
MultiClassification, 2187
MultiCrossCorrelation, 1062
NonPosVariancesException, 1075
MultipleComparisons, 803

NaiveBayesClassifier, 1902
Network, 2090
Node, 2118
NonlinearRegression, 720
Derivative, 735
Function, 734
NegativeFreqException, 733
NegativeWeightException, 733
TooManyIterationsException, 734
NonlinLeastSquares, 369
Function, 378
Jacobian, 379
TooManyIterationsException, 378
NonNegativeLeastSquares, 456
TooManyIterException, 460
TooMuchTimeException, 461
NormalDistribution, 1313
NormalityTest, 864
NoVariationInputException, 868
NormalPD, 2339
NormOneSample, 632
NormTwoSample, 638
NpChart, 1802
NumericalDerivatives, 490
Function, 509
Jacobian, 509

ODE, 230
OdeAdamsGear, 240
DidNotConvergeException, 249
Function, 247
Jacobian, 247
MaxFcnEvalsExceededException, 249
SingularMatrixException, 250
ToleranceTooSmallException, 248
OdeRungeKutta, 236
DidNotConvergeException, 240
Function, 238
ToleranceTooSmallException, 239
OutputLayer, 2116
OutputPerceptron, 2120

ParetoChart, 1830
PartialCovariances, 617
InvalidMatrixException, 622
InvalidPartialCorrelationException, 623
PChart, 1805
Pdf, 1280
AltSeriesAccuracyLossException, 1295
PDFGradientInterface, 2324
PDFHessianInterface, 2324
Perceptron, 2119
Physical, 569
PickEvent, 1673
PickListener, 1674
Pie, 1728
PieSlice, 1732
PointLight, 1860
PoissonDistribution, 1319
Polar, 1741
PolynomialKernel, 2044
PooledCovariances, 623
PredictiveModel, 1945
PredictiveModelException, 1958
StateChangeException, 1959
SumOfProbabilitiesNotOneException, 1960
VariableType, 1960
PrintMatrix, 583
PrintMatrixFormat, 588
ProbabilityDistribution, 1311, 2321
ProportionalHazards, 1211
ClassificationVariableLimitException, 1230

QR, 99
QuadraticProgramming, 421
InconsistentSystemException, 427
NoLPSolutionException, 428
ProblemUnboundedException, 427
SolutionNotFoundException, 429
Quadrature, 218
Function, 224
QuasiNewtonTrainer, 2125
BlockGradObjective, 2133

BlockObjective, 2133
Error, 2130
GradObjective, 2132
Objective, 2131
QUEST, 2289

RadialBasis, 202
Function, 212
Gaussian, 214
HardyMultiquadric, 213
RadialBasisKernel, 2042
Random, 1324
BaseGenerator, 1349
RandomSamples, 1362
RandomSequence, 1361
RandomTrees, 2298
ReflectiveOperationException, 2309
Ranks, 659
RChart, 1785
RegressionBasis, 741
RegressorsForGLM, 698

ScaleFilter, 2202
SChart, 1796
SelectionRegression, 741
NoVariablesException, 754
Statistics, 754
Sfun, 511
ShewhartControlChart, 1771
SigmoidKernel, 2040
SignTest, 845
SingularMatrixException, 141
Sort, 650
SparseCholesky, 85
NotSPDException, 91
NumericFactor, 92
SymbolicFactor, 91
SparseLP, 380
CholeskyFactorizationAccuracyException, 397
DiagonalWeightMatrixException, 396
DualInfeasibleException, 399
IllegalBoundsException, 405
IncorrectlyActiveException, 404
IncorrectlyEliminatedException, 404
InitialSolutionInfeasibleException, 400
PrimalInfeasibleException, 398
PrimalUnboundedException, 398
ProblemUnboundedException, 401
TooManyIterationsException, 401
ZeroColumnException, 402
ZeroRowException, 403
SparseMatrix, 19
SparseArray, 29
Spline, 155
Spline2D, 184
Spline2DInterpolate, 188
Spline2DLeastSquares, 197
SplineData, 1717
StepwiseRegression, 756
CoefficientTTests, 768
CyclingIsOccurringException, 767
NoVariablesEnteredException, 767
Summary, 595
SuperLU, 45
SupportVectorMachine, 2003
CloneNotSupportedException, 2012
ReflectiveOperationException, 2012
Surface, 1870
ZFunction, 1880
SVClassification, 2013
SVD, 103
DidNotConvergeException, 107
SVOneClass, 2025
SVRegression, 2029
SymEigen, 148

TableMultiWay, 682
BalancedTable, 688
UnbalancedTable, 690
TableOneWay, 671
TableTwoWay, 676
Text, 1646
TimeSeries, 1088
TimeSeriesClassFilter, 2222
TimeSeriesFilter, 2220
TimeSeriesOperations, 1096
CombineMethod, 1107
Function, 1105
MergeRule, 1106
Tokenizer, 1456
ToolTip, 1648
Trainer, 2124
Transform, 1626

TransformDate, 1627
Tree, 2234
Treemap, 1755
    Legend, 1763
TreeNode, 2229

UChart, 1812
UnsupervisedNominalFilter, 2211
UnsupervisedOrdinalFilter, 2215
UserBasisRegression, 736

VectorAutoregression, 1108
Version, 2347

Warning, 2348
WarningObject, 2351
WilcoxonRankSum, 849

XbarR, 1779
XbarS, 1789
XmR, 1799

ZeroPolynomial, 328
    DidNotConvergeException, 332
ZerosFunction, 333
    Function, 338
ZeroSystem, 339
    DidNotConvergeException, 345
    Function, 346
    Jacobian, 346
    ToleranceTooSmallException, 347
    TooManyIterationsException, 347