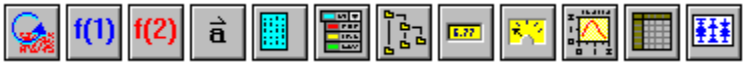
Math2 - Version 1.5



The unit *MATH2* offers some of the most fundamental procedures of univariate statistics (such as mean values, standard deviations, or the correlation coefficient for a series of data pairs). In addition, several more advanced mathematical methods (such as the calculation of eigenvectors and principal components, a simple clustering algorithm, and the approximation of the normal and the t-distribution) are contained in this unit. The unit *MATH2* also contains a simple, yet powerful class (TCurveFit) which supports curve fitting of bivariate data.

MATH2 uses the unit *DCommon* which supplies some oftenly used numerical constants and a special procedure type (FeedbackProc). Note that only those declarations of *DCommon* are documented in this help file which are important for the unit *MATH2*.

Math1 is released under the shareware concept. The unregistered version is fully functional with one exception: the shareware version runs properly only when the Delphi, or C++Builder IDE is also up and running.

Product information
Interface of unit MATH2
Installation of MATH2
Delphi and C++Builder issues

Class TCurveFit:

CalcGaussFit
CalcHyperbolFit
CalcLinFit
CalcLogFit
CalcParabolFit
CalcReciLinFit
CalcStatistics
CorrCoeff
EnterStatValue
Init
MeanDiff
MeanX
MeanY
NumData
StdDevDiff
StdDevX
StdDevY

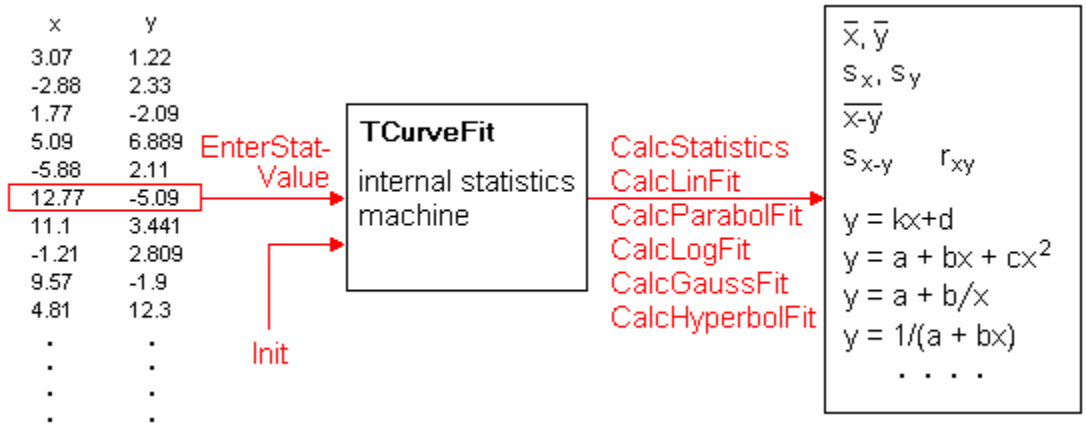
Various procedures and functions:

CalcCovar
CalcEigVec
CalcFishQ
CalcGaussKernel
CalcGaussKernelMat
CalcPrincComp
FeedBackProcType
FindCenters
FindNearestNeighbors
GetEigenResult
MathFeedBackProc
MeanDistanceKNN
nDistri
nDistriQuantile
ProcStat
RemoveEigenMatrix
tDistriQuantile

TCurveFit

See also:

The class TCurveFit provides a means of calculating regression estimators for certain types of functions. In addition, TCurveFit calculates the most important statistical parameters, such as the mean values, the standard deviation, and the correlation coefficient of a series of data pairs. In order to utilize the class TCurveFit, you have to enter the data pairs (x and y) by the method EnterStatValue. The regression parameters can be obtained then by calling the appropriate regression method.



The following table summarizes the routines of TCurveFit:

Init	initialize the processing machine
EnterStatValue	enter a pair of data
CalcStatistics	calculate mean values, standard deviations, and the correlation coefficient
CalcGaussFit	calculate the best Gaussean fit (normal distribution)
CalcHyperbolFit	calculate the best hyperbolic fit to the data
CalcLinFit	calculate the best linear fit to the data
CalcLogFit	calculate the best logarithmic fit
CalcParabolFit	calculate the best parabolic fit to the data
CalcReciLinFit	calculate the best reciprocal linear fit to the data

In addition, TCurveFit also provides all important univariate parameters for the entered data pairs:

CorrCoeff	correlation coefficient
MeanDiff	mean of differences between x and y
MeanX	mean of x values
MeanY	mean of y values
NumData	number of data pairs
StdDevDiff	standard deviation of differences between x and y
StdDevX	standard deviation of x values
StdDevY	standard deviation of y values

CalcCovar

See also: CorrCoeff

Declaration:
```
function CalcCovar (InData, CovarMat: TMatrix; LoC, HiC: integer;
LoR, HiR: integer; Scaling: integer): boolean;
```

The routine CalcCovar calculates the covariance matrix, or alternatively, the scatter matrix, or the correlation matrix for a given data matrix. The resulting covariance matrix is of the dimension **(HiC-LoC+1)*(HiC-LoC+1)**, this means that the rows of the data matrix are seen as objects and the columns are seen as variables.

The parameter **InData** contains the input data. The parameter **CovarMat** holds a pointer to a matrix which will contain the covariance matrix to be calculated.

The parameters **LoC**, **HiC**, **LoR**, and **HiR** specify the range of the data matrix, which should be used for the calculation. This enables the user to calculate 'local' covariance matrices which originate only from a part of the data.

