LMath 0.6.1: mathematical library for Free Pascal — Lazarus

Reference Guide

Edited by Viatcheslav V. Nesterov

September 26, 2024

Contents

Chapter 1

About the library

LMath, further development of DMath library from Jean Debord, is a general purpose mathematical library for FreePascal (FPC) and Lazarus. LMath provides routines and demo programs for numerical analysis, including mathematical functions, probabilities, matrices, optimization, linear and nonlinear equations, integration, Fast Fourier Transform, random numbers, curve fitting, statistics and graphics. It is organized as a set of lazarus packages. Such organization makes it easily extensible and helps to include only really needed features in your project.

DMath stands for Delphi Math, and is a continuation of an earlier work which was named TPMath, for Turbo Pascal Math. Continuing this tradition, this library is called LMath: Lazarus Math.

About the documentation

DMath comes with very nice manual which describes not only the library itself, but a lot of underlying theory. This document, written by Jean Debord, is included with LMath library. Practically everything what you find in this manual remains true for LMath as well. However, new code is, of course, not covered by DMath manual.

This Reference Guide is relatively terse, but complete. It contains formal descriptions of every routine, variable or constant found in the library and may serve as a brief reference. It covers both old and new procedures. Each chapter describes one package so you can easily find where needed function is located. I suggest that DMath.pdf is used to study the most of the library, and this document for the acquaintance with new possibilities and for quick reference.

All elements added or modified in LMath library are labelled in the margins, as shown here. If this symbol is related to a unit identifier, it means that the entire unit is new. Accompanying Guide to New Functions in LMath provides more detailed introduction to new functions written for LMath. LMath

1.1 Structure of the Library

The library is organized as 13 relatively small packages:

1. lmGenMath. This package defines several important data structures, used later in the whole library, some utility functions, basic math functions and special functions. All other packages of the library depend on lmGenMath.

2. lmMathUtil. Various function for manipulations with arrays, sorting and formatting. Depends on lmGenMath.

3. lmLinearAlgebra. Operations over vectors and matrices. Depends on lmMathUtil and lmGenMath.

4. lmPolynoms. Evaluation of polynomials, polynomial roots finding, polynomial critical points finding. Depends on lmGenMath and lmLinearAlgebra.

5. lmIntegrals. Numeric integrating and solving differential equations. Depends on lmGenMath.

6. lmRandoms. Generation of random numbers. Depends on lmGenMath.

7. lmMathStat Descriptive statistics, hypothesis testing, collection of various distributions. Depends on lmGenMath and lmLinearAlgebra.

8. lmOptimum. Algorithms of function minimization. Somewhat artificially, unit uEval for evaluation of an expression, is included into this package. Depends on lmGenMath, lmLinearAlgebra, lmRandoms, lmMathStat.

9. lmNonLinEq. Finding of roots of non-linear equations. Depends on lmGenMath, lmLinearAlgebra, lmOptimum.

10. lmDSP. Functions for digital signal processing. Convolution, collection of filters and Fourier Transform procedures.

11. lmRegression. Functions for linear and non-linear regression, curve fitting. Collection of common models. Unit uFFT for fast Fourier Transform is located also here. Depends on lmLinearAlgebra, lmPolynoms, lmOptimum, lmMathUtil.

12. lmSpecRegress. Collection of field-specific models for data fitting. Depends on lmGenMath and lmRegression.

13. lmPlotter. Routines for data and functions plotting. Depends on lmGenMath, lmMathUtil and LCL.

All these packages are described in the following chapters.

1.2 Installation, Compilation and use of the library

Download and unpack file *LMath_and_Components06.zip*. Open and compile packages lmGenMath, lmMathUtil, lmLinearAlgebra, lmPolynoms, lmIntegrals, lmRandoms, lmMathStat, lmOptimum, lmNonLinEq, lmRegression, lmSpecRegress, lmDSP, lmPlotter, in this order. Afterwards you may compile LMath.pkg, which simply depends on all these packages. Therefore, if you add dependency on LMath to your project, it is not necessary to add every single package.

Alternatively, if you have Project Groups package installed in your Lazarus IDE, you can open and compile LMath project group, select LMath target and compile it.

If you are going to use LMComponents, open LMComponents package, compile and install it.

LMComponents is an object-oriented extension of LMath. It contains TCoordSys component which serves for graphical representation of functions or data, several dialogs and input controls, as well as DSP filters implemented as components which allow real time filtering of a signal during an acquisition.

For the sake of maximally broad compatibility, rather conservative optimization options are selected in LMath packages (optimization level 2, no options for modern processors selected). Depending on your system, you may want to increase this level and use options for modern processors. For example, you may want to define

```
-CfAVX2
-CpCOREAVX2
-OoFASTMATH
-OoLOOPUNROLL
```

to effectively use AVX2 registers.

"Do not generate debug information" is selected by default in the options of LMath packages. If you want to step into LMath procedures debugging your programs, change it. If you find a bug in LMath, please, report it at the official site at Sourseforge.

Chapter 2

Package lmGenMath

2.1 Description

Package lmGenMath (General Maths) includes units which introduce basic data types (Float, Complex, TRealPoint, TInterval) and structures which are used later throughout the whole library; defines operations over these data types; defines error handling mechanism used in the library; implements basic and special math functions and contains several utility routines which strictly speaking do not belong to the field of numeric analysis, but are needed for input-output operations. All other packages of the library depend on lmGenMath package.

2.2 Unit utypes

2.2.1 Description

Global types and constants, dynamic arrays. Importantly, type `Float` (see below, ref. 2.2.2.2) is defined here, which should be used as a type for real numbers in the programs using LMath.

2.2.2 Types

Float

Declaration `Float = Double;`

Description This type defines floating point numbers. By default it is DOUBLE (8-byte real). Other types may be selected by defining the complile options: SINGLEREAL (Single precision, 4 bytes) EXTENDEDREAL (Extended precision. 10 bytes, except of Windows 64 where Extended is an alias for Double (8 bytes)).

TRealPoint

LMath

Declaration

```
TRealPoint = record
  X: Float;
  Y: Float;
end;
```

Description Represents point on a cartesian plane. Unit uRealPoints defines operations over TRealPoint as a vector in 2-dimentional space.

TIntegerPoint

LMath

Declaration

```
PIntegerPoint = ^TIntegerPoint
TIntegerPoint = record
  X: Integer;
  Y: Integer;
end;
```

Description Represents point on a cartesian plane, for example on a screen.

Complex

Declaration
```
Complex = record
  X: Float;
  Y: Float;
end;
```

Description Represents Complex number. Unit uComplex defines operations and functions with this type.

RNG_Type

Declaration `RNG_Type = (...);`

Description Random number generators. See chapter 6 for its use.

Values `RNG_MWC` Multiply-With-Carry

`RNG_MT` Mersenne Twister

`RNG_UVAG` Universal Virtual Array Generator

TRegMode

Declaration `TRegMode = (...);`

Description Curve fit. Defines regression mode (weighted, unweighted); see Chapter 11.

Values `OLS`

`WLS` Regression mode

TOptAlgo

Declaration `TOptAlgo = (...);`

Description Optimization algorithms for nonlinear regression (chapter 11).

Values `NL_MARQ` Marquardt algorithm

`NL_SIMP` Simplex algorithm

`NL_BFGS` BFGS algorithm

`NL_SA` Simulated annealing

`NL_GA` Genetic algorithm

StatClass

Declaration
```
StatClass = record  { Statistical class }
  Inf : Float;          { Lower bound }
  Sup : Float;          { Upper bound }
  N   : Integer;        { Number of values }
  F   : Float;          { Frequency }
  D   : Float;          { Density }
end;
```

Description This type represents a class (bin) in a statistical distribution and is used by function distrib and several other functions from `lmMathStat` package. `Inf` is a lower bound of a bin, `Sup` is its upper bound; `N` is a count of values which fell into this bin (n_i); `F` is a frequency of the bin ($F = n_i/N$), where N is a number of values in the whole distributed population and `D` is probability density ($F/(Sup - Inf)$). See file DMath.pdf, Section 16.5 and 16.6 and chapter 7 for details.

TRegTest

Declaration

```
TRegTest = record      { Test of regression }
  Vr       : Float;    { Residual variance }
  R2       : Float;    { Coefficient of determination }
  R2a      : Float;    { Adjusted coeff. of determination }
  F        : Float;    { Variance ratio (explained/residual) }
  Nu1, Nu2 : Integer;  { Degrees of freedom }
end;
```

Description This type is used by `RegTest` and `WRegTest` functions from `lmMathStat` package. See DMath.pdf, 17.6.2 and Chapter 7.

TRealPointVector

LMath

Declaration `TRealPointVector = array of TRealPoint;`

TVector

Declaration `TVector = array of Float;`

TIntVector

Declaration `TIntVector = array of Integer;`

TCompVector

Declaration `TCompVector = array of Complex;`

TBoolVector

Declaration `TBoolVector = array of Boolean;`

TStrVector

Declaration `TStrVector = array of String;`

TCompOperator

LMath 0.5

Declaration `TCompOperator = (LT, LE, EQ, GE, GT, NE);`

Description This type is used as a parameter in some comparators for element-wise functions with TVector and TMatrix. See 13.5.

TMatrix

Declaration `TMatrix = array of TVector;`

TIntMatrix

Declaration `TIntMatrix = array of TIntVector;`

TCompMatrix

Declaration `TCompMatrix = array of TCompVector;`

TBoolMatrix

Declaration `TBoolMatrix = array of TBoolVector;`

TStrMatrix

Declaration `TStrMatrix = array of TStrVector;`

TFunc

Declaration `TFunc = function(X : Float) : Float;`

Description Float Function of one float variable

TTestFunc

LMath

Declaration `TTestFunc = function(X:Float):boolean;`

Description Boolean function of one `Float` variable. Used in some array-related procedures in uVecUtils unit (13.5).

TIntTestFunc

Declaration `TIntTestFunc = function(X:Integer):boolean;`

Description Boolean function of one integer variable. Used in some array-related procedures in uVecUtils unit (13.5).

TIntFloatFunc

Declaration `TIntFloatFunc = function(X:Integer):Float;`

Description Float function of one `integer` variable. Used in some array-related procedures in uVecUtils unit (13.5).

TFuncNVar

LMath

Declaration `TFuncNVar = function(X : TVector) : Float;`

Description Function of several variables placed in a vector

TEquations

Declaration `TEquations = procedure(X, F : TVector);`

Description Nonlinear equation system. Used in equation solvers.

TDiffEqs

Declaration `TDiffEqs = procedure(X : Float; Y, Yp : TVector);`

Description Differential equation system

TJacobian

Declaration `TJacobian = procedure(X : TVector; D : TMatrix);`

Description Jacobian

TGradient

Declaration `TGradient = procedure(X, G : TVector);`

Description Gradient

THessGrad

Declaration `THessGrad = procedure(X, G : TVector; H : TMatrix);`

Description Hessian and Gradient

TParamFunc LMath

Declaration `TParamFunc = function(X : Float; Params : Pointer) : Float;`

TComparator LMath 0.5

Declaration `TComparator = function(Val, Ref : float) : boolean;`

Description Function for general comparisons of `Float`. Used for example in the element-wise procedures and functions over `TVector` and `TMatrix`. See 13.5.

TStatClassVector

Declaration `TStatClassVector = array of StatClass;`

Description Vector of "StatClass". Used in "uDistrib" unit for generation of a frequency and density table of a randomly distributed variable.

TRegFunc

Declaration `TRegFunc = function(X : Float; B : TVector) : Float;`

Description Regression function which is used by NLFit and other procedures of non-linear fitting.

TDerivProc

Declaration `TDerivProc = procedure(X, Y : Float; B, D : TVector);`

Description Procedure to compute the derivatives of the regression function with respect to the regression parameters.

TCobylaObjectProc

LMath 0.5

Declaration `TCobylaObjectProc = procedure (N, M : integer; const X : TVector; out F : Float; CON : TVector);`

Description Objective function supplied to COBYLA minimization algorythm. `N` is number of arguments to be adjusted; `M` is number of constraints; TVector `X[N]` is the current vector of variables. TVector `Con[M+2]` is vector of constraint values. The subroutine should return values of the objective and constraint functions in `F` and `CON[1],CON[2],...,CON[M], respectively`. `Con[M+1]` and `Con[M+2]` are used internally. Note that we are trying to adjust `X` so that `F(X)` is as small as possible subject to the constraint functions being nonnegative. Importantly, constraints can be violated during the calculation! This means that if your objective function contains, for example, $\ln(x)$, hence you have a constrain $x > 0$, you cannot rely on this constrain during the calculation and must check for validity of your values yourself.

TMintVar

Declaration `TMintVar = (...);`

Description Variable of the integrated Michaelis equation: Time, Substrate conc., Enzyme conc.