The parameter **Scaling** determines how to scale the data before calculating the result matrix. This will lead to three different matrices, depending on the value of **Scaling**:

Scaling	Action	Resulting matrix
0	none	scatter matrix of **InData**
1	mean centering	covariance matrix of **InData**
2	standardization	correlation matrix of **InData**

Standardization of the data (sometimes called 'autoscaling') is performed by scaling the data in a way that the mean of each columns becomes zero and the variance becomes 1.0. The covariance matrix of standardized data is usually called 'correlation matrix'.

The function CalcCovar returns the value TRUE if the calculation of the covariance matrix has been performed successfully. A FALSE value indicates that the calculation has not been completed successfully (mostly due to lack of memory or too small a matrix of **CovarMat**).

Hint: The user is responsible by his own that the matrix **CovarMat** has been declared large enough to hold the resulting covariance matrix. This matrix has to be quadratic and of size (**HiC**-**LoC**+1)*(**HiC**-**LoC**+1). If the matrix **CovarMat** is too large the covariance matrix is stored in the matrix elements with the lowest indices (starting at 1). The unused elements are not changed at all.

Example: COVAR.DPR
The example program COVAR.DPR shows the application of the function CalcCovar. The user can enter values into a matrix and calculate either the scatter matrix, the covariance matrix, or the correlation matrix.

CalcDiffMat

See also: CalcCovar

Declaration:
```
function CalcDiffMat (InData, DiffMat: TMatrix; LoC, HiC: integer;
LoR, HiR: integer; Scaling: integer): boolean;
```

The routine CalcDiffMat calculates the difference matrix from a given data matrix. The resulting matrix is of the dimension (**HiC-LoC**+1)*(**HiC-LoC**+1), this means that the rows of the data matrix are seen as objects and the columns are seen as variables.

The parameter **InData** holds a pointer to the matrix of the input data. This matrix is an object of the class TMatrix. The parameter **DiffMat** holds a pointer to a matrix which will contain the difference matrix to be calculated.

The parameters **LoC**, **HiC**, **LoR**, and **HiR** specify the range of the data matrix, which should be used for the calculation. This enables to calculate 'local' difference matrices which originates from only a part of the data. Normally the calculation will be performed for the whole data matrix **InData** (**LoC**=1, **HiC=InData.NrOfColumns**, **LoR**=1, and **HiR=InData.NrOfRows**).

The parameter **Scaling** determines how to scale the data before calculating the result matrix. This will lead to three different matrices, depending on the parameter **Scaling**:

Scaling	Action	Resulting matrix
0	none	difference matrix of **InData**
1	mean centering	??? matrix of **InData**
2	standardization	??? matrix of **InData**

Standardization of the data (sometimes called 'autoscaling') is performed by scaling the data in a way that the mean of each columns becomes zero and the variance becomes 1.0.

The function CalcDiffMat returns the value TRUE if the calculation of the difference matrix has been performed successfully. A returned value of FALSE indicates that the calculation could not finished properly (mostly due to lack of memory or too small a matrix **DiffMat**).

Hint: The user is responsible by his own that the matrix **DiffMat** has been declared large enough to hold the resulting difference matrix. This matrix has to be quadratic and of size (**HiC-LoC**+1)*-(**HiC-LoC**+1). If the matrix **DiffMat** is too large the difference matrix is stored in the matrix elements with the lowest indices (starting at 1). The unused elements are not changed at all.

CalcEigVec

See also: CalcPrincComp, GetEigenResult, RemoveEigenMatrix

Declaration: `function CalcEigVec (InMat: TMatrix): boolean;`

The routine CalcEigVec calculates the eigenvectors and eigenvalues of a symmetrical matrix **`InMat`**. The parameter **`InMat`** is a pointer to an instance of the class **`Matrix`**. The function CalcEigVec returns the value TRUE, if the calculations have been completed successfully.

The results of CalcEigVec (i.e. the eigenvalues and the eigenvectors) are stored in memory and can be read by the routine GetEigenResult. Please note that the 16-bit version of the unit MATH2 allows only a maximum of 90 eigenvectors, whereas the 32-bit version allows 1000 eigenvectors.

Hint: The eigenvalues and eigenvectors are stored in memory as long as either a new calculation is performed or the results are removed from the memory by explicitly calling the routine RemoveEigenMatrix.

Example: The statement **`success := CalcEigVec (data);`** calculates the eigenvectors and eigenvalues of the matrix **`data`**. The variable **`success`** is set TRUE, if the calculation has been performed successfully.

CalcFishQ

See also: CalcStatistics

Declaration: `function CalcFishQ (m1,m2,s1,s2: Double): Double;`

The Fisher ratio is often used to enumerate the degree of overlap of two distributions. The procedure CalcFishQ calculates the Fisher ratio from the mean values and the standard deviations of the two distributions.

Hint: If the sum of the standard deviations falls below 10E-6, a value of 10E-6 is substituted for it.

CalcGaussKernel

See also: CalcGaussKernelMat

Declaration:
```
function CalcGaussKernel (Probe, RefCenter: TVector; Width:
Double): Double;
```

The function CalcGaussKernel calculates the value of a given n-dimensional Gaussean kernel. The parameters **`RefCenter`** and **`Width`** define the position and the width of the kernel. The parameter **`Probe`** specifies the position in the n-dimensional space where to calculate the value of the kernel.

Hint: The user has to ensure that the dimensionalities of the vectors **`Probe`** and **`RefCenter`** are equal. A mismatch of these would result in erroneous results without notification.