Values `Var_T`

`Var_S`

`Var_E`

Str30

Declaration `Str30 = String[30];`

Description Graphics

TScale

Declaration `TScale = (...);`

Description

Values `LinScale`

`LogScale`

TGrid

Declaration `TGrid = (...);`

Description

Values `NoGrid`

`HorizGrid`

`VertiGrid`

`BothGrid`

TArgC

Declaration `TArgC = 1..MaxArg;`

TWrapper

Declaration `TWrapper = function(ArgC : TArgC; ArgV : TVector) : Float;`

2.2.3 Constants

Euler

LMath 0.6

Declaration Euler = 2.71828182845904523536;

Description Euler's number (logarithm base)

Pi

Declaration `Pi = 3.14159265358979323846;`

Description π

Ln2

Declaration `Ln2 = 0.69314718055994530942;`

Description $\ln(2)$

Ln10

Declaration `Ln10 = 2.30258509299404568402;`

Description $\ln(10)$

LnPi

Declaration `LnPi = 1.14472988584940017414;`

Description $\ln(\pi)$

InvLn2

Declaration `InvLn2 = 1.44269504088896340736;`

Description $1/\ln(2)$

InvLn10

Declaration `InvLn10 = 0.43429448190325182765;`

Description $1/\ln(10)$

TwoPi

Declaration `TwoPi = 6.28318530717958647693;`

Description $2\,\pi$

PiDiv2

Declaration `PiDiv2 = 1.57079632679489661923;`

Description $\pi/2$

SqrtPi

Declaration `SqrtPi = 1.77245385090551602730;`

Description $\sqrt{\pi}$

Sqrt2Pi

Declaration `Sqrt2Pi = 2.50662827463100050242;`

Description $\sqrt{2\pi}$

InvSqrt2Pi

Declaration `InvSqrt2Pi = 0.39894228040143267794;`

Description $1/\sqrt{2\pi}$

LnSqrt2Pi

Declaration `LnSqrt2Pi = 0.91893853320467274178;`

Description $\ln(\sqrt{(}2\pi))$

Ln2PiDiv2

Declaration `Ln2PiDiv2 = 0.91893853320467274178;`

Description $\ln(2\pi)/2$

Sqrt2

Declaration `Sqrt2 = 1.41421356237309504880;`

Description $\sqrt{2}$

Sqrt2Div2

Declaration `Sqrt2Div2 = 0.70710678118654752440;`

Description $\sqrt{2}/2$

Gold

Declaration `Gold = 1.61803398874989484821;`

Description

$$GoldenMean = \frac{1+\sqrt{5}}{2}$$

CGold

Declaration `CGold = 0.38196601125010515179;`

Description 2 - GOLD

MachEp

Declaration `MachEp = 2.220446049250313E-16;`

Description Floating point precision: 2^{-52}

MaxNum

Declaration `MaxNum = 1.797693134862315E+308;`

Description Max. floating point number: 2^{1024}

MinNum

Declaration `MinNum = 2.225073858507202E-308;`

Description Min. floating point number: 2^{-1022}

MaxLog

Declaration `MaxLog = 709.7827128933840;`

Description Max. argument for Exp = Ln(MaxNum)

MinLog

Declaration `MinLog = -708.3964185322641;`

Description Min. argument for Exp = Ln(MinNum)

MaxFac

Declaration `MaxFac = 170;`

Description Max. argument for Factorial

MaxGam

Declaration `MaxGam = 171.624376956302;`

Description Max. argument for Gamma

MaxLgm

Declaration `MaxLgm = 2.556348E+305;`

Description Max. argument for LnGamma

NaN

LMath

Declaration `NaN = 0.0/0.0;`

Infinity

LMath

Declaration `Infinity = 1.0/0.0;`

NegInfinity

LMath

Declaration `NegInfinity = -1.0/0.0;`

C_infinity

Declaration `C_infinity : Complex = (X : MaxNum; Y : 0.0);`

C_zero

Declaration `C_zero : Complex = (X : 0.0; Y : 0.0);`

C_one

Declaration `C_one : Complex = (X : 1.0; Y : 0.0);`

C_i

Declaration `C_i : Complex = (X : 0.0; Y : 1.0);`

C_pi

Declaration `C_pi : Complex = (X : Pi; Y : 0.0);`

C_pi_div_2

Declaration `C_pi_div_2 : Complex = (X : PiDiv2; Y : 0.0);`

MaxSize

Declaration `MaxSize = 32767;`[1]

Description Max. array size: $2^{15} - 1$

MaxArg

Declaration `MaxArg = 26;`

Description Max number of arguments for a function

2.2.4 Variables

DefaultZeroEpsilon

Declaration `DefaultZeroEpsilon: Float = MachEp / 8;`

Description This value is used in functions IsZero, IsNegative and IsPositive for approximate comparison with zero. Set it according to the scale of your calculations using procedure SetZeroEpsilon (see 2.2.5.8).

[1] All values of machine constants are given for `Float = Double`

DefaultEpsilon

Declaration `DefaultEpsilon: Float = MachEp;`

Description This value is used for comparison of floats in function SameValue. The function scales it according to the scale of the values of the floats been compared. Set it to the scale of your values with procedure SetEpsilon.

2.2.5 Functions and Procedures

IsZero

LMath

Declaration `function IsZero(F: Float; Epsilon: Float = -1): Boolean;`

Description Compares to zero using epsilon. If Epsilon is -1, DefaultEpsilon as set with prior call to SetZeroEpsilon is used. if `SetZeroEpsilon` was not called, $MachEp/8$ is used.

IsNan

LMath

Declaration `function IsNan(F : Float): Boolean;`

Description Test if given value is NAN

SameValue

LMath

Declaration
```
function SameValue(A, B: Float; epsilon : float): Boolean;
overload;
function SameValue(A, B: Float): Boolean; overload;
```

Description Tests for approximate equality of two floats. First function returns true if $|A - B| < |\epsilon|$. Second one uses relative test: it returns true if $|A - B| < |\epsilon \cdot \max(A, B)|$ where ϵ is `DefaultEpsilon` which can be set by call to SetEpsilon procedure. if `SetEpsilon` was not called, DefaultEpsilon = MachEp.

SetAutoInit

Declaration `procedure SetAutoInit(AutoInit : Boolean);`

Description Sets the auto-initialization of arrays

DimVector

LMath

Declaration
```
procedure DimVector(var V : TVector; Ub : Integer);
procedure DimVector(var V : TIntVector; Ub : Integer);
procedure DimVector(var V : TCompVector; Ub : Integer);
procedure DimVector(var V: TRealPointVector; Ub : Integer);
procedure DimVector(var V : TBoolVector; Ub : Integer);
procedure DimVector(var V : TStrVector; Ub : Integer);
overload;
```

Description Creates a vector V[0..Ub] based on a type corresponding to a parameter (Float, Integer, Complex, RealPoint, Boolean; Strings. In DMath, procedures with different names were used; LMath uses overloading).

DimMatrix

LMath

Declaration
```
procedure DimMatrix(var A : TMatrix; Ub1, Ub2 : Integer);
procedure DimMatrix(var A : TIntMatrix; Ub1, Ub2 : Integer);
procedure DimMatrix(var A : TCompMatrix; Ub1, Ub2 : Integer);
procedure DimMatrix(var A : TBoolMatrix; Ub1, Ub2 : Integer);
procedure DimMatrix(var A : TStrMatrix; Ub1, Ub2 : Integer);
overload;
```

Description Creates a matrix A[0..Ub1, 0..Ub2] of corresponding type (Float; Integer; Complex; Boolean; String. In DMath, procedures with different names were used; LMath uses overloading).

SetEpsilon

LMath

Declaration `procedure SetEpsilon(AEpsilon: float);`

Description Sets default epsilon for SameValue

SetZeroEpsilon

LMath

Declaration `procedure SetZeroEpsilon(AZeroEpsilon: Float);`

Description Sets default epsilon for comparison of a number to rero (IsZero function) and to compare two numbers near zero.

2.3 Unit uErrors

2.3.1 Constants

uerrorError constants are defined here.

Error codes for mathematical functions:

MathOK, FOk, MatOk, OptOK No error

FDomain Argument domain error

FSing Function singularity

FOverflow Overflow range error

FUnderflow Underflow range error

FTLoss Total loss of precision

FPLoss Partial loss of precision

Error codes for matrix operations:

MatNonConv Non-convergence

MatSing Quasi-singular matrix

MatErrDim Non-compatible dimensions

MatNotPD Matrix not positive definite

Error codes for optimization and nonlinear equations:

OptNonConv Non-convergence

OptSing Quasi-singular hessian matrix

OptBigLambda Too high Marquardt parameter

Error codes for nonlinear regression

NLMaxPar Maximal number of parameters exceeded

NLNullPar Initial parameter equal to zero

Error codes for COBYLA algorithm: LMath 0.5

cobMaxFunc Return from subroutine Cobyla because the maxfun limit has been reached

cobRoundErrors Return from procedure Cobyla because rounding errors are becoming damaging.

cobDegenerate Degenerate gradient

Error codes for linear programming: LMath 0.5

lpBadConstraintCount Bad input constraint counts

lpBadSimplexTableau Bad input tableau

lpBadVariablesCount Bad variables count

File operations error:

lmFileError File input/output error

DFT errors: LMath 0.5

lmDFTError Internal DFT Error

lmDSPFilterWinError Filter window is longer than data

lmTooHighFreqError Cutoff frequency exceeds 0.5 of sampling rate

lmPolesNumError Number of poles for Chebyshev filter not in [2,4,6,8,10]

lmFFTError FFT: number of samples is not power of two

lmFFTBadRipple Ripple setting for Chebyshev filter is not in allowed range (must be 0 to 29%)

lmStatEmptyData Zero length array or array containing only undefined values passed to a statistical procedure

ErrorMessage

LMath

Declaration
```
ErrorMessage : array[0..MaxErrIndex] of String = ('No
error',
'Argument domain error',
'Function singularity',
'Overflow range error',
'Underflow range error',
'Total loss of precision',
'Partial loss of precision',
'Non-convergence',
'Quasi-singular matrix',
'Non-compatible dimensions',
'Matrix not positive definite',
'Non-convergence',
'Quasi-singular hessian matrix',
'Too high Marquardt parameter',
'Max. number of parameters exceeded',
'Initial parameter equal to zero',
'Return from subroutine Cobyla because the maxfun limit has
been reached',
'Return from procedure Cobyla because rounding errors are
becoming damaging.',
'Degenerate gradient',
'LinSimplex:bad input constraint counts',
'Bad input tableau in LinSimplex',
'LinSimplex: bad variables count',
'Internal DFT Error',
'Filter window is longer than data',
'Cutoff frequency must not exceed 0.5 of sampling rate',
'Number of poles must be even',
'FFT: number of samples must be power of two',
'Ripple must be 0 to 29%'
'Empty Dataset', );
```

Description Array of messages corresponding to standard error codes defined in this unit. Elements of this array are returned by MathErrMessage function.

2.3.2 Functions and Procedures

SetErrCode

LMath

Declaration
```
procedure SetErrCode(ErrCode : Integer; EMessage:string =
'');
```

Description Sets the error code and optionally error message. If error message is empty and `ErrorCode` is one of standard codes defined in this unit, corresponding standard message from ErrorMessage array is used. Optional argument EMessage was introduced in LMath.

DefaultVal

LMath

Declaration `function DefaultVal(ErrCode : Integer; DefVal : Float; EMessage:string = '') : Float;`

Description Sets error code and default function value. Optional argument EMessage introduced in LMath.

MathErr

Declaration `function MathErr : Integer;`

Description Returns error code.

MathErrMessage

LMath

Declaration `function MathErrMessage : string;`

Description Returns error message.

2.4 Unit uminmax

2.4.1 Description

Minimum, maximum, sign and exchange.

2.4.2 Functions and Procedures

Min

LMath

Declaration `function Min(X, Y : Float) : Float; overload;`
`function Min(X, Y : Integer) : Integer; overload;`

Description Minimum of 2 values. Universal Min function introduced by LMath in place of IMin and FMin which were used in DMath.

Max

LMath

Declaration `function Max(X, Y : Float) : Float; overload;`
`function Max(X,Y : integer) : integer; overload;`

Description Maximum of 2 values. Minimum of 2 values. Universal Max function introduced by LMath in place of IMax and FMax which were used in DMath.

Sgn

Declaration `function Sgn(X : Float) : Integer; overload;`

Description Sign (returns 1 if $X \geq 0$, -1 otherwise.)

Sgn0

Declaration `function Sgn0(X : Float) : Integer;`

Description Sign (returns 1 if $X > 0$; 0 if $X = 0$ (using IsZero); and -1 if $X < 0$)

DSgn

Declaration `function DSgn(A, B : Float) : Float;`

Description if b is negative, result is -A otherwize result is A.

Sign

LMath

Declaration `function Sign(X: Float):integer`

Description Compatibility with Math unit. Same as `Sgn0`.

IsNegative, IsPositive

Declaration
```
function IsNegative(X: float):boolean;
function IsNegative(X: Integer):boolean;
function IsPositive(X: float):boolean;
function IsPositive(X: Integer):boolean;
```

Description With float, IsPositive returns true if X > DefaultZeroEpsilon and IsNegative returns true if X < -DefaultZeroEpsilon.

InRange

LMath

Declaration
```
function InRange(AVal, AMin, AMax:float):boolean;
function InRange(AVal, AMin, AMax:integer):boolean;
```

description Returns `True` if `AVal` is within the range defined by `AMin,AMax` including the borders. Sets `FDomain` Error and returns `False` if $AMin > AMax$.

Swap

LMath

Declaration
```
procedure Swap(var X, Y : Float); overload;
procedure Swap(var X, Y : Integer); overload;
```

Description Exchange 2 arguments.

2.5 Unit uround

2.5.1 Description

Rounding functions. Based on FreeBASIC version contributed by R. Keeling

2.5.2 Functions and Procedures

RoundTo

Declaration `function RoundTo(X : Float; N : Integer) : Float;`

Description Rounds X to N decimal places

Ceil

Declaration `function Ceil(X : Float) : Integer;`

Description Ceiling function

Floor

Declaration `function Floor(X : Float) : Integer;`

Description Floor function

2.6 Unit umath

2.6.1 Description

Logarithms, exponentials and power

2.6.2 Functions and Procedures

Expo

Declaration `function Expo(X : Float) : Float;`

Description Exponential

Exp2

Declaration `function Exp2(X : Float) : Float;`

Description 2^X

Exp10

Declaration `function Exp10(X : Float) : Float;`

Description 10^X

Log

Declaration `function Log(X : Float) : Float;`

Description Natural logarithm

Log2

Declaration `function Log2(X : Float) : Float;`

Description $\log_2(X)$

Log10

Declaration `function Log10(X : Float) : Float;`

Description Decimal logarithm

LogA

Declaration `function LogA(X, A : Float) : Float;`

Description $log_A(X)$

IntPower

Declaration `function IntPower(X : Float; N : Integer) : Float;`

Description X^N, N is integer.