CalcGaussKernelMat

See also: CalcGaussKernel

Declaration:
```
function CalcGaussKernelMat (Probe: TVector; RefCenterMat:
TMatrix; RefCenterIx: integer; Width: Double): Double;
```

The function CalcGaussKernelMat calculates the value of a given n-dimensional Gaussean kernel. The parameters **`RefCenterMat`**, **`RefCenterIx`**, and **`Width`** define the position and the width of the Gaussean kernel. The parameter **`Probe`** specifies a point in the n-dimensional space where to calculate the value of the kernel.

The routine CalcGaussKernelMat is equivalent to CalcGaussKernel with the exception that the position of the kernel is defined by the row **`RefCenterIx`** of the matrix **`RefCenterMat`**. This can be of some benefit when calculating the effect of several Gaussean kernels which are defined by the rows of a matrix.

Hint: The user has to ensure that the dimension of the vector **`Probe`** and the first dimension of the matrix **`RefCenterMat`** are equal. A mismatch of these would result in erroneous results without notification.

CalcGaussFit

See also: CalcLinFit, CalcLogFit

Class: TCurveFit

Declaration: `procedure CalcGaussFit (var k0, k1, k2, FitQual: double);`

The procedure CalcGaussFit calculates the best fitting Gaussean curve (normal distribution) for a given set of data. The curve is determined by the equation

$$y = k_0 e^{-\frac{(x-k_1)^2}{k_2}}$$

The values of x and y are given by the data samples, the parameters **`k0`**, **`k1`**, and **`k2`** are estimated by CalcGaussFit using a least squares approximation.

The data points [x,y] have to be entered using the routine EnterStatValue. A minimum number of 3 values is required in order to apply CalcGaussFit. Do not forget to reset the statistics calculation before entering any new data sets (use the method Init)

In addition to the parameters **`k0`**, **`k1`**, and **`k2`**, CalcGaussFit returns the goodness of fit **`FitQual`**. This parameter may vary between 0.0 and 1.0, indicating the best possible fit if **`FitQual`** equals 1.0.

Hint: The quality of fit calculated by CalcGaussFit is not adjusted for the degree of freedoms in the regression parameters.

Example: CURVEFIT.DPR
This application lets you draw points in a chart and calculate various types of regression curves. Note, that CURVEFIT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

CalcHyperbolFit

See also: CalcReciLinFit

Class: TCurveFit

Declaration: `CalcHyperbolFit (var k0,k1: double; var FitQual: double);`

The procedure CalcHyperbolFit calculates the best fitting Hyperbola for a given set of data. The Hyperbola is determined by the equation

$$y = k_0 + \frac{k_1}{x}$$

The values of x and y are given by the data samples, the parameters **`k0`**, and **`k1`** are estimated by CalcHyperbolFit using a least squares approximation.

The data points [x,y] have to be entered using the routine EnterStatValue. A minimum number of 2 values is required in order to apply CalcHyperbolFit. Do not forget to reset the statistics calculation before entering any new data sets (use Init)

In addition to the parameters **`k0`**, and **`k1`**, CalcHyperbolFit returns the goodness of fit in the parameter **`FitQual`**. This quality of fit may vary between 0.0 and 1.0, indicating the best possible fit if **`FitQual`** equals 1.0.

Hint: The quality of fit calculated by CalcHyperbolFit is not adjusted for the degree of freedoms in the regression parameters.

Example: CURVEFIT.DPR
This application lets you draw points in a chart and calculate various types of regression curves. Note, that CURVEFIT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

CalcLinFit

See also: CalcLogFit, CalcParabolFit, CalcReciLinFit

Class: TCurveFit

Declaration: `CalcLinFit (var k, d, FitQual: Double);`

The procedure CalcLinFit estimates the best fit of a straight line to a sample of two-dimensional data points using linear regression. The line is defined by the equation

$$y = kx + d.$$

The data points have to be entered by using the procedure EnterStatValue. A minimum number of 2 values is required in order to apply CalcLinFit. Do not forget to reset the statistics calculation before entering any new data sets (use Init)

The procedure CalcLinFit returns the following parameters: **`k`** and **`d`** define the slope and the offset of the line, and **`FitQual`** returns the goodness of fit of the regression. **`FitQual`** equals the square of the correlation coefficient. A good representation of the data samples yields a value near to 1.0 for **`FitQual`**.

Hint: The quality of fit calculated by CalcLinFit is not adjusted for the degree of freedoms in the regression parameters.

Example: CURVEFIT.DPR
This application lets you draw points in a chart and calculate various types of regression curves. Note, that CURVEFIT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

CalcLogFit

See also: CalcLinFit

Class: TCurveFit

Declaration: `procedure CalcLogFit (var k0, k1, FitQual: double);`

The procedure CalcLogFit calculates the best fitting logarithmic curve for a given set of data. The curve is determined by the equation

$$y = k0 + k1 * \ln(x)$$

The values of x and y are given by the data samples, the parameters **`k0`**, and **`k1`** are estimated by CalcLogFit using a least squares approximation.

The data points [x,y] have to be entered using the routine EnterStatValue. A minimum number of 2 values is required in order to apply CalcLogFit. Do not forget to reset the statistics calculation before entering any new data sets (use the method Init)

In addition to the parameters **`k0`**, and **`k1`**, CalcLogFit the quality of fit **`FitQual`**. This parameter may vary between 0.0 and 1.0, indicating the best possible fit if **`FitQual`** equals 1.0.

Hint: The quality of fit calculated by CalcLogFit is not adjusted for the degree of freedoms in the regression parameters.

Example: CURVEFIT.DPR
This application lets you draw points in a chart and calculate various types of regression curves. Note, that CURVEFIT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

CalcParabolFit

See also: CalcHyperbolFit, CalcLinFit

Class: TCurveFit

Declaration: `CalcParabolFit (var k0,k1,k2: Double; var FitQual: Double);`

The procedure CalcParabolFit calculates the best fitting parabola for a given set of data. The parabola is determined by the equation

$$y = k_0 + k_1 x + k_2 x^2$$

The values of x and y are given by the data samples, the parameters **k0**, **k1**, and **k2** are estimated by CalcParabolFit using a least squares approximation.