Power

Declaration `function Power(X, Y : Float) : Float;`

Description X^Y, $X \geq 0$

Operators

Operator ** (power) is defined. Base is float, exponent may be integer or float. LMath

2.7 Unit ugamma

2.7.1 Description

Gamma function and related functions. Translated from C code in Cephes library (http://www.moshier.net)

2.7.2 Functions and Procedures

SgnGamma

Declaration `function SgnGamma(X : Float) : Integer;`

Description SgnGamma(X : Float) : Integer; Sign of Gamma function

Stirling

Declaration `function Stirling(X : Float) : Float;`

Description Stirling(X : Float) : Float; Stirling's formula for the Gamma function

StirLog

Declaration `function StirLog(X : Float) : Float;`

Description StirLog(X : Float) : Float; Approximate Ln(Gamma) by Stirling's formula, for X >= 13

Gamma

Declaration `function Gamma(X : Float) : Float;`

Description Gamma(X : Float) : Float; Gamma function

LnGamma

Declaration `function LnGamma(X : Float) : Float;`

Description LnGamma(X : Float) : Float; natural logarithm of Gamma function

2.8 Unit uigamma

2.8.1 Description

Incomplete Gamma function and related functions. Translated from C code in (Cephes library).

2.8.2 Functions and Procedures

IGamma

Declaration `function IGamma(A, X : Float) : Float;`

Description Incomplete Gamma function.

JGamma

Declaration `function JGamma(A, X : Float) : Float;`

Description Complement of incomplete Gamma function

Erf

Declaration `function Erf(X : Float) : Float;`

Description Error function

Erfc

Declaration `function Erfc(X : Float) : Float;`

Description Complement of error function

2.9 Unit udigamma

2.9.1 Description

DiGamma and TriGamma functions.
Contributed by Philip Fletcher (FLETCHP@WESTAT.com)

2.9.2 Functions and Procedures

DiGamma

Declaration `function DiGamma(X : Float ) : Float;`

Description Calculates the Digamma or Psi function =

$$\frac{d(\ln(\Gamma(X)))}{dX}$$

Parameters: Input, real X, the argument of the Digamma function, $X > 0$. Output, real Digamma, the value of the Digamma function at X.

TriGamma

Declaration `function TriGamma(X : Float ) : Float;`

Description Trigamma calculates the Trigamma or Psi Prime function =

$$\frac{d^2(\ln(\Gamma(X)))}{(dX)^2}$$

2.10 Unit ubeta

2.10.1 Description

Beta function

2.10.2 Functions and Procedures

Beta

Declaration `function Beta(X, Y : Float) : Float;`

Description Beta(X, Y : Float) : Float; Computes Beta(X, Y) =

$$\frac{\Gamma(X) \cdot \Gamma(Y)}{\Gamma(X+Y)}$$

2.11 Unit uibeta

2.11.1 Description

Incomplete Beta function.
Translated from C code in Cephes library (http://www.moshier.net)

2.11.2 Functions and Procedures

IBeta

Declaration `function IBeta(A, B, X : Float) : Float;`

Description Incomplete Beta function

2.12 Unit ulambert

2.12.1 Description

Lambert's function
Translated from Fortran code by Barry et al. (http://www.netlib.org/toms/743)

2.12.2 Functions and Procedures

LambertW

Declaration `function LambertW(X : Float; UBranch, Offset : Boolean) : Float;`

Description Lambert's W function: $Y = \mathrm{W}(X) \Longrightarrow X = Y\,\mathrm{e}^{Y}, \quad X \geq -1/\mathrm{e}$
X is Lambert's function argument;
UBranch must be set to True for computing the upper branch:
$(X \geq -1/\mathrm{e},\ \mathrm{W}(X) \geq -1)$;
UBranch is false for computing the lower branch:
$(-1/\mathrm{e} \leq X < 0,\ \mathrm{W}(X) \leq -1)$;
Offset must be set to true for computing $\mathrm{W}(X - 1/\mathrm{e}), \quad X \geq 0$, False for computing W(X).

2.13 Unit ufact

2.13.1 Description

Factorial

2.13.2 Functions and Procedures

Fact

Declaration `function Fact(N : Integer) : Float;`

Description Fact(N : Integer) : Float; N!

2.14 Unit utrigo

2.14.1 Description

Trigonometric functions

2.14.2 Functions and Procedures

Pythag

Declaration `function Pythag(X, Y : Float) : Float;`

Description $\sqrt{X^2 + Y^2}$

FixAngle

Declaration `function FixAngle(Theta : Float) : Float;`

Description Set Theta in $-\pi..\pi$

Tan

Declaration `function Tan(X : Float) : Float;`

Description Tangent

ArcSin

Declaration `function ArcSin(X : Float) : Float;`

Description Arc sinus

ArcCos

Declaration `function ArcCos(X : Float) : Float;`

Description Arc cosinus

ArcTan2

Declaration `function ArcTan2(Y, X : Float) : Float;`

Description Angle (Ox, OM) with M(X,Y)

2.15 Unit uhyper

2.15.1 Description

Hyperbolic functions

2.15.2 Functions and Procedures

Sinh

Declaration `function Sinh(X : Float) : Float;`

Description Hyperbolic sine

Cosh

Declaration `function Cosh(X : Float) : Float;`

Description Hyperbolic cosine

Tanh

Declaration `function Tanh(X : Float) : Float;`

Description Hyperbolic tangent

ArcSinh

Declaration `function ArcSinh(X : Float) : Float;`

Description Inverse hyperbolic sine

ArcCosh

Declaration `function ArcCosh(X : Float) : Float;`

Description Inverse hyperbolic cosine

ArcTanh

Declaration `function ArcTanh(X : Float) : Float;`

Description Inverse hyperbolic tangent

SinhCosh

Declaration `procedure SinhCosh(X : Float; out SinhX, CoshX : Float);`

Description Sinh & Cosh

2.16 Unit ucomplex

2.16.1 Description

Complex number library
Based on ComplexMath Delphi library by E. F. Glynn
http://www.efg2.com/Lab/Mathematics/Complex/index.html
Later ideas from uComplex unit by Pierre Müller were used.

2.16.2 Operators

LMath

Following operators over complex numbers or real and complex numbers defined: :=, +, -, *, /, =.

2.16.3 Procedures and functions

Cmplx

Declaration `function Cmplx(X, Y : Float) : Complex;`

Description Returns the complex number X + iY

Polar

Declaration `function Polar(R, Theta : Float) : Complex;`

Description Returns a complex number corresponding to polar coordinates (R, Theta): $R(\cos(\theta) + i\,\sin(\theta))$

CFloat

Declaration `function CReal(Z : Complex) : Float;`

Description Returns the real part of Z

CImag

Declaration `function CImag(Z : Complex) : Float;`

Description Returns the imaginary part of Z

CSgn

Declaration `function CSgn(Z : Complex) : Integer;`

Description Complex sign

Swap

Declaration `procedure Swap(var X, Y : Complex); overload;`

Description Exchanges two complex numbers

SameValue

Declaration `function samevalue(z1, z2 : complex) : boolean; overload;`

Description compares two compex numbers using relative epsilon

CAbs

Declaration `function CAbs(Z : Complex) : Float;`

Description Modulus of Z

CAbs2

Declaration `function CAbs2(Z : Complex) : Float;`

Description Squared modulus of Z

CArg

Declaration `function CArg(Z : Complex) : Float;`

Description Argument of Z

CConj

Declaration `function CConj(Z : Complex) : Complex;`

Description Complex conjugate

CSqr

Declaration `function CSqr(Z : Complex) : Complex;`

Description Complex square

CInv

Declaration `function CInv(Z : Complex) : Complex;`

Description Complex inverse

CSqrt

Declaration `function CSqrt(Z : Complex) : Complex;`

Description Principal part of complex square root

CLn

Declaration `function CLn(Z : Complex) : Complex;`

Description Principal part of complex logarithm

CExp

Declaration `function CExp(Z : Complex) : Complex;`

Description Complex exponential

CRoot

Declaration `function CRoot(Z : Complex; K, N : Integer) : Complex;`

Description All N-th roots: $Z^{1/N}$, $K = 0..N-1$

CPower

Declaration `function CPower(A, B : Complex) : Complex;`

Description Power with complex exponent

CIntPower

Declaration `function CIntPower(A : Complex; N : Integer) : Complex;`

Description Power with integer exponent

CRealPower

Declaration `function CRealPower(A : Complex; X : Float) : Complex;`

Description Power with Float exponent

CPoly

Declaration `function CPoly(Z : Complex; Coef : TVector; Deg : Integer) : Complex;`

Description Evaluate polynom with compex argument

CSin

Declaration `function CSin(Z : Complex) : Complex;`

Description Complex sine

CCos

Declaration `function CCos(Z : Complex) : Complex;`

Description Complex cosine

CSinCos

Declaration `procedure CSinCos(Z : Complex; out SinZ, CosZ : Complex);`

Description Complex sine and cosine

CTan

Declaration `function CTan(Z : Complex) : Complex;`

Description Complex tangent

CArcSin

Declaration `function CArcSin(Z : Complex) : Complex;`

Description Complex arc sine

CArcCos

Declaration `function CArcCos(Z : Complex) : Complex;`

Description Complex arc cosine

CArcTan

Declaration `function CArcTan(Z : Complex) : Complex;`

Description Complex arc tangent

CSinh

Declaration `function CSinh(Z : Complex) : Complex;`

Description Complex hyperbolic sine

CCosh

Declaration `function CCosh(Z : Complex) : Complex;`

Description Complex hyperbolic cosine

CSinhCosh

Declaration `procedure CSinhCosh(Z : Complex; out SinhZ, CoshZ : Complex);`

Description Complex hyperbolic sine and cosine

CTanh

Declaration `function CTanh(Z : Complex) : Complex;`

Description Complex hyperbolic tangent

CArcSinh

Declaration `function CArcSinh(Z : Complex) : Complex;`

Description Complex hyperbolic arc sine

CArcCosh

Declaration `function CArcCosh(Z : Complex) : Complex;`

Description Complex hyperbolic arc cosine

CArcTanh

Declaration `function CArcTanh(Z : Complex) : Complex;`

Description Complex hyperbolic arc tangent

CLnGamma

Declaration `function CLnGamma(Z : Complex) : Complex;`

Description Logarithm of Gamma function

2.17 Unit uIntervals

LMath

2.17.1 Description

This unit defines type TInterval which represents an interval on a real numbers axis and defines functions to find if a given value belongs to the interval, if two intervals intersect and to find the intersection, or if one interval is completely contained in another one. Length method of TInterval record returnes the difference between borders of the interval. This entire unit was introduced in LMath.

2.17.2 Types

TInterval record

Declaration

```
TInterval = record
     Lo:Float;
     Hi:Float;
     function Length:float
end;
```

2.17.3 Functions and Procedures

IntervalsIntersect

Declaration

```
function IntervalsIntersect
              (Lo1, Hi1, Lo2, Hi2:Float):boolean;

function IntervalsIntersect
              (Lo1, Hi1, Lo2, Hi2:Integer):boolean;

function IntervalsIntersect
              (Interval1, Interval2:TInterval):boolean;
```

Description returns `true` if intervals `[Lo1; Hi1]` and `[Lo2; Hi2]` or `Interval1, Interval2` intersect.

Contained

Declaration `function Contained(ContainedInterval, ContainingInterval : TInterval) : boolean;`

Description True if `ContainedInterval` is completely inside containing i.e. `ContainedInterval.Lo` > `ContainingInterval.Lo and ContainedInterval.Hi` < `ContainingInterval.Hi`

Intersection

Declaration `function Intersection(Interval1, Interval2 : TInterval) : TInterval;`

Description Returns intersection of `Interval1` and `Interval2`. If no intersection, result is (0;0).

Inside

Declaration `function Inside(V:Float; AInterval:TInterval):boolean;`
`function Inside(V:float; ALo, AHi:float):boolean; overload;`

Description True if `V` is inside interval defined by its borders (`ALo, AHi`, similar to Math.InRange) or by `AInterval`.

IntervalDefined

Declaration `function IntervalDefined(AInterval:TInterval):boolean;`

Description True if `AInterval.Lo` < `AInterval.Hi`.

DefineInterval

Declaration `function DefineInterval(ALo,AHi:Float):TInterval;`

Description Constructor of TInterval from ALo and AHi.

MoveInterval

Declaration `function MoveInterval(V:float; var AInterval:TInterval);`

Description Move interval *by* a value (it is added to both Lo and Hi).

MoveIntervalTo

Declaration `procedure MoveIntervalTo(V:Float; var AInterval:TInterval);`

Description Move interval *to* a value (Lo is set to this value, Hi adjusted such that length remains constant).

2.18 Unit uRealPoints

LMath

2.18.1 Description

uRealPoints introduces operations over TRealPoint as over vectors on 2-dimensional Cartesian plane. Introduced in LMath.

2.18.2 Operators

Following operators are defined: +, -, * (scalar multiplication); * (dot product).

2.18.3 Functions and Procedures

SameValue

Declaration `function SameValue(P1,P2:TRealPoint; epsilonX:float = -1;`
`epsilonY:float = -1):boolean; overload; inline;`

Description Comparison of TRealPoint using epsilon; epsilon for X and for Y are defined separately. If epsilon is -1, default value as defined by SetEpsilon will be used. If SetEpsilon was not used, it is MachEp

rpPoint

Declaration `function rpPoint(AX, AY:float):TRealPoint;`

Description Constructor of TRealPoint from two floats.

rpSum

Declaration `function rpSum(P1,P2:TRealPoint):TRealPoint;`

Description Summation of TRealPoint.

rpSubtr

Declaration `function rpSubtr(P1, P2:TRealPoint):TRealPoint;`

Description Subtraction of TRealPoint.

rpMul

Declaration `function rpMul(P:TRealPoint; S:Float):TRealPoint;`

Description Multiplication of TRealPoint by Scalar.

rpDot

Declaration `function rpDot(P1, P2:TRealPoint):Float;`

Description Dot product of TRealPoint.

rpLength

Declaration `function rpLength(P:TRealPoint):Float;`

Description Length of vector, represented by TRealPoint.

Distance

Declaration `function Distance(P1, P2:TRealPoint):Float;`

Description Distance between two TRealPoint on cartesian plane.

2.19 Unit uIntPoints

LMath

2.19.1 Description

Unit `uIntPoints` introduces operations over `TIntegerPoint`

2.19.2 Procedures and functions

ipPoint

Declaration `function ipPoint(AX, AY:integer):TIntegerPoint;`

Description Constructor of `TIntegerPoint` from two integers.

ipSum,ipSubtr

Declaration `function ipSum(P1,P2:TIntegerPoint):TIntegerPoint;`
`function ipSubtr(P1, P2:TIntegerPoint):TIntegerPoint;`

Description Element-wise subtraction and summation of `TIntegerPoint`.

ipMul

Declaration `function ipMul(P:TIntegerPoint; S:integer):TIntegerPoint;`

Description Multiply elements of TIntegerPoint with Float.

2.19.3 Operators

+,-

Summation or substraction of two TIntegerPoint.

*

Multiply components of TIntegerPoint with a Float.

2.20 Unit uScaling

2.20.1 Description

Compute an appropriate interval for a set of values.

2.20.2 Functions and Procedures

FindScale

Declaration `procedure FindScale(X1, X2 : Float; MinDiv, MaxDiv : Integer; out Min, Max, Step : Float);`

Description Determines an interval [Min, Max] including the values from X1 to X2, and a subdivision Step of this interval. Input parameters : X1, X2 = min. & max. Values to be included MinDiv = minimum nb of subdivisions MaxDiv = maximum nb of subdivisions. Output parameters : Min, Max, Step.

AutoScale

Declaration `procedure AutoScale(X : TVector; Lb, Ub : Integer; Scale : TScale; out XMin, XMax, XStep : Float);`

Description Finds an appropriate scale for plotting the data in X[Lb..Ub]

Chapter 3

Package lmLinearAlgebra: Operations with Vectors and Matrices

3.1 Description

This package includes procedures for linear algebra: operations with vectors, matrices, systems of linear equations.

3.2 Unit uMatrix

LMath 0.5

3.2.1 Description

LMath 0.6

This unit introduces several basic functions and defines several operators over vectors and matrices. Functions which have `Ziel = nil` default parameter, place result in `Ziel` beginning from `Ziel[ResLb]` and return it as a result. If `Ziel` is nil, it is allocated, otherwise it must have a size sufficient to accomodate the result. For vectors, length of `Ziel` must be at least `Ub - Lb + ResLb`, for matrices it must be at least `[Lb..Ub1,Lb..Ub2]`. Note that `ResLb` default value is 1. Reason is that many routines in DMath and LMath were written originally in Fortran, and its inheritance makes us to use arrays beginning from index 1.

Modern Pascal allows to define open array parameters and, importantly, pass subarrays to the procedures and functions. Use of open arrays allows to make calls more concise when a complete array must be passed, yet even more flexible when only partial array is needed. Beginning from LMath Ver.0.6, attempt is made to develop, along with old `TVector, Lb, Ub` convention, a new, more clear and flexible syntax using open arrays in place of `TVector` as formal parameters. At this stage this feature is only developing and experimental.

What is important, these two forms can be different not only syntactically, but semantically as well. Let's consider two functions:
`function SomeFunc(V:TVector; Lb, Ub:integer):TVector;` and
`function SomeFunc(V:array of float):TVector`
which somehow transform passed array and return result of the transformation. Now let's consider following calls:
`V2 := SomeFunc(V,2,7);`
and
`V3 := Somefunc(V[2..7])`
In the first case the whole array is passed to the function and and returned, but transformaton is applied only elements from 2 to 7; elements 0 and 1, as well as everything after 7, are returned unchanged. In the second case, the function receives only elements 2..7 and only they are returned. To avoid this issue, in some cases functions were converted to procedures with output parameters made also open arrays. Advantage of this approach is that result of the transformation can be assigned to static arrays or subarrays; disadvantage, of course, that procedures cannot be used as parts of expressions.

Finally, two-dimentional arrays cannot be passed as open arrays, therefore only dynamic array functions for `TMatrix` exist.

3.2.2 Functions and Procedures

VecFloatAdd,VecFloatSubtr,VecFloatMul,VecFloatDiv (for TVector)

Declaration
```
function VecFloatAdd(V:TVector; R:Float; Lb, Ub : integer;
Ziel : TVector = nil; ResLb : integer = 1): TVector;
function VecFloatSubtr(V:TVector; R:Float; Lb, Ub : integer;
Ziel : TVector = nil; ResLb : integer = 1): TVector;
function VecFloatDiv(V:TVector; R:Float; Lb, Ub : integer;
Ziel : TVector = nil; ResLb : integer = 1): TVector;
function VecFloatMul(V:TVector; R:Float; Lb, Ub : integer;
Ziel : TVector = nil; ResLb : integer = 1): TVector;
```

Description Every of these functions adds, subtracts, multiplies or divides every element of the vector `V` with float `R`, beginning from element `Lb` and to element `Ub`. To modify original `V`, make `Ziel = V`.

VecFloatAdd,VecFloatSubtr,VecFloatMul,VecFloatDiv (for Open Array)

LMath 0.6

Declaration
```
procedure VecFloatAdd (V:array of float; R:Float; var Ziel :
array of float); overload;
procedure VecFloatSubtr(V:array of float; R:Float; var Ziel :
array of float); overload;
procedure VecFloatDiv (V:array of float; R:Float; var Ziel :
array of float); overload;
procedure VecFloatMul (V:array of float; R:Float; var Ziel :
array of float); overload;
```

Description These functions add, subtract, multiply or divide every element of the input array (`V`) and place result into output array (`Ziel`). Dynamic, static or partial these functions adds, subtracts, multiplies or divides every element of the vector arrays may be used both for input and output. For example, `VecFloatDiv(V[5..9], 4, V[10..14])` is a valid call. Input and output arrays must be of equal length, otherwise error `MatErrDim` is set and output array is not modified.

MatFloatAdd,MatFloatSubtr,MatFloatMul,MatFloatDiv

Declaration
```
function MatFloatAdd(M:TMatrix; R:Float; Lb, Ub1, Ub2 :
integer; Ziel : TMatrix = nil) : TMatrix;
function MatFloatSubtr(M:TMatrix; R:Float; Lb, Ub1, Ub2 :
integer; Ziel : TMatrix = nil) : TMatrix;
function MatFloatDiv(M:TMatrix; R:Float; Lb, Ub1, Ub2 :
integer; Ziel : TMatrix = nil) : TMatrix;
function MatFloatMul(M:TMatrix; R:Float; Lb, Ub1, Ub2 :
integer; Ziel : TMatrix = nil) : TMatrix;
```

Description These functions are similar to `VecFloat*`, but are defined for matrix. Operation is done element-wize for submatrix `M[Lb..Ub1,Lb..Ub2]`.

VecAdd,VecSubtr,VecElemMul,VecDiv

Declaration `procedure VecAdd(V1,V2:array of float; var Ziel : array of float);`
`procedure VecSubtr(V1,V2:array of float; var Ziel : array of float);`
`procedure VecElemMul(V1,V2:array of float; var Ziel : array of float);`
`procedure VecDiv(V1,V2:array of float; var Ziel : array of float);`

Description Another set of element-wise procedures. This time, element of second array is added to, substracted from, multiplied with or is a divider of each element of a first array. Result is placed to `Ziel` array, which can be a subarray. `V1`, `V2` and `Ziel` must have equal dimentions, otherwise error `MatErrDim` is set. In `VecDiv`, if any of `V2` elements is zero, `FDomain` error is set and `Ziel` remains unmodified.

VecDotProd

Declaration `function VecDotProd(V1,V2:TVector; Lb, Ub : integer) : float;`
`function VecDotProd(const V1,V2:array of float) : float;`

Description Dot product of the vectors `V1` and `V2`.

VecOuterProd

Declaration `function VecOuterProd(V1, V2:TVector; Lb, Ub1, Ub2 : integer; Ziel : TMatrix = nil):TMatrix;`
`function VecOuterProd(const V1, V2:array of float; Ziel : TMatrix = nil):TMatrix;`

Description Outer product of the vectors `V1` and `V2`. Result is placed into `Ziel`; if on input is `nil`, it is allocated. Length of `Ziel` column must be equal to length of the first vector and length of `Ziel row` must be equal to the length of second vector, otherwise `MatErrDim` error is set and `nil` is returned.

VecCrossProd

Declaration `function VecCrossProd(V1, V2:TVector; Lb: integer; Ziel :TVector = nil):TVector;`
`procedure VecCrossProd(V1, V2: array of float; var Ziel: array of float);`

Description Cross product of two vectors. Function puts result into `Ziel` if it is supplied, otherwise allocates it itself. For procedure call, `Ziel` must exist. Use of subarrays is possible both for input and output arrays. The procedure checks length of all arrays and sets error `MaterrDim` if any of them is different from 3.

VecEucLength

Declaration `function VecEucLength(V:TVector; LB, Ub : integer) : float;`
`function VecEucLength(const V: array of float): float;`

Description Returns euclidian length of a vector (dot product with itself).

MatVecMul

Declaration `function MatVecMul(M:TMatrix; V:TVector; LB: integer; Ziel: TVector = nil): TVector;`
`procedure MatVecMul(M: TMatrix; V: array of float; var Ziel: array of float);`

Description Product of matrix `M` and vector `V`. Length of matrix rows must be equal to length of `V`.

MatMul

Declaration `MatMul(A, B : TMatrix; LB : integer; Ziel : TMatrix = nil) : TMatrix;`

Description Product of matrix `A` with matrix `B`. Length of row in B must be equal to length of column in `A`.

MatTranspose

Declaration `function MatTranspose(M:TMatrix; LB: integer; Ziel: TMatrix = nil): TMatrix;) : TMatrix;`

Description Transposes a matrix `M`. If `Ziel` is not `nil`, length of its columns must be equal to length of rows in `M` and vice versa.

MatTransposeInPlace

Declaration `procedure MatTransposeInPlace(M:TMatrix; Lb, Ub : integer);`

Description Transpose square matrix and place result in itself.

3.2.3 Operators

Operators `+,-,*,/` are defined for `array of float` and `Float` as well as for `TMatrix` and `Float`. They add the float to every element of the vector or matrix.

Operators `+,-` are defined for two arrays of float. They must be of equal length; these are element-wize operations too. Result of operation is `TVector`, which is allocated by the operator.

Vector operators work with subarrays, thus, line

```
NewVect := V1[5..8]+V2[7..9]
```

is valid and produces expected result. Unfortunately, only dynamic arrays can be result of an operation; no static arrays and no subarrays.

3.3 Unit ugausjor

3.3.1 Description

Solution of a system of linear equations by Gauss-Jordan method

3.3.2 Functions and Procedures

GaussJordan

Declaration `procedure GaussJordan(A : TMatrix; Lb, Ub1, Ub2 : Integer; var Det : Float);`

Description Transforms a matrix according to the Gauss-Jordan method.

Input parameters: `A` = system matrix; `Lb` = lower matrix bound in both dimentions; `Ub1, Ub2` = upper matrix bounds.

Output parameters: `A` = transformed matrix; `Det` = determinant of `A`.

Possible results: `MatOk`: No error; `MatErrDim`: Non-compatible dimensions; `MatSing`: Singular matrix.

3.4 Unit ulineq

3.4.1 Description

Solution of a system of linear equations with a single constant vector by Gauss-Jordan method.

3.4.2 Functions and Procedures

LinEq

Declaration `procedure LinEq(A : TMatrix; B : TVector; Lb, Ub : Integer; out Det : Float);`

Description Solves a linear system with the Gauss-Jordan method.

Input parameters: `A` = system matrix; `B` = constant vector; `Lb, Ub` = lower and upper array bounds.

Output parameters: `A` = inverse matrix; `B` = solution vector; `Det` = determinant of `A`.

Possible results: `MatOk`: No error; `MatSing`: Singular matrix.

3.5 Unit ubalance

3.5.1 Description

Balances a matrix and tries to isolate eigenvalues.

3.5.2 Functions and Procedures

Balance

Declaration `procedure Balance( A : TMatrix; Lb, Ub : Integer; out I_low, I_igh : Integer; Scale : TVector);`

Description A contains the input matrix to be balanced. `Lb, Ub` are the lowest and highest indices of the elements of `A`. On output: `A` contains the balanced matrix. `I_low` and `I_igh` are two integers such that $\mathbf{A}_{i,j}$ is equal to zero if (1) $i > j$ and (2) $j \in Lb, \ldots, I\_low{-}1$ or $i \in I\_igh+1, \ldots, Ub$.

Scale contains information determining the permutations and scaling factors used.

3.6 Unit ubalbak

3.6.1 Description

Back transformation of eigenvectors.

3.6.2 Functions and Procedures

BalBak

Declaration `procedure BalBak(Z : TMatrix; Lb, Ub, I_low, I_igh : Integer; Scale : TVector; M : Integer);`

Description This procedure is a translation of the EISPACK subroutine Balbak. This procedure forms the eigenvectors of a real general matrix by back transforming those of the corresponding balanced matrix determined by Balance.

On input: `Z` contains the real and imaginary parts of the eigenvectors to be back transformed. `Lb, Ub` are the lowest and highest indices of the elements of `Z` `I_low` and `I_igh` are integers determined by `Balance`. `Scale` contains information determining the permutations and scaling factors used by `Balance`. `M` is the index of the latest column of `Z` to be back transformed.

On output: `Z` contains the real and imaginary parts of the transformed eigenvectors in its columns Lb..M.

3.7 Unit ucholesk

3.7.1 Description

Cholesky factorization of a positive definite symmetric matrix

3.7.2 Functions and Procedures

Cholesky

Declaration `procedure Cholesky(A, L : TMatrix; Lb, Ub : Integer);`

Description Cholesky decomposition. Factors the symmetric positive definite matrix **A** as a product **LL'** where **L** is a lower triangular matrix. This procedure may be used as a test of positive definiteness.