The data points [x,y] have to be entered using the routine EnterStatValue. A minimum number of 3 values is required in order to apply CalcParabolFit. Do not forget to reset the statistics calculation before entering any new data sets (use the method Init)

In addition to the parameters **k0**, **k1**, and **k2**, CalcParabolFit returns the goodness of fit **FitQual**. This parameter may vary between 0.0 and 1.0, indicating the best possible fit if **FitQual** equals 1.0.

Hint: The quality of fit calculated by CalcParabolFit is not adjusted for the degree of freedoms in the regression parameters.

Example: CURVEFIT.DPR
This application lets you draw points in a chart and calculate various types of regression curves. Note, that CURVEFIT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

CalcReciLinFit

See also: CalcHyperbolFit, CalcLinFit, CalcParabolFit

Class: TCurveFit

Declaration: `procedure CalcReciLinFit (var k0, k1, FitQual: double);`

The procedure CalcReciLinFit calculates the best fitting reciprocal line curve for a given set of data. The curve is determined by the equation

$$y = \frac{1}{k_0 + k_1 x}$$

The values of x and y are given by the data samples, the parameters **`k0`**, and **`k1`** are estimated by CalcReciLinFit using a least squares approximation.

The data points [x,y] have to be entered using the routine EnterStatValue. A minimum number of 2 values is required in order to apply CalcReciLinFit. Do not forget to reset the statistics calculation before entering any new data sets (use the method Init)

In addition to the parameters **`k0`**, and **`k1`** CalcReciLinFit returns the goodness of fit **`FitQual`**. This parameter may vary between 0.0 and 1.0, indicating the best possible fit if **`FitQual`** = 1.0.

Hint: The quality of fit calculated by CalcReciLinFit is not adjusted for the degree of freedoms in the regression parameters.

Example: CURVEFIT.DPR
This application lets you draw points in a chart and calculate various types of regression curves. Note, that CURVEFIT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

CalcPrincComp

See also: RemoveEigenMatrix, GetEigenResult

Declaration:
```
function CalcPrincComp (InData: TMatrix; LoC, HiC: integer; LoR,
HiR: integer; Scaling: integer): boolean;
```

The procedure CalcPrincComp calculates the principal components of the matrix **Indata** (Principal Component Analysis, PCA). The rows of the data matrix are objects, the columns are variables. The resulting principal components can be read using the routine GetEigenResult. All eigenvalues are scaled to their sum equal 1.0. The eigenvectors are arranged according to decreasing eigenvalues, thus the first eigenvector corresponds to the first principal component.

The function CalcPrincComp allocates some memory for the storage of the covariance and the eigenvector matrix. The memory for the eigenvector matrix is not deallocated after finishing CalcPrincComp and must be deallocated explicitly by calling the routine RemoveEigenMatrix. If CalcPrincComp is called several times the old eigenvector matrix is released and a new one is allocated.

The parameters **LoC**, **HiC**, **LoR**, and **HiR** specify the area of the matrix which is used for the principal component analysis. This facilitates the calculation of the principal components of only parts of the input data. Usually, the PCA is applied to all the available data (**LoC**=1, **HiC**=InData.NrOfColumns, **LoR**=1, and **HiR**=InData.NrOfRows).

The parameter **Scaling** determines, whether the data should be before calculating the principal components:

Scaling	Result
0	PCA is based on scatter matrix of **InData**
1	PCA is based on covariance matrix of **InData**
2	PCA is based on correlation matrix of **InData**

The function CalcPrincComp returns a TRUE value, if the PCA has been completed successfully. If FALSE is returned, the calculation has been aborted (mostly due to lack of memory).

CalcStatistics

See also: Init, EnterStatValue, CalcLinFit

Class: TCurveFit

Declaration:
```
CalcStatistics (var NumData: longint; var MeanX, MeanY, StdevX,
StdevY, MeanDiff, StdevDiff, rxy: Double);
```

The procedure CalcStatistics calculates the mean values, the standard deviations and the correlation coefficient of pairs of numbers. In addition, the mean and the standard deviation of the difference of the pairs are calculated.

In order to make use of CalcStatistics the user has first initialize the internal registers (Init). Thereafter the pairs of x-y-values have to be entered to the register by using the procedure EnterStatValue. The procedure CalcStatistics evaluates the parameters using the data entered to far. It calculates the mean values and the standard deviations of x, y, and the difference x-y, as well as the correlation coefficient between x and y. In addition, the number of data points entered is returned in the parameter **`NumData`**.

This procedure can also be used to calculate the mean value and standard deviation for a single series of data. In this case one of the values of the data pair [x,y] should be set to zero when using the procedure EnterStatValue. The calculated correlation coefficient is then, of course, invalid.

Hint 1: Since it is only meaningful to calculate the standard deviation and the correlation coefficient with at least 3 data points the procedure CalcStatistics refuses to calculate these measures if less than 3 data points have been entered. The standard deviations and the correlation coefficient are set to 0.0 in this case.

Hint 2: All the parameters returned by CalcStatistics can also be accessed individually by the appropriate properties (CorrCoeff, MeanDiff, MeanX, MeanY, NumData, StdDevDiff, StdDevX, StdDevY).

CorrCoeff

See also: EnterStatValue, CalcStatistics, NumData

Class: TCurveFit

Declaration: `property CorrCoeff: double;`

The read-only property CorrCoeff returns the correlation coefficient between the x- and y-values entered by the method EnterStatValue. Note, that at least 3 data pairs have to be entered to calculate a valid correlation coefficient.