Possible results: `MatOk`: No error; `MatNotPD`: Matrix not positive definite.

3.8 Unit uelmhes

3.8.1 Description

Reduction of a square matrix to upper Hessenberg form.

3.8.2 Functions and Procedures

ElmHes

Declaration `procedure ElmHes(A : TMatrix; Lb, Ub, I_low, I_igh : Integer; I_int : TIntVector);`

3.9 Unit ueltran

3.9.1 Description

Save transformations used by ElmHes.

3.9.2 Functions and Procedures

Eltran

Declaration `procedure Eltran(A : TMatrix; Lb, Ub, I_low, I_igh : Integer; I_int : TIntVector; Z : TMatrix);`

Description On input:

`A` contains the multipliers which were used in the reduction by `Elmhes` in its lower triangle below the subdiagonal.

`Lb, Ub` are the lowest and highest indices of the elements of `A`.

`I_low` and `I_igh` are integers determined by the balancing procedure Balance. If Balance has not been used, set `I_low=Lb`, `I_igh=Ub`.

`I_int` contains information on the rows and columns interchanged in the reduction by `Elmhes`. Only elements I_`low` through `I_igh` are used.

On output:

`Z` contains the transformation matrix produced in the reduction by `Elmhes`.

3.10 Unit uhqr

3.10.1 Description

Eigenvalues of a real upper Hessenberg matrix by the QR method.

3.10.2 Functions and Procedures

Hqr

Declaration `procedure Hqr(H : TMatrix; Lb, Ub, I_low, I_igh : Integer; Lambda : TCompVector);`

Description On input:

`H` contains the upper Hessenberg matrix.

`Lb, Ub` are the lowest and highest indices of the elements of `H`.

`I_low` and `I_igh` are integers determined by the balancing subroutine Balance. If `Balance` has not been used, set `I_low` = `Lb`, `I_igh` = `Ub`.

On output:

H has been destroyed.

`Wr` and `Wi` contain the real and imaginary parts, respectively, of the eigenvalues. The eigenvalues are unordered except that complex conjugate pairs of values appear consecutively with the eigenvalue having the positive imaginary part first.

The function returns an error code: `MathOK` for normal return, -j if the limit of $30N$ iterations is exhausted while the j-th eigenvalue is being sought. (N being the size of the matrix). The eigenvalues should be correct for indices j+1,...,`Ub`.

3.11 Unit uhqr2

3.11.1 Description

Eigenvalues and eigenvectors of a real upper Hessenberg matrix.

3.11.2 Functions and Procedures

Hqr2

Declaration `procedure Hqr2(H : TMatrix; Lb, Ub, I_low, I_igh : Integer; Lambda : TCompVector; Z : TMatrix);`

Description On input:

`H` contains the upper Hessenberg matrix.

`Lb, Ub` are the lowest and highest indices of the elements of `H`.

`I_low` and `I_igh` are integers determined by the balancing subroutine Balance. If `Balance` has not been used, set `I_low=Lb`, `I_igh=Ub`.

`Z` contains the transformation matrix produced by `Eltran` after the reduction by `Elmhes`, if performed. If the eigenvectors of the Hessenberg matrix are desired, `Z` must contain the identity matrix.

On output:

`H` has been destroyed.

`Wr` and `Wi` contain the real and imaginary parts, respectively, of the eigenvalues. The eigenvalues are unordered except that complex conjugate pairs of values appear consecutively with the eigenvalue having the positive imaginary part first.

`Z` contains the real and imaginary parts of the eigenvectors. If the i-th eigenvalue is real, the i-th column of `Z` contains its eigenvector. If the i-th eigenvalue is complex with positive imaginary part, the i-th and (i+1)-th columns of `Z` contain the real and imaginary parts of its eigenvector. The eigenvectors are unnormalized. If an error exit is made, none of the eigenvectors has been found.

The function returns an error code: zero for normal return, -j if the limit of $30N$ iterations is exhausted while the j-th eigenvalue is being sought (N being the size of the matrix). The eigenvalues should be correct for indices j+1,...,`Ub`.

3.12 Unit ujacobi

3.12.1 Description

Eigenvalues and eigenvectors of a symmetric matrix

3.12.2 Functions and Procedures

Jacobi

Declaration `procedure Jacobi(A : TMatrix; Lb, Ub, MaxIter : Integer; Tol : Float; Lambda : TVector; V : TMatrix);`

Description Eigenvalues and eigenvectors of a symmetric matrix by the iterative method of Jacobi.

Input parameters: `A` = matrix; `Lb` = index of first matrix element; `Ub` = index of last matrix element; `MaxIter` = maximum number of iterations; `Tol` = required precision.

Output parameters: `Lambda` = eigenvalues in decreasing order; `V` = matrix of eigenvectors (columns).

Possible results: `MatOk`, `MatNonConv`.

The eigenvectors are normalized, with their first component > 0 This procedure destroys the original matrix `A`.

3.13 Unit ulu

3.13.1 Description

LU decomposition

3.13.2 Functions and Procedures

LU_Decomp

Declaration `procedure LU_Decomp(A : TMatrix; Lb, Ub : Integer);`

Description LU decomposition. Factors the square matrix **A** as a product **LU**, where **L** is a lower triangular matrix (with unit diagonal terms) and **U** is an upper triangular matrix. This routine is used in conjunction with `LU_Solve` to solve a system of equations.

Input parameters: `A` = matrix; `Lb` = index of first matrix element; `Ub` = index of last matrix element.

Output parameter: `A` = contains the elements of `L` and `U`.

Possible results: `MatOk`, `MatSing`.

NB: This procedure destroys the original matrix `A`.

LU_Solve

Declaration `procedure LU_Solve(A : TMatrix; B : TVector; Lb, Ub : Integer; X : TVector);`

Description Solves a system of equations whose matrix has been transformed by `LU_Decomp`.

Input parameters: `A` = result from `LU_Decomp`; `B` = constant vector; `Lb, Ub` = as in `LU_Decomp`.

Output parameter: `X` = solution vector.

3.14 Unit uqr

3.14.1 Description

QR decomposition

Ref.: 'Matrix Computations' by Golub & Van Loan Pascal implementation contributed by Mark Vaughan.

3.14.2 Functions and Procedures

QR_Decomp

Declaration `procedure QR_Decomp(A : TMatrix; Lb, Ub1, Ub2 : Integer; R : TMatrix);`

Description QR decomposition. Factors the matrix **A** (n x m, with *ngem*) as a product **QR** where **Q** is a $n \times m$ column-orthogonal matrix, and **R** a $m \times m$ upper triangular matrix. This routine is used in conjunction with `QR_Solve` to solve a system of equations.

Input parameters: `A` = matrix; `Lb` = index of first matrix element; `Ub1` = index of last matrix element in 1st dimention; `Ub2` = index of last matrix element in 2nd dimention.

Output parameter: `A` = contains the elements of **Q**; `R` = upper triangular matrix.

Possible results: `MatOk`, `MatErrDim`, `MatSing`.

NB: This procedure destroys the original matrix `A`.

QR_Solve

Declaration `procedure QR_Solve(Q, R : TMatrix; B : TVector; Lb, Ub1, Ub2 : Integer; X : TVector);`

Description QR decomposition. Factors the matrix **A** ($n \times m$, with $n \geq m$) as a product **QR** where **Q** is a ($n \times m$) column-orthogonal matrix, and **R** a ($m \times m$) upper triangular matrix. This routine is used in conjunction with QR_Solve to solve a system of equations.

Input parameters : `A` = matrix; `Lb` = index of first matrix element; `Ub1` = index of last matrix element in 1st dimention; `Ub2` = index of last matrix element in 2nd dimention.

Output parameter: `A` = contains the elements of **Q**; `R` = upper triangular matrix.

Possible results: `MatOk, MatErrDim, MatSing`.

NB: This procedure destroys the original matrix `A`

3.15 Unit usvd

3.15.1 Description

Singular value decomposition

3.15.2 Functions and Procedures

SV_Decomp

Declaration `procedure SV_Decomp(A : TMatrix; Lb, Ub1, Ub2 : Integer; S : TVector; V : TMatrix);`

Description Singular value decomposition. Factors the matrix **A** ($n \times m$, with $n \geq m$) as a product **USV'** where **U** is a ($n \times m$) column-orthogonal matrix, **S** a ($m \times m$) diagonal matrix with elements ≥ 0 (the singular values) and **V** a ($m \times m$) orthogonal matrix. This routine is used in conjunction with `SV_Solve` to solve a system of equations.

Input parameters: `A` = matrix; `Lb` = index of first matrix element; `Ub1` = index of last matrix element in 1st dimention; `Ub2` = index of last matrix element in 2nd dimention.

Output parameters: `A` = contains the elements of `U`; `S` = vector of singular values; `V` = orthogonal matrix.

Possible results: `MatOk`: No error; `MatNonConv`: Non-convergence; `MatErrDim`: Non-compatible dimensions ($n < m$).

NB: This procedure destroys the original matrix `A`.

SV_SetZero

Declaration `procedure SV_SetZero(S : TVector; Lb, Ub : Integer; Tol : Float);`

Description Sets the singular values to zero if they are lower than a specified threshold.

Input parameters: `S` = vector of singular values; `Tol` = relative tolerance. Threshold value will be $Tol \cdot \mathrm{Max}(S)$; `Lb` = index of first vector element; `Ub` = index of last vector element.

Output parameter : `S` = modified singular values.

SV_Solve

Declaration `procedure SV_Solve(U : TMatrix; S : TVector; V : TMatrix; B : TVector; Lb, Ub1, Ub2 : Integer; X : TVector);`

Description Solves a system of equations by singular value decomposition, after the matrix has been transformed by SV_Decomp, and the lowest singular values have been set to zero by `SV_SetZero`.

Input parameters: `U, S, V` = vector and matrices from `SV_Decomp`; `B` = constant vector; `Lb, Ub1, Ub2` = as in `SV_Decomp`.

Output parameter: `X` = solution vector =

$$\mathbf{V} \cdot \mathbf{Diag}(1/s_i) \cdot \mathbf{U'B}, \text{ for } s_i \neq 0$$

SV_Approx

Declaration `procedure SV_Approx(U : TMatrix; S : TVector; V : TMatrix; Lb, Ub1, Ub2 : Integer; A : TMatrix);`

Description Approximates a matrix **A** by the product **USV'**, after the lowest singular values have been set to zero by SV_SetZero.

Input parameters: `U, S, V` = vector and matrices from `SV_Decomp`; `Lb, Ub1, Ub2` = as in `SV_Decomp`.

Output parameter: `A` = approximated matrix.

3.16 Unit ueigsym

3.16.1 Description

Eigenvalues and eigenvectors of a symmetric matrix (SVD method).

3.16.2 Functions and Procedures

EigenSym

Declaration `procedure EigenSym(A : TMatrix; Lb, Ub : Integer; Lambda : TVector; V : TMatrix);`

Description Eigenvalues and eigenvectors of a symmetric matrix by singular value decomposition.

Input parameters: `A` = matrix; `Lb` = index of first matrix element; `Ub` = index of last matrix element.

Output parameters: `Lambda` = eigenvalues in decreasing order; `V` = matrix of eigenvectors (columns).

Possible results : `MatOk, MatNonConv`.

The eigenvectors are normalized, with their first component > 0. This procedure destroys the original matrix `A`.

3.17 Unit ueigval

3.17.1 Description

Eigenvalues of a general square matrix

3.17.2 Functions and Procedures

EigenVals

Declaration `procedure EigenVals(A : TMatrix; Lb, Ub : Integer; Lambda : TCompVector);`

3.18 Unit ueigvec

3.18.1 Description

Eigenvalues and eigenvectors of a general square matrix.

3.18.2 Functions and Procedures

EigenVect

Declaration `procedure EigenVect(A : TMatrix; Lb, Ub : Integer; Lambda : TCompVector; V : TMatrix);`

Chapter 4

Package lmPolynoms: Units to Solve and Explore Polynomials

4.1 Description

This package contains several units to find polynom roots and critical points, and to evaluate polynomials and rational fractions.

4.2 Unit upolynom

4.2.1 Description

Evaluates polynomials and rational fractions.

4.2.2 Functions and Procedures

Poly

Declaration `function Poly(X : Float; Coef : TVector; Deg : Integer) : Float;`

Description Evaluates the polynomial : $P(X) = Coef_0 + Coef_1 X + Coef_2 X^2 + \cdots + Coef_{Deg} X^{Deg}$

RFrac

Declaration `function RFrac(X : Float; Coef : TVector; Deg1, Deg2 : Integer) : Float;`

Description Evaluates the rational fraction :

$$F(X) = \frac{Coef_0 + Coef_1 X + Coef_2 X^2 + \cdots + Coef_{Deg} X^{Deg}}{1 + Coef_{Deg+1} X + Coef_{Deg+3} X^2 + \cdots + Coef_{Deg+Deg2} X^{Deg}}$$

4.3 Unit urootpol

4.3.1 Description

Find roots of an arbitrary polynomial. If $Deg \leq 4$, finds analytical solution using units `urtpol1..urtpol4`, otherwise finds them numerically from the companion matrix.

4.3.2 Functions and Procedures

RootPol

Declaration `function RootPol(Coef : TVector; Deg : Integer; Z : TCompVector) : Integer;`

Description Solves the polynomial equation:
$Coef_0 + Coef_1 X + Coef_2 X^2 + \cdots + Coef_{Deg} X^{Deg} = 0$

Returns number of real roots. If an error occurred during the search for the i-th root, the function returns (-i). The roots should be correct for indices (i+1)..Deg. The roots are unordered.

4.4 Unit urtpol1

Linear equation

4.4.1 Functions and Procedures

RootPol1

Declaration `function RootPol1(A, B : Float; var X : Float) : Integer;`

Description Solves the linear equation $A + BX = 0$. Returns 1 if no error ($B \neq 0$); -1 if X is undetermined ($A = B = 0$); -2 if no solution ($A \neq 0,\ B = 0$).

4.5 Unit urtpol2

4.5.1 Description

Roots of Quadratic equation.

4.5.2 Functions and Procedures

RootPol2

Declaration `function RootPol2(Coef : TVector; Z : TCompVector) : Integer;`

Description Solves the quadratic equation:
$Coef_0 + Coef_1 * X + Coef_2 X^2 = 0$

4.6 Unit urtpol3

4.6.1 Description

Cubic equation

4.6.2 Functions and Procedures

RootPol3

Declaration `function RootPol3(Coef : TVector; Z : TCompVector) : Integer;`

Description Solves the cubic equation:
$Coef_0 + Coef_1 X + Coef_2 X^2 + Coef_3 X^3 = 0$

4.7 Unit urtpol4

4.7.1 Description

Roots of a quartic equation

4.7.2 Functions and Procedures

RootPol4

Declaration `function RootPol4(Coef : TVector; Z : TCompVector) : Integer;`

Description Solves the quartic equation:
$Coef_0 + Coef_1 X + Coef_2 X^2 + Coef_3 X^3 + Coef_4 X^4 = 0$

4.8 Unit ucrtptpol

LMath

4.8.1 Description

This unit defines routines to find a derivative of a polynomial and its critical points. Introduced in LMath.

4.8.2 Functions and Procedures

DerivPolynom

Declaration `procedure DerivPolynom(Coef:TVector; Deg:integer; DCoef:TVector; out DDeg:integer);`

Description Finds derivative of a polynomial, which is polynomial of lesser degree. Input parameters: Coef: coefficients of polynomial; Deg: degree of polynomial. Output: DCoef: coefficients of derivative polynomial; DDeg: degree of derivative polynom (Deg - 1).

CriticalPoints

Declaration `function CriticalPoints(Coef:TVector; Deg:integer; var CrtPoints: TRealPointVector; var PointTypes: TIntVector; ResLb : integer = 1):integer;`

Description Finds extrema of polynomial. Input: Coef: coefficients of polynomial; Deg: degree of polynomial. Output: CRTPoints: Critical points; $CRTPoints_i.X$ is abscissa, $CRTPoints_i.Y$ is function value at each of $CrtPoints_i$; Types: type of critical points: -1: it is minimum, 0: no extremum; +1: maximum. Indexing begins from `ResLb`, default value is 1. Returns number of critical points. If CrtPoints and PointTypes are too short, their length is adjusted by the function.

4.9 Unit upolutil

4.9.1 Description

Utility functions to handle roots of polynomials

4.9.2 Functions and Procedures

SetRealRoots

Declaration `function SetRealRoots(Deg : Integer; Z : TCompVector; Tol : Float) : Integer;`

Description Set the imaginary part of a root to zero if it is less than a fraction Tol of its real part. This root is therefore considered real. The function returns the total number of real roots.

SortRoots

Declaration `procedure SortRoots(Deg : Integer; Z : TCompVector);`

Description Sort roots so that:

(1) The Nr real roots are stored in elements [1..Nr] of vector Z, in increasing order.

(2) The complex roots are stored in elements [(Nr + 1)..Deg] of vector Z and are unordered.

Chapter 5

Package lmIntegrals: Numeric Integrating and Solving Differential Equations

5.1 Unit ugausleg

5.1.1 Description

Gauss-Legendre integration

5.1.2 Functions and Procedures

GausLeg

Declaration `function GausLeg(Func : TFunc; A, B : Float) : Float;`

Description Computes the integral of function Func from A to B by the Gauss-Legendre method

GausLeg0

Declaration `function GausLeg0(Func : TFunc; B : Float) : Float;`

Description Computes the integral of function Func from 0 to B by the Gauss-Legendre method

Convol

Declaration `function Convol(Func1, Func2 : TFunc; T : Float) : Float;`

Description Computes the convolution product of two functions Func1 and Func2 at time T by the Gauss-Legendre method.

5.2 Unit urkf

5.2.1 Description

Numerical integration of a system of differential equations by the Runge-Kutta-Fehlberg (RKF) method.
Adapted from a Fortran-90 program available at:
http://www.csit.fsu.edu/ burkardt/f_src/rkf45/rkf45.f90

5.2.2 Functions and Procedures

RKF45

Declaration `procedure RKF45(F : TDiffEqs; Neqn : Integer; Y, Yp : TVector; var T : Float; Tout, RelErr, AbsErr : Float; var Flag : Integer);`

Description RKF45 carries out the Runge-Kutta-Fehlberg method.

This routine is primarily designed to solve non-stiff and mildly stiff differential equations when derivative evaluations are inexpensive. It should generally not be used when the user is demanding high accuracy.

This routine integrates a system of Neqn first-order ordinary differential equations of the form:

$$\frac{dY_i}{dT} = \mathrm{F}(T, Y_1, Y_2, ..., Y_N eqn)$$

where the $Y_1..Y_{Neqn}$ are given at T.

Typically the subroutine is used to integrate from T to $Tout$ but it can be used as a one-step integrator to advance the solution a single step in the direction of $Tout$. On return, the parameters in the call list are set for continuing the integration. The user has only to call again (and perhaps define a new value for $Tout$).

Before the first call, the user must

- supply the F in form: `procedure(X : Float; Y, Yp : TVector)` to evaluate the right hand side;
- initialize the parameters: `Neqn, Y[1:Neqn], T, Tout, RelErr, AbsErr, Flag`. In particular, `T` should initially be the starting point for integration, `Y` should be the value of the initial conditions, and `Flag` should normally be +1.

Normally, the user only sets the value of `Flag` before the first call, and thereafter, the program manages the value. On the first call, `Flag` should normally be +1 (or -1 for single step mode.) On normal return, `Flag` will have been reset by the program to the value of 2 (or -2 in single step mode), and the user can continue to call the routine with that value of `Flag`.

(When the input magnitude of `Flag` is 1, this indicates to the program that it is necessary to do some initialization work. An input magnitude of 2 lets the program know that that initialization can be skipped, and that useful information was computed earlier.)

The routine returns with all the information needed to continue the integration. If the integration reached `Tout`, the user need only define a new `Tout` and call again. In the one-step integrator mode, returning with `Flag` = -2, the user must keep in mind that each step taken is in the direction of the current TOUT. Upon reaching `Tout`, indicated by the output value of FLAG switching to 2, the user must define a new `Tout` and reset `Flag` to -2 to continue in the one-step integrator mode.

In some cases, an error or difficulty occurs during a call. In that case, the output value of `Flag` is used to indicate that there is a problem that the user must address. These values include:

- 3, integration was not completed because the input value of `RelErr`, the relative error tolerance, was too small. `RelErr` has been increased appropriately for continuing. If the user accepts the output value of `RelErr`, then simply reset `Flag` to 2 and continue.
- 4, integration was not completed because more than `MAXNFE` (3000) derivative evaluations were needed. This is approximately `(MAXNFE/6)` steps. The user may continue by simply calling again. The function

counter will be reset to 0, and another `MAXNFE` function evaluations are allowed.

- 5, integration was not completed because the solution vanished, making a pure relative error test impossible. The user must use a non-zero `AbsErr` to continue. Using the one-step integration mode for one step is a good way to proceed.
- 6, integration was not completed because the requested accuracy could not be achieved, even using the smallest allowable stepsize. The user must increase the error tolerances `AbsErr` or `RelErr` before continuing. It is also necessary to reset `Flag` to 2 (or -2 when the one-step integration mode is being used). The occurrence of `Flag` = 6 indicates a trouble spot. The solution is changing rapidly, or a singularity may be present. It often is inadvisable to continue.
- 7, it is likely that this routine is inefficient for solving this problem. Too much output is restricting the natural stepsize choice. The user should use the one-step integration mode with the stepsize determined by the code. If the user insists upon continuing the integration, reset `Flag` to 2 before calling again. Otherwise, execution will be terminated.
- 8, invalid input parameters, indicates one of the following: $Neqn \leq 0$;
  T = Tout and $|Flag| \neq 1$;
  $RelErr < 0$ or $AbsErr < 0$;
  Flag = 0 or Flag not in [-2..8].

Modified:

27 March 2004

Author:

H A Watts and L F Shampine, Sandia Laboratories, Albuquerque, New Mexico.

Reference:

E. Fehlberg, Low-order Classical Runge-Kutta Formulas with Stepsize Control, NASA Technical Report R-315.

L F Shampine, H A Watts, S Davenport, Solving Non-stiff Ordinary Differential Equations - The State of the Art, SIAM Review, Volume 18, pages 376-411, 1976.

Parameters:

- Input, F, a user-supplied function to evaluate the derivatives Y(T), of the form: `procedure(X : Float; Y, Yp : TVector);`
- Input, `Neqn`, the number of equations to be integrated;
- Input/output, `Y[1..Neqn]`, the current solution vector at T;
- Output, `YP[1..Neqn]`, the current value of the derivative of the dependent variable. The user should not set or alter this information;
- Input/output, T, the current value of the independent variable;

- Input, `Tout`, the output point at which solution is desired. `Tout` = T is allowed on the first call only, in which case the routine returns with Flag = 2 if continuation is possible.
- Input/output, `RelErr, AbsErr,` the relative and absolute error tolerances for the local error test. At each step the code requires:

  $$abs(localerror) \leq RelErr * abs(Y) + AbsErr$$

  for each component of the local error and the solution vector Y. `RelErr` cannot be "too small". If the routine believes `RelErr` has been set too small, it will reset `RelErr` to an acceptable value and return immediately for user action.
- Input/output, `Flag`, indicator for status of integration. On the first call, set `Flag` to +1 for normal use, or to -1 for single step mode. On return, a value of 2 or -2 indicates normal progress, while any other value indicates a problem that should be addressed.

5.3 Unit utrapint

5.3.1 Description

Trapezoidal integration

5.3.2 Functions and Procedures

TrapInt

Declaration `function TrapInt(X, Y : TVector; N : Integer) : Float;`

Description Integration by trapezoidal rule, from (X[0], Y[0]) to (X[N], Y[N])

ConvTrap

Declaration `procedure ConvTrap(Func1, Func2 : TFunc; T, Y : TVector; N : Integer);`

Description Computes the convolution product of 2 functions Func1 and Func2 by the trapezoidal rule over an array T[0..N] of equally spaced abscissas, with T[0] = 0. The result is returned in Y[0..N]

Chapter 6

Package lmRandoms: Random Numbers From Different Intervals and Distrubutions

6.1 Unit uranmt

6.1.1 Description

Mersenne Twister Random Number Generator

A C-program for MT19937, with initialization improved 2002/1/26. Coded by Takuji Nishimura and Makoto Matsumoto.

Adapted for DMath by Jean Debord - Feb. 2007

Before using, initialize the state by using init_genrand(seed) or init_by_array(init_key, key_length) (respectively InitMT and InitMTbyArray in the TPMath version).

Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

6.1.2 Functions and Procedures

InitMT

Declaration `procedure InitMT(Seed : Integer);`

Description Initializes MT generator with a seed

InitMTbyArray

Declaration `procedure InitMTbyArray(InitKey : MTKeyArray; KeyLength : Word);`

Description Initialize MT generator with an array InitKey[0..(KeyLength - 1)]

IRanMT

Declaration `function IRanMT : Integer;`

Description Generates a Random number on [-2ˆ31 .. 2ˆ31 - 1] interval

6.1.3 Types

MTKeyArray

Declaration `MTKeyArray = array[0..623] of Cardinal;`

6.2 Unit uranmwc

6.2.1 Description

Marsaglia's Multiply-With-Carry random number generator

6.2.2 Functions and Procedures

InitMWC

Declaration `procedure InitMWC(Seed : Integer);`

Description Initializes the 'Multiply with carry' random number generator.

IRanMWC

Declaration `function IRanMWC : Integer;`

Description Returns a 32 bit random number in [-2^31 ; 2^31-1]

6.3 Unit uranuvag

6.3.1 Description

UVAG The Universal Virtual Array Generator by Alex Hay zenjew@hotmail.com Adapted to DMath by Jean Debord

In practice, Cardinal (6-7 times the output of Word) is the IntType of choice, but to demonstrate UVAG's scalability here, IntType can be defined as any integer data type. IRanUVAG globally provides (as rndint) an effectively infinite sequence of IntTypes, uniformly distributed (0, 2^(8*sizeof(IntType))-1). Output (bps) is dependent solely on IntSize=sizeof(IntType) and CPU speed. UVAG cycles at twice the speed of the 64-bit Mersenne Twister in a tenth the memory, tests well in DIEHARD, ENT and NIST and has a huge period. It is suitable for cryptographic purposes in that state(n) is not determinable from state(n+1). Most attractive is that it uses integers of any size and requires an array of only 255 + sizeof(IntType) bytes. Thus it is easily adapted to 128 bits and beyond with negligible memory increase. Lastly, seeding is easy. From near zero entropy (s[]=0, rndint > 0), UVAG bootstraps itself to full entropy in under 300 cycles. Very robust, no bad seeds.

6.3.2 Functions and Procedures

InitUVAGbyString

Declaration `procedure InitUVAGbyString(KeyPhrase : string);`

Description Initializes the generator with a string

InitUVAG

Declaration `procedure InitUVAG(Seed : Integer);`

Description Initializes the generator with an integer

IRanUVAG

Declaration `function IRanUVAG : Integer;`

Description Returns a 32-bit random integer

6.4 Unit urandom

6.4.1 Description

Random number generators

6.4.2 Functions and Procedures

SetRNG

Declaration `procedure SetRNG(RNG : RNG_Type);`

Description Select generator and set default initialization: RNG_MWC = Multiply-With-Carry; RNG_MT = Mersenne Twister; RNG_UVAG = Universal Virtual Array Generator.

InitGen

Declaration `procedure InitGen(Seed : Integer);`

Description Initialize generator.

IRanGen

Declaration `function IRanGen : Integer;`

Description 32-bit random integer in $[-2^{31}..2^{31}-1]$.