EnterStatValue

See also: Init, CalcStatistics, CalcLinFit

Class: TCurveFit

Declaration: `EnterStatValue (x,y: double);`

The procedure EnterStatValue is used to enter data points for later statistical evaluation. The parameters **`x`** and **`y`** are of the type **`double`**. A maximum of approx. 2 billion data pairs can be stored; a larger number of calls to EnterStatValue results in a corrupted calculation of the statistical parameters.

FeedBackProcType

See also: ProcStat, MathFeedBackProc

Declaration: `FeedBackProcType = procedure (StateCnt: longint);`

This procedure type is used as call-back routine in lengthy mathematical calculations. It is declared in unit *DCommon*.

FindCenters

See also: MeanDistanceKNN, FindNearestNeighbors

Declaration:
```
FindCenters (InMat: TMatrix; RowLo, RowHi: integer; NumCent:
integer;  var Centers: TMatrix; var MeanDist: double);
```

If a matrix is seen as a collection of measurements (rows=objects, columns=variables) it is sometimes useful to calculate prototype objects which are representative for the whole data set. This calculation of the prototypes is equal to the searching of clusters in an n-dimensional space, where n is the number of variables per object.

The procedure FindCenters calculates the specified number of centers **NumCent**. The parameters **RowLo** and **RowHi** determine which objects are used for the calculation.

The centers are calculated as follows: First, the two objects are searched which exhibit the smallest distance of each other. This pair of data is now replaced by their center of gravity. The calculation of the center of gravity allows for the number of data points already used in preceding unifications. Now the unification is repeated until only **NumCent** points are left. These points are equal to the inquired centers.

The procedure FindCenters returns the found prototypes in the matrix **Centers** and the mean distance between them in the parameter **MeanDist**. The user has to take care of by himself that the matrix **Centers** is large enough to accept all calculated prototype points (i.e. **Centers** has to have at least **NumCent** rows and **InMat.NrOfCOlumns** columns). If the matrix 'Centers' is too small the result will be truncated accordingly. If it is too large, rows from **NumCent**+1 on are not defined and excess columns are set to zero values.

Hint 1: The time it takes to compute the prototypes increases with the third power of the number of objects. Thus it is strongly recommended not to use this algorithm for more than approx. 200 objects.

Hint 2: FindCenters stores the original data of the matrix in a file named MAT.$$$ in order to reload the data after the calculation of the prototype points. The auxiliary file MAT.$$$ is automatically deleted after the execution of FindCenters.

Example: FINDCENT.DPR
The example application FINDCENT shows the usage of FindCenters to find centers in a two-dimensional data array. Note, that FINDCENT.DPR uses the components RCHART and NUMLAB for the user interface, which are not included with the MATH2 package.

FindNearestNeighbors

See also: FindCenters, MeanDistanceKNN

Declaration:
```
FindNearestNeighbors (k: integer; InMat: TMatrix; FirstObj,
LastObj: integer; DatVec: TVector; KNNList: TMatrix);
```

This routine calculates the k nearest neighbors of the vector **DatVec** in the matrix **InMat**. The rows are considered to be objects, the columns are the variables. The user has to ensure that both the x-dimension of **InMat** and the length of **DatVec** are equal to each other. The result is returned as an ordered list in the matrix **KNNList** (size 2 x k), the first row specifying the nearest neighbor. The first column of **KNNList** holds the indices into the data matrix **InMat**, the second column holds the distances to these neighbors. The parameters **FirstObj** and **LastObj** define the index of the first and the last object of matrix **Inmat** to be included with the kNN search.

GetEigenResult

See also: CalcEigVec, CalcPrincComp

Declaration: `function GetEigenResult (EigVecNum: integer; VecElem: integer): real;`

The routine GetEigenResult returns one element of a specific eigenvector. The parameter **`EigVecNum`** defines the number of the eigenvector, the parameter **`VecElem`** specifies the number of the element of that vector. If **`VecElem`** is set to zero the routine returns the eigenvalue of the eigenvector **`EigVecNum`**. Any invalid parameters result in a zero return value.

Hint: The function GetEigenResult is also used in connection with the procedure CalcPrincComp to read the principal components.

Example: The statement "`if (GetEigenResult (2,0) > 0.1) then ...`" checks if the eigenvalue of the second eigenvector is larger than 0.1.

Literature References

See also:

Hartung .....

Init

See also: EnterStatValue, CalcStatistics, CalcLinFit, CalcParabolFit

Class: TCurveFit

Declaration: `Init;`

The procedure Init initializes the internal variables which are used to calculate fundamental statistical parameters (cf. CalcStatistics). Be sure to apply Init before entering any data.

MathFeedBackProc

See also: ProcStat

Declaration: `MathFeedBackProc : FeedBackProcType;`

Some math calculations can consume a lot of computer time during which the application may show no reaction. In order to avoid this unfavourable situation, all the potentially time-consuming routines (i.e. CalcCovar, CalcEigVec, CalcPrincComp, and FindCenters) have implemented a call to a special procedure, MathFeedBackProc, which is executed at regular intervals. The global variable ProcStat is automatically incremented and passed as a parameter to MathFeedBackProc.

The procedure MathFeedBackProc is in fact a procedure pointer which is initially set to NIL. The user may assign any procedure of the type FeedBackProcType to it.

Below you find short instructions how to implement a call-back routine:

1. Declare the routine which does the actual feedback (e.g. showing a progress bar, or something like that). This procedure has to have exactly the same declaration as FeedBackProcType. Don't forget to declare this procedure as **`far`**:

   ```
   procedure ShowProgress (cnt: longint); far;

   begin      { here the display of the progress takes place }
   Form1.Label1.Caption := IntToStr (cnt);
   end;
   ```

   Note, that the **`cnt`** parameter of the call back procedure passes the value of the variable ProcStat.