IRanGen31

Declaration `function IRanGen31 : Integer;`

Description 31-bit random integer in $[0..2^{31}-1]$.

RanGen1

Declaration `function RanGen1 : Float;`

Description 32-bit random real in [0,1].

RanGen2

Declaration `function RanGen2 : Float;`

Description 32-bit random real in [0,1).

RanGen3

Declaration `function RanGen3 : Float;`

Description 32-bit random real in (0,1).

RanGen53

Declaration `function RanGen53 : Float;`

Description 53-bit random real in [0,1).

6.5 Unit urangaus

6.5.1 Description

Gaussian random numbers

6.5.2 Functions and Procedures

RanGaussStd

Declaration `function RanGaussStd : Float;`

Description Computes 2 random numbers from the standard normal distribution, returns one and saves the other for the next call.

RanGauss

Declaration `function RanGauss(Mu, Sigma : Float) : Float;`

Description Returns a random number from a Gaussian distribution with mean Mu and standard deviation Sigma.

6.6 Unit uranmult

6.6.1 Description

Random number from a multinormal distribution.

6.6.2 Functions and Procedures

RanMult

Declaration `procedure RanMult(M : TVector; L : TMatrix; Lb, Ub : Integer; X : TVector);`

Description Generates a random vector X from a multinormal distribution. M is the mean vector, L is the Cholesky factor (lower triangular) of the variance-covariance matrix.

RanMultIndep

Declaration `procedure RanMultIndep(M, S : TVector; Lb, Ub : Integer; X : TVector);`

Description Generates a random vector X from a multinormal distribution with uncorrelated variables. M is the mean vector, S is the vector of standard deviations.

Chapter 7

Package lmMathStat: Distributions and Hypothesis Testing

7.1 Unit umeansd

7.1.1 Description

Various statistics of a vector: min, max, mean, standard deviation, sum. Every function exists in two forms: older, with TVector, Lb, Ub convention and newer, with open arrays. Older functions are left for backward compatibility and are not recommended for use in a new code.

7.1.2 Functions and Procedures

Min

Declaration
```
function Min(X : TVector; Lb, Ub : Integer) : Float;
function Min(constref X:array of float) : float;
```

Description Minimum of sample X

Max

Declaration
```
function Max(X : TVector; Lb, Ub : Integer) : Float;
Max(constref X : array of float) : Float;
```

Description Maximum of sample X

Sum

LMath

Declaration
```
function Sum(X:TVector; Lb, Ub : integer) : Float;
Sum(constref X:array of float) : Float;
```

Description Sum of sample X.

Mean

Declaration
```
function Mean(X : TVector; Lb, Ub : Integer) : Float;
Mean(constref X:array of float) : Float;
```

Description Mean of sample X

StDev

Declaration
```
function StDev(X:TVector; Lb,Ub:Integer; M:Float) : Float;
StDev(constref X : array of float; M : Float) : Float;
```

Description Standard deviation estimated from sample X

StDevP

Declaration
```
function StDevP(X:TVector; Lb, Ub:Integer; M:Float) : Float;
StDevP(constref X : array of float; M : Float) : Float;
```

Description Standard deviation of population

7.2 Unit umeansd_md

LMath

7.2.1 Description

Mean and standard deviations, aware of missing data. Completely written for LMath.

7.2.2 Functions and Procedures

Defined

Declaration `function Defined(F:Float):boolean;`
`function Defined(A: array of Float; N : integer):boolean;`

Description returns true if F (or A[N]) is not NAN or Missing data. If $N > High(A)$ returns `False`.

Undefined

Declaration `function Undefined(F:Float):boolean;`
`function Undefined(A: array of Float; N : integer):boolean;`

Description returns true if F (or A[N]) is NAN or Missing data and if $N > High(A)$.

RemoveUndefined

Declaration `function RemoveUndefined(X : TVector; Lb:integer = 0; Ub: integer = -1):TVector;`

Description Returns an array which contains onla defined elements from the source array.

SetMD

Declaration `procedure SetMD(aMD:float);`

Description set missing data code

FirstDefined

Declaration `function FirstDefined(X:TVector; Lb:integer = 0; Ub:Integer = -1):integer;`

Description Finds first defined element in array. Returns -1 if nothing found. If $Ub = -1$, array is searched to the end. Hence, Lb and Ub parameters may be omitted for a search of the whole array.

LastDefined

Declaration `function LastDefined(X:TVector; Lb:integer = 0; Ub:Integer = -1):integer;`

Description Finds last defined element in array. Returns -1 if nothing found. If $Ub = -1$, array is searched from the end. Hence, Lb and Ub parameters may be omitted for a search of the whole array.

ValidN

Declaration `function ValidN(X:TVector; Lb, Ub:Integer):integer;`
`function ValidN(constref X:array of Float):integer;`

Description Number of valid (defined) elements in array

Min

Declaration `function Min(X : TVector; Lb, Ub : Integer) : Float;`
`function Min(constref X:array of float) : float;`

Description Minimum of sample X

Max

Declaration `function Max(X : TVector; Lb, Ub : Integer) : Float;`
`function Max(constref X:array of float) : float;`

Description Maximum of sample X

Mean

Declaration `function Mean(X : TVector; Lb, Ub : Integer) : Float;`
`function Mean(constref X : array of float) : Float;`

Description Mean of sample X

StDev

Declaration `function StDev(X : TVector; Lb, Ub : Integer) : Float;`
`function StDev(constref X : array of float) : Float;`

Description Standard deviation estimated from sample X

StDevP

Declaration `function StDevP(X : TVector; Lb, Ub : Integer) : Float;`
`function StDevP(constref X : array of float) : Float;`

Description Standard deviation of population

StdErr

Declaration `function StdErr(X : TVector; Lb, Ub : Integer) : Float;`
`function StdErr(constref X : array of float) : Float;`

Description Standard error of mean

7.2.3 Variables

MissingData

Declaration `MissingData: Float = NAN;`

7.3 Unit umedian

7.3.1 Description

Median

7.3.2 Functions and Procedures

Median

Declaration `function Median(X : TVector; Lb, Ub : Integer) : Float;`
`function Median(X:array of float) : float;`

Description Returns median for vector X. Importantly, vector `X` is rearranged by the algorithm. If you need original vector, use the second form of the function, where vector is passed by value.

7.4 Unit udistrib

7.4.1 Description

Statistical distribution

7.4.2 Functions and Procedures

DimStatClassVector

Declaration `function DimStatClassVector(out C : TStatClassVector; A, B, H : float):integer;`

Description Allocates an array of statistical classes (histogram bins, see 2.2.2.8) for description of `StatClass`. `A` is lower border of histogram; `B` is upper border; `H` is bin width. Function calculates number of bins and allocates them. Number of bins is returned. If allocation is impossible, nil is returned.

Distrib

Declaration `procedure Distrib(X : TVector; Lb, Ub : Integer; A, H : Float; C : TStatClassVector);`

Description Distributes the values of array X[Lb..Ub] into M classes with equal width H, according to the following scheme:

```
C[1]    C[2]                        C[M]
]-------]-------].......]-------]-------]
A       A+H     A+2H                    B
```

such that $B = A + MH$. Number of classes `M` is High(C); note that it is 1-based.

distExtract*

LMath 0.5

Declaration procedure distExtractX(C : TStatClassVector; out Xv : TVector);
procedure distExtractN(C : TStatClassVector; out Nv : TIntVector);
procedure distExtractFreq(C : TStatClassVector; out Fv : TVector);
procedure distExtractDensity(C : TStatClassVector; out Dv : TVector);

Description `TStatClassVector` is a vector of `StatClass` records, which contain various information about a bin of statistical distribution. `DistExtract*` family procedures extract particular information from these records and place it into `TVector` or `TIntVector`. `distExtractX` returns middle of each bin in the distribution; `distExtractN`: count of values falling into the bin; `distExtractFreq`: frequency value for this bin; `distExtractDensity`: a probability density value.

7.5 Unit uskew

7.5.1 Description

Skewness and kurtosis

7.5.2 Functions and Procedures

Skewness

Declaration `function Skewness(X : TVector; Lb, Ub : Integer; M, Sigma : Float) : Float;`

Kurtosis

Declaration `function Kurtosis(X : TVector; Lb, Ub : Integer; M, Sigma : Float) : Float;`

7.6 Unit ubinom

7.6.1 Description

Binomial coefficient and probability of distribution. Function for cumulative probability `FBinom` is located in the unit `uIBtDist` (7.13).

7.6.2 Functions and Procedures

Binomial

Declaration `function Binomial(N, K : Integer) : Float;`

Description Binomial coefficient $\binom{N}{K}$

PBinom

Declaration `function PBinom(N : Integer; P : Float; K : Integer) : Float;`

Description Probability of binomial distribution for `K` successes in `N` attempts and probability of success `P`.

7.7 Unit upoidist

7.7.1 Description

Poisson distribution

7.7.2 Functions and Procedures

PPoisson

Declaration `function PPoisson(Mu : Float; K : Integer) : Float;`

Description Probability of Poisson distribution. Probability to observe K if mean is μ.

7.8 Unit uexpdist

7.8.1 Description

Exponential distribution

7.8.2 Functions and Procedures

DExpo

Declaration `function DExpo(A, X : Float) : Float;`

Description Density of exponential distribution with parameter A.

FExpo

Declaration `function FExpo(A, X : Float) : Float;`

Description Cumulative probability function for exponential distribution with parameter A.

7.9 Unit unormal

7.9.1 Description

Density of standard normal distribution.

7.9.2 Functions and Procedures

DNorm

Declaration `function DNorm(X : Float) : Float;`

Description Density of standard normal distribution

DGaussian

LMath

Declaration `function DGaussian(X, Mean, Sigma: float) : float;`

Description Density of gaussian distribution with orbitrary math. expectation Mean and standard deviation Sigma.

7.10 Unit uinvnorm

7.10.1 Description

Inverse of Normal distribution function. Translated from C code in Cephes library (http://www.moshier.net)

7.10.2 Functions and Procedures

InvNorm

Declaration `function InvNorm(P : Float) : Float;`

Description Inverse of Normal distribution function

Returns the argument, X, for which the area under the Gaussian probability density function (integrated from $-\infty$ to X) is equal to P.

7.11 Unit uigmdist

7.11.1 Description

Probability functions related to the incomplete Gamma function

7.11.2 Functions and Procedures

FGamma

Declaration `function FGamma(A, B, X : Float) : Float;`

Description Cumulative probability for Gamma distribution with parameters A and B.

FPoisson

Declaration `function FPoisson(Mu : Float; K : Integer) : Float;`

Description Cumulative probability for Poisson distribution.

FNorm

Declaration `function FNorm(X : Float) : Float;`

Description Cumulative probability for standard normal distribution.

PNorm

Declaration `function PNorm(X : Float) : Float;`

Description Prob($|U| > X$) for standard normal distribution.

FKhi2

Declaration `function FKhi2(Nu : Integer; X : Float) : Float;`

Description Cumulative prob. for χ^2 distrib. with Nu d.o.f.

PKhi2

Declaration `function PKhi2(Nu : Integer; X : Float) : Float;`

Description Prob(Khi2 > X) for χ^2 distribution with Nu d.o.f.

7.12 Unit ugamdist

7.12.1 Description

Probability functions related to the Gamma function

7.12.2 Functions and Procedures

DBeta

Declaration `function DBeta(A, B, X : Float) : Float;`

Description Density of Beta distribution with parameters A and B.

DGamma

Declaration `function DGamma(A, B, X : Float) : Float;`

Description Density of Gamma distribution with parameters A and B.

DKhi2

Declaration `function DKhi2(Nu : Integer; X : Float) : Float;`

Description Density of χ^2 distribution with Nu d.o.f.

DStudent

Declaration `function DStudent(Nu : Integer; X : Float) : Float;`

Description Density of Student distribution with Nu d.o.f.

DSnedecor

Declaration `function DSnedecor(Nu1, Nu2 : Integer; X : Float) : Float;`

Description Density of Fisher-Snedecor distribution with Nu1 and Nu2 d.o.f.

7.13 Unit uibtdist

7.13.1 Description

Probability functions related to the incomplete Beta function.

7.13.2 Functions and Procedures

FBeta

Declaration `function FBeta(A, B, X : Float) : Float;`

Description Cumulative probability for Beta distrib. with param. A and B

FBinom

Declaration `function FBinom(N : Integer; P : Float; K : Integer) : Float;`

Description Cumulative probability for binomial distrib.

FStudent

Declaration `function FStudent(Nu : Integer; X : Float) : Float;`

Description Cumulative probability for Student distrib. with Nu d.o.f.

PStudent

Declaration `function PStudent(Nu : Integer; X : Float) : Float;`

Description Prob(—t— > X) for Student distrib. with Nu d.o.f.

FSnedecor

Declaration `function FSnedecor(Nu1, Nu2 : Integer; X : Float) : Float;`

Description Cumulative prob. for Fisher-Snedecor distrib. with Nu1 and Nu2 d.o.f.

PSnedecor

Declaration `function PSnedecor(Nu1, Nu2 : Integer; X : Float) : Float;`

Description Prob(F > X) for Fisher-Snedecor distrib. with Nu1 and Nu2 d.o.f.