2. Before starting the time-consuming math procedure (**`FindCenters`**, in this example), assign the feedback procedure to the procedure variable MathFeedbackProc. You should also set the variable ProcStat to some defined value. After the completion of the time-consuming procedure you may reset the MathFeedbackProc to NIL.

   ```
   ...
   ProcStat := 0;
   MathFeedbackProc := ShowProgress;
   FindCenters (Data, 1, Data.NrOfRows, NCenters, Centers, Md);
   MathFeedbackProc := NIL;
   ...
   ```

Hint 1: An example how to implement a call-back routine can also be found in the sample program FINDCENT.DPR.

Hint 2: Alternatively, you can use a timer interrupt which displays the contents of the variable ProcStat to display the progress of some calculation.

MeanDiff

See also: EnterStatValue, CalcStatistics, NumData, MeanX, MeanY

Class: TCurveFit

Declaration: `property MeanDiff: double;`

The read-only property MeanDiff returns the mean value of the difference between the x- and the y-values entered by the method EnterStatValue:

$$MeanDiff = \frac{\Sigma(x_i - y_i)}{NumData}$$

MeanX

See also: EnterStatValue, CalcStatistics, NumData, MeanDiff, MeanY

Class: TCurveFit

Declaration: `property MeanX: double;`

The read-only property MeanX returns the mean value of the x-values entered by the method EnterStatValue.

MeanY

See also: EnterStatValue, CalcStatistics, NumData, MeanX, MeanDiff

Class: TCurveFit

Declaration: `property MeanY: double;`

The read-only property MeanY returns the mean value of the y-values entered by the method EnterStatValue.

MeanDistanceKNN

See also: FindCenters

Declaration:
```
MeanDistanceKNN (InMat: TMatrix; kn: integer; var DistVec:
TVector);
```

The procedure MeanDistanceKNN calculates the mean distance of the **kn** nearest neighbors of each sample in the matrix **InMat**. The rows of the matrix **InMat** are seen as objects (samples), the columns are regarded as variables (measurements) which define the data space.

The results are stored in the vector **DistVec**. The user has to assure that the vector **DistVec** is large enough to accept the distances of all objects (**DistVec** has to have at least **InMat.NrOfRows** rows). If the vector **DistVec** is too small the result will be truncated accordingly.

nDistri

See also: tDistriQuantile, nDistriQuantile

Declaration: `function nDistri (x: double): double;`

The function nDistri returns the integral of the normal distribution between minus infinity and the value of the parameter **x**. The returned value may vary between 0.0 and 1.0.

Hint: The calculated value is an approximation based on a formula given by Hartung [Hartung Vol. 1, page 890] and is correct to about 3 decimal places.

nDistriQuantile

See also: tDistriQuantile, nDistri

Declaration: `function nDistriQuantile (Gamma: Double): Double;`

The function nDistriQuantile returns the quintile value of the standard normal distribution for a given significance level **`Gamma`**. The parameter **`Gamma`** may take any value in the open interval (0,1). If nDistriQuantile is called using an invalid value of **`Gamma`**, a value of 0.0 is returned.

Hint 1: Do not mix up the function nDistriQuantile with the Gaussean density function and the cumulative normal distribution. nDistriQuantile returns the argument x for a given definite integral of value **`Gamma`** for this density function.

Hint 2: The calculated value is an approximation based on a formula given by Hartung [Hartung Vol. 1, page 891] and is correct to about 3 decimal places.

NumData

See also: EnterStatValue, CalcStatistics

Class: TCurveFit

Declaration: `property NumData: longint;`

The read-only property NumData returns the number of data pairs entered so far by the method EnterStatValue. The method Init resets NumData to zero.

ProcStat

See also: MathFeedBackProc, CalcCovar, CalcEigVec, CalcPrincComp, FindCenters

Declaration: `ProcStat: longint;`

This globally defined variable serves as a means for communicating the state of some potentially lengthy calculations to the outside world. It is incremented on a regular basis by certain routines (CalcCovar, CalcEigVec, CalcPrincComp, and FindCenters). Thus a timer interrupt routine can access this variable and display its value during the calculations. This method provides an efficient way to check whether the system hangs or is involved in lengthy calculations. The initialization of ProcStat is left up to the user.

Alternatively, a routine showing the progress of some calculation can be assigned to the call-back routine MathFeedBackProc. This routine is called whenever ProcStat is increased.

RemoveEigenMatrix

See also: CalcEigVec, CalcPrincComp

Declaration: `RemoveEigenMatrix;`

The procedure RemoveEigenMatrix removes an eigenvector matrix from the memory. If no matrix is stored in the memory, the call to RemoveEigenMatrix is without any consequence.

StdDevDiff

See also: EnterStatValue, CalcStatistics, NumData, StdDevX, StdDevY

Class: TCurveFit

Declaration: `property StdDevDiff: double;`

The read-only property StdDevDiff returns the standard deviation of the difference between the x- and the y-values entered by the method EnterStatValue.

StdDevX

See also: EnterStatValue, CalcStatistics, NumData, StdDevDiff, StdDevY

Class: TCurveFit

Declaration: `property StdDevX: double;`

The read-only property StdDevX returns the standard deviation of the x-values entered by the method EnterStatValue.

StdDevY

See also: EnterStatValue, CalcStatistics, NumData, StdDevX, StdDevDiff

Class: TCurveFit

Declaration: `property StdDevY: double;`

The read-only property StdDevY returns the standard deviation value of the y-values entered by the method EnterStatValue.

tDistriQuantile

See also: nDistri, nDistriQuantile

Declaration: `function tDistriQuantile (Gamma: Double; ndata: integer): Double;`

The function tDistriQuantile returns the value of the t-distribution (Students distribution) for a given significance level **`Gamma`** and a specified number of data points **`ndata`**. The parameter **`Gamma`** may take any value in the open interval (0,1). The parameter **`ndata`** may take values between 1 and MaxInt. If tDistriQuantile is called using invalid arguments a value of 0.0 is returned.

Hint: The calculated value is an approximation based on a formula given by Hartung [Hartung Vol. 1, page 892] and is correct to about 3-4 decimal places.

Delphi and C++Builder Issues

See also: Installation of Math2

Any component of SDL comes both for the Delphi and the C++Builder environment. Thus, you get full support both for Delphi and C++Builder with our components. The following general directory layout of the delivered zip files makes it easier to access the parts you need:

Please note, that the RES files of the sample applications are supplied both as a 16-bit and a 32-bit version (denoted by the extension RES16, and RES32). You have to rename the proper version to RES before compiling the sample applications.

Registration of Math2

See also: How to order

The unit Math2 is a shareware component. You can register it for US$ 29.60. In return, you get the newest version of Math2 and unlimited free updates by email. The registered version is, of course, fully functional and does no longer need the Delphi, or the C++Builder IDE. The user may then create stand-alone programs which can be deployed without paying any further licences.

Please note, that the unit *MATH2* is based on three other units (Math1, Vector, and Matrix), which are included with the registered version of *MATH2*. The sources of *MATH2* are available separately at a price of US$ 150,- (again, including the sources of Math1, Vector, and Matrix).

Click here for further details and infos on available discounts.

Installation of Math2

See also: Delphi and C++Builder Issues

The installation of the unit is a two-part process. First you have to copy the DCU file(s) to the Delphi (C++ Builder) library. Secondly, you should install the help file of this component into the help system of Delphi (C++ Builder).

Note: The unit MATH2 is **not** a visual component. You therefore cannot install it in the VCL palette. The routines of MATH2 are used by including MATH2 into the `'uses'` statement of your program.

Delphi installation

C++ Builder installation

Interface of Math2

Note: you may have to switch to small fonts to get a legible interface list.

```
interface
  uses dcommon, math1, matrix, vector;

  type
    CombiType  = array[0..255] of byte;
    TCurveFit =
      class (TObject)
      private
        sumx, sumy        : double;
        sumxq, sumyq      : double;
        sumDiff, SumDiffq : double;
        sumxy             : double;
        sumx2y, sumx3     : double;
        sumx4             : double;
        sum1byy           : double;
        sum1byyq          : double;
        sumxbyy           : double;
        sumybyx           : double;
        sum1byx           : double;
        sum1byxq          : double;
        sumlnx            : double;
        sumlnxq           : double;
        sumylnx           : double;
        sumlny            : double;
        sumlnyq           : double;
        sumxlny           : double;
        sumxqlny          : double;
        FNumData          : longint;
        function GetMeanX: double;
        function GetMeanY: double;
        function GetStdDevX: double;
        function GetStdDevY: double;
        function GetMeanDiff: double;
        function GetStdDevDiff: double;
        function GetRxy: double;
      public
        constructor Create;
        destructor Destroy; override;
        procedure Init;
        procedure EnterStatValue (x,y: double);
        procedure CalcStatistics (var NumData: longint;
             var MeanX, MeanY, StdevX, StdevY, MeanDiff,
             StdevDiff, rxy: double);
        procedure CalcGaussFit (var k0, k1, k2, FitQual: double);
        procedure CalcLinFit (var k, d, FitQual: double);
        procedure CalcLogFit (var k0, k1, FitQual: double);
        procedure CalcParabolFit (var k0, k1, k2, FitQual: double);
        procedure CalcReciLinFit (var k0, k1, FitQual: double);
        procedure CalcHyperbolFit (var k0, k1, FitQual: double);
        property  NumData: longint read FNumData;
        property  MeanX: double read GetMeanX;
        property  MeanY: double read GetMeanY;
        property  StdDevX: double read GetStdDevX;
        property  StdDevY: double read GetStdDevY;
        property  MeanDiff: double read GetMeanDiff;
        property  StdDevDiff: double read GetStdDevDiff;
```

```
      property  CorrCoeff: double read GetRxy;
    end;

var
  ProcStat         : longint;                   { state of process }
  MathFeedBackProc : FeedBackProcType;  { global math feedback procedure }

function  CalcCovar
           (InData : TMatrix;                          { input data }
          CovarMat : TMatrix;                  { covariance matrix }
          LoC, HiC : integer;                   { range of columns }
          LoR, HiR : integer;                      { range of rows }
           Scaling : integer)   { 0=none, 1=mean cent., 2=autoscl. }
                   : boolean;                    { TRUE if success }
function  CalcEigVec
            (InMat : TMatrix)             { symmetric input matrix }
                   : boolean;                    { TRUE if success }
function  CalcFishQ
            (m1,m2,                      { mean values, class 1 & 2 }
             s1,s2 : double)                 { standard deviations }
                   : double;                        { Fisher ratio }
function  CalcGaussKernel
            (Probe : TVector;                     { probe position }
         RefCenter : TVector;                   { center of kernel }
             Width : double)                    { width of kernel }
                   : double;                              { result }
function  CalcGaussKernelMat
            (Probe : TVector;                     { probe position }
      RefCenterMat : TMatrix;           { matrix of kernel centers }
       RefCenterIx : integer;       { index into the kernel matrix }
             Width : double)                    { width of kernel }
                   : double;                              { result }
function  CalcPrincComp
           (InData : TMatrix;              { pointer to data array }
          LoC, HiC : integer;                   { range of columns }
          LoR, HiR : integer;                      { range of rows }
           Scaling : integer)  { 0=none, 1=mean cent., 2=autoscal. }
                   : boolean;                    { TRUE if success }
procedure FindCenters
            (InMat : TMatrix;                          { data matrix }
      RowLo, RowHi : integer;                { first & last object }
           NumCent : integer;                  { number of centers }
        var Centers : TMatrix;                 { matrix of centers }
      var MeanDist : double);                      { mean distance }
procedure FindNearestNeighbors
                (k : integer;                { number of neighbors }
             InMat : TMatrix;              { matrix to be searched }
          FirstObj : integer;                       { first object }
           LastObj : integer;                        { last object }
            DatVec : TVector;             { vector to be searched }
           KNNList : TMatrix);                            { result }
function  GetEigenResult
        (EigVecNum : integer;              { number of eigenvector }
           VecElem : integer)                     { vector element }
                   : real;                        { matrix element }
procedure MeanDistanceKNN
            (InMat : TMatrix;                          { data matrix }
                kn : integer;             { # of nearest neighbors }
          FirstRow : integer;            { first object to be used }
           LastRow : integer;             { last object to be used }
        var DistVec: TVector);               { result for each obj }
function  nDistri
                (x : double)                              { argument }
```

```
                    : double;          {integral of norm.dens.funct.}
function  nDistriQuantile
             (Gamma : double)                 { significance level }
                    : double;                { normal distribution }
procedure RemoveEigenMatrix;
function tDistriQuantile
             (Gamma : double;                 { significance level }
              ndata : integer)                    { number of data }
                    : double;                     { t-distribution }
```

Free Additional Help Files

Back to the Contents

For technical reasons, this help file is restricted to a single component or unit. If you want to have the cumulative help for all components available from SDL, you may download a free copy of the cumulative help from one of the following web sites:

http://www.lohninger.com/

or

http://qspr03.tuwien.ac.at/lo/

How to order

Back to the Contents

For registration please run the program REGISTER.EXE (which prints the registration form) and send the printed form together with the proper amount of money (preferrably credit cards or cash, for cheques please add US$ 5,-) to the following address:

Software Development Lohninger
P.O.B. 123
A-1061 Vienna
Austria, Europe
email: helpdesk@lohninger.com

The components are preferrably delivered by email. For quick delivery (within 24 hours), please fax the order form (credit card orders, only) to the fax number indicated at our home page:

http://www.lohninger.com/

There are several additional Delphi / C++Builder components, and other software products available from SDL. Ordering several components (w/o source code) together will entitle you to a discount:

2 components	-10 %
3..5 components	-20 %
more than 5	-30 %

Special offer 1: The maximum registration fee is limited to US$55,- (take all components and pay only US$ 55,-). *This special offer and the discount are not applicable to the prices of the sources.*

Special offer 2: Purchase the sources of all of our components for only US$410,-

Both offers include

- **unlimited free updates** by email
- **free** registered copies of any **future components**
- **free support for both the Delphi and the C++Builder** version of the components

For the latest release of SDL´s Delphi and C++Builder components and for pricing information see **http://www.lohninger.com/** or write to the address above.

C++Builder Installation

See also: Delphi installation

1. Please mind that **the unit is not a visual component**. Therefore, this unit cannot be installed in the C++Builder VCL.

   1.1 Copy the OBJ file(s) to the directory where the C++Builder library is located (`XXX/CBuilder/Lib`).
   1.2 Copy the HPP file(s) into the include directory (`XXX/CBuilder/Include`)

2. Adding the help file to the C++Builder help system

   2.1 (optional) Copy the HLP file(s) to the help directory of C++Builder
   2.2 Run the OpenHelp application from the C++Builder group menu.
   2.3 Click on the "Add" button, and add the help file(s) to the "Available Help Files" list
   2.4 Select the help file(s) and click on ">"
   2.5 Click on OK. Thereafter the C++Builder help system knows about the new component(s).

As an alternative, you could install the cumulative help file LOCOMP.HLP which covers all components of SDL. The cumulative help is available for free at the following web site:

`htpp://www.lohninger.com/`

Delphi Installation

See also: C++ Builder installation

1. Please mind that **the unit is not a visual component**. Therefore, this unit cannot be installed in the Delphi VCL.

   1.1 Copy the DCU file(s) to the directory where the Delphi library is located.

2. Merging the help keywords into the master help index

   2.1 Copy the HLP and the KWF files to the directory where the master help file DELPHI.HLP is located (usually `\DELPHI\BIN`).
   2.2 Run the HELPINST application from the Delphi group menu.
   2.3 Open the master index DELPHI.HDX
   2.4 Add the keyword file (.KWF)
   2.1 Save and compile the new index. Thereafter the Delphi master help index includes the keywords for the component's Help screens.

   As an alternative, you could install the cumulative help file LOCOMP.HLP into the Delphi *Tools* menu. This approach is far better and much more reliable than merging the help files into the master help index. The cumulative help is available for free at the following web site:

   htpp://www.lohninger.com/

Note: Assure yourself that the PATH environment variable includes the directory where the help file is stored. Otherwise the Windows help system will not find the new help file. Alternatively, you can copy the help file to the Windows directory.

A personal remark

We are putting much effort into the production of most reliable components and units for other programmers. Everybody who is using our software gets our expertise at a competitive price. But this is only possible if everybody who uses our software, registers it and pays for it. So, if you don't want to kill the shareware idea, don't hesitate to support our work and register this piece of software.

Hans Lohninger
Vienna, Austria.