7.14 Unit uinvbeta

7.14.1 Description

Inverses of incomplete Beta function, Student and F-distributions.
Translated from C code in Cephes library (http://www.moshier.net)

7.14.2 Functions and Procedures

InvBeta

Declaration `function InvBeta(A, B, Y : Float) : Float;`

Description Inverse of incomplete Beta function. Given P, the function finds X such that IBeta(A, B, X) = Y

InvStudent

Declaration `function InvStudent(Nu : Integer; P : Float) : Float;`

Description Inverse of Student's t-distribution function Given probability P, finds the argument X such that FStudent(Nu, X) = P

InvSnedecor

Declaration `function InvSnedecor(Nu1, Nu2 : Integer; P : Float) : Float;`

Description Inverse of Snedecor's F-distribution function Given probability P, finds the argument X such that FSnedecor(Nu1, Nu2, X) = P

7.15 Unit uinvgam

7.15.1 Description

Inverses of incomplete Gamma function and χ^2 distribution.
Translated from C code in Cephes library (http://www.moshier.net)

7.15.2 Functions and Procedures

InvGamma

Declaration `function InvGamma(A, P : Float) : Float;`

Description Given P, the function finds X such that IGamma(A, X) = P It is best valid in the right-hand tail of the distribution, P > 0.5

InvKhi2

Declaration `function InvKhi2(Nu : Integer; P : Float) : Float;`

Description Inverse of Khi-2 distribution function

Returns the argument, X, for which the area under the Khi-2 probability density function (integrated from 0 to X) is equal to P.

7.16 Unit ustudind

7.16.1 Description

Student t-test for independent samples.

7.16.2 Overview

`StudIndep`

7.16.3 Functions and Procedures

StudIndep

Declaration `procedure StudIndep(N1, N2 : Integer; M1, M2, S1, S2 : Float; var T : Float; var DoF : Integer);`

Description Student t-test for independent samples.

Input parameters: N1, N2 = samples sizes; M1, M2 = samples means; S1, S2 = samples SD's (computed with StDev); Output parameters: T = Student's t; DoF = degrees of freedom.

7.17 Unit ustdpair

7.17.1 Description

Student t-test for paired samples.

7.17.2 Overview

`StudPaired`

7.17.3 Functions and Procedures

StudPaired

Declaration `procedure StudPaired(X, Y : TVector; Lb, Ub : Integer; var T : Float; var DoF : Integer);`

Description Student t-test for paired samples.

Input parameters: X, Y = samples; Lb, Ub = lower and upper bounds. Output parameters: T = Student's t; DoF = degrees of freedom.

7.18 Unit uanova1

7.18.1 Description

One-way analysis of variance.

7.18.2 Functions and Procedures

AnOVa1

Declaration `procedure AnOVa1(Ns : Integer; N : TIntVector; M, S : TVector; var V_f, V_r, F : Float; var DoF_f, DoF_r : Integer);`

Description Input parameters: Ns = number of samples; N = samples sizes; M = samples means; S = samples SD's (computed with StDev). Output parameters: V_f, V_r = variances (factorial, residual)L F = ratio Vf / Vr; DoF_f, DoF_r = degrees of freedom.

7.19 Unit uanova2

7.19.1 Description

Two-way analysis of variance

7.19.2 Functions and Procedures

AnOVa2

Declaration `procedure AnOVa2(NA, NB, Nobs : Integer; M, S : TMatrix; V, F : TVector; DoF : TIntVector);`

Description Input parameters : NA = number of modalities for factor A; NB = number of modalities for factor B; Nobs = number of observations for each sample; M = matrix of means (factor A as lines, factor B as columns); S = matrix of standard deviations. Output parameters: V = variances (factor A, factor B, interaction, residual); F = variance ratios (factor A, factor B, interaction); DoF = degrees of freedom (factor A, factor B, interaction, residual).

7.20 Unit ubartlet

7.20.1 Description

Bartlett's test (comparison of several variances)

7.20.2 Overview

`Bartlett`

7.20.3 Functions and Procedures

Bartlett

Declaration `procedure Bartlett(Ns : Integer; N : TIntVector; S : TVector; var Khi2 : Float; var DoF : Integer);`

Description Input parameters: Ns = number of samples; N = samples sizes; S = samples SD's (computed with StDev). Output parameters: Khi2 = Bartlett's χ^2; DoF = degrees of freedom.

7.21 Unit ukhi2

7.21.1 Description

χ^2 test

7.21.2 Functions and Procedures

Khi2_Conform

Declaration `procedure Khi2_Conform(N_cls : Integer; N_estim : Integer; Obs : TIntVector; Calc : TVector; out Khi2 : Float; out DoF : Integer);`

Description χ^2 test for conformity. N_cls is the number of classes; N_estim the number of estimated parameters; Obs[1..N_cls] and Calc[1..N_cls] the observed and theoretical distributions. The statistic is returned in Khi2 and the number of d. o. f. in DoF.

Khi2_Indep

Declaration `procedure Khi2_Indep(N_lin : Integer; N_col : Integer; Obs : TIntMatrix; out Khi2 : Float; out DoF : Integer);`

Description Khi-2 test for independence N_lin and N_col are the numbers of lines and columns Obs[1..N lin, 1..N col] is the matrix of observed distributions. The statistic is returned in G and the number of d. o. f. in DoF.

7.22 Unit usnedeco

7.22.1 Description

Snedecor's F-test (comparison of two variances).

7.22.2 Functions and Procedures

Snedecor

Declaration `procedure Snedecor(N1, N2 : Integer; S1, S2 : Float; var F : Float; var DoF1, DoF2 : Integer);`

Description Snedecor's F-test (comparison of two variances).

Input parameters: N1, N2 = samples sizes; S1, S2 = samples SD's (computed with StDev). Output parameters: F = Snedecor's F; DoF1, DoF2 = degrees of freedom.

7.23 Unit uwoolf

7.23.1 Description

Woolf test

7.23.2 Functions and Procedures

Woolf_Conform

Declaration `procedure Woolf_Conform(N_cls : Integer; N_estim : Integer; Obs : TIntVector; Calc : TVector; out G : Float; out DoF : Integer);`

Description Woolf test for conformity. N_cls is the number of classes; N_estim is the number of estimated parameters; Obs[1..N_cls] and Calc[1..N_cls] are the observed and theoretical distributions. The statistic is returned in G and the number of d. o. f. in DoF.

Woolf_Indep

Declaration `procedure Woolf_Indep(N_lin : Integer; N_col : Integer; Obs : TIntMatrix; out G : Float; out DoF : Integer);`

Description Woolf test for independence. N_lin and N_col are the numbers of lines and columns; Obs[1..N_lin, 1..N_col] is the matrix of observed distributions. The statistic is returned in G and the number of d. o. f. in DoF.

7.24 Unit unonpar

7.24.1 Description

Non-parametric tests

7.24.2 Functions and Procedures

Mann_Whitney

Declaration `procedure Mann_Whitney(N1, N2 : Integer; X1, X2 : TVector; out U, Eps : Float);`

Description Mann-Whitney test N1 and N2 are the sample sizes X1[1..N1] and X2[1..N2] are the two samples. The procedure returns Mann-Whitney's statistic in U and the associated normal variable in Eps.

Wilcoxon

Declaration `procedure Wilcoxon(X, Y : TVector; Lb, Ub : Integer; out Ndiff : Integer; out T, Eps : Float);`

Description Wilcoxon test X[Lb..Ub] and Y[Lb..Ub] are the two samples. Output: the number of non-zero differences in Ndiff, Wilcoxonś statistic in T and the associated normal variable in Eps.

Kruskal_Wallis

Declaration `procedure Kruskal_Wallis(Ns : Integer; N : TIntVector; X : TMatrix; out H : Float; out DoF : Integer);`

Description Kruskal-Wallis test Ns is the number of samples, N[1..Ns] is the vector of sizes and X the sample matrix (with the samples as columns). Output: Kruskal-Wallis statistic in H and the number of d. o. f. in DoF.

7.25 Unit upca

7.25.1 Description

Principal component analysis

7.25.2 Functions and Procedures

VecMean

Declaration `procedure VecMean(X : TMatrix; Lb, Ub, Nvar : Integer; M : TVector);`

Description Computes the mean vector (M) from matrix X.

Input : X[Lb..Ub, 1..Nvar] = matrix of variables. Output : M[1..Nvar] = mean vector.

VecSD

Declaration `procedure VecSD(X : TMatrix; Lb, Ub, Nvar : Integer; M, S : TVector);`

Description Computes the vector of standard deviations (S) from matrix X.

Input : X, Lb, Ub, Nvar, M. Output : S[1..Nvar].

MatVarCov

Declaration `procedure MatVarCov(X : TMatrix; Lb, Ub, Nvar : Integer; M : TVector; V : TMatrix);`

Description Computes the variance-covariance matrix (V) from matrix X.

Input : X, Lb, Ub, Nvar, M Output : V[1..Nvar, 1..Nvar]

MatCorrel

Declaration `procedure MatCorrel(V : TMatrix; Nvar : Integer; R : TMatrix);`

Description Computes the correlation matrix (R) from the variance-covariance matrix (V).

Input : V, Nvar Output : R[1..Nvar, 1..Nvar]

PCA

Declaration `procedure PCA(R : TMatrix; Nvar : Integer; Lambda : TVector; C, Rc : TMatrix);`

Description Performs a principal component analysis of the correlation matrix R.

Input: R[1..Nvar, 1..Nvar] = Correlation matrix.

Output: Lambda[1..Nvar] = Eigenvalues of the correlation matrix (in descending order); C[1..Nvar, 1..Nvar] = Eigenvectors of the correlation matrix (stored as columns); Rc[1..Nvar, 1..Nvar] = Correlations between principal factors and variables (Rc[I,J] is the correlation coefficient between variable I and factor J). NB : This procedure destroys the original matrix R.

ScaleVar

Declaration `procedure ScaleVar(X : TMatrix; Lb, Ub, Nvar : Integer; M, S : TVector; Z : TMatrix);`

Description Scales a set of variables by subtracting means and dividing by SD's.

Input : X, Lb, Ub, Nvar, M, S Output : Z[Lb..Ub, 1..Nvar] = matrix of scaled variables.

PrinFac

Declaration `procedure PrinFac(Z : TMatrix; Lb, Ub, Nvar : Integer; C, F : TMatrix);`

Description Computes principal factors. Input: Z[Lb..Ub, 1..Nvar] = matrix of scaled variables; C[1..Nvar, 1..Nvar] = matrix of eigenvectors from PCA. Output : F[Lb..Ub, 1..Nvar] = matrix of principal factors.

7.26 Unit ucorrel

7.26.1 Description

Correlation coefficient

7.26.2 Functions and Procedures

Correl

Declaration `function Correl(X, Y : TVector; Lb, Ub : Integer) : Float;`

Description Correlation coefficient between samples X and Y.

Chapter 8

Package lmOptimum: Algorithms of Optimization

8.1 Description

This package contains a collection of algorithm to minimize function of one or several variables.
In the calls to all optimization procedures, function of one variable must be defined as

```
function MyFunc(X:float):float;
```

which corresponds to TFunc type; initial minimum guess value usually must be supplied in X:float parameter.
Function of several variables must be defined as

```
function MyFunc(X:TVector):Float;
```

which corresponds to TFuncNVar type; initial guess for minimum is supplied in X:TVector parameter.

8.2 Unit uminbrak

8.2.1 Description

Brackets a minimum of a function

8.2.2 Functions and Procedures

MinBrack

Declaration `procedure MinBrack(Func : TFunc; var A, B, C: Float; out Fa, Fb, Fc : Float);`

Description Given two points (A, B) this procedure finds a triplet (A, B, C) such that: (i) A < B < C (ii) A, B, C are within the golden ratio (iii) Func(B) < Func(A) and Func(B) < Func(C). The corresponding function values are returned in `Fa`, `Fb`, `Fc`.

SetBrakConstrain

Declaration `procedure SetBrakConstrain(L, R: Float);`

Description Set initial constrain, such that function minimum will be searched only within [L,R] interval.

8.3 Unit ugoldsrc

8.3.1 Description

Minimization of a function of one variable by Golden Search method.

8.3.2 Functions and Procedures

GoldSearch

Declaration `procedure GoldSearch(Func : TFunc; A, B : Float; MaxIter : Integer; Tol : Float; var Xmin, Ymin : Float);`

Description Performs a golden search for the minimum of function Func.

Input parameters: Func = objective function; A, B = two points near the minimum; MaxIter = maximum number of iterations; Tol = required precision (should not be less than the square root of the machine precision).

Output parameters: Xmin, Ymin = coordinates of minimum.

Possible results : OptOk, OptNonConv.

8.4 Unit usimplex

8.4.1 Description

Function minimization by the simplex method

8.4.2 Functions and Procedures

SaveSimplex

Declaration `procedure SaveSimplex(FileName : string);`

Description Opens a file to save the Simplex iterations.

Simplex

Declaration `procedure Simplex(Func : TFuncNVar; X : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; var F_min : Float);`

Description Minimization of a function of several variables by the simplex method of Nelder and Mead.
Input parameters: Func = objective function; X = initial (guess) minimum coordinates; Lbound, Ubound = indices of first and last variables in X vector; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined minimum coordinates; F_min = function value at minimum.

The function MathErr returns one of the following codes:

OptOk = no error; OptNonConv = non-convergence.

8.5 Unit ubfgs

8.5.1 Description

Minimization of a function of several variables by the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method

8.5.2 Functions and Procedures

SaveBFGS

Declaration `procedure SaveBFGS(FileName : string);`

Description Save BFGS iterations in a file.

BFGS

Declaration `procedure BFGS(Func : TFuncNVar; Gradient : TGradient; X : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; var F_min : Float; G : TVector; H_inv : TMatrix);`

Description Minimization of a function of several variables by the Broyden-Fletcher-Goldfarb-Shanno method.

Input parameters: Func:TFuncNVar = objective function; Gradient:TGradient = procedure to compute gradient; X = initial guess minimum coordinates; Lb, Ub = indices of first and last variables in X; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined minimum coordinates; F_min = function value at minimum; G = gradient vector; H_inv = inverse hessian matrix.

Possible results: OptOk, OptNonConv.

8.6 Unit unewton

8.6.1 Description

Minimization of a function of several variables by the Newton-Raphson method

8.6.2 Overview

`SaveNewton`

`Newton`

8.6.3 Functions and Procedures

SaveNewton

Declaration `procedure SaveNewton(FileName : string);`

Description Save Newton-Raphson iterations in a file

Newton

Declaration `procedure Newton(Func : TFuncNVar; HessGrad : THessGrad; X : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; var F_min : Float; G : TVector; H_inv : TMatrix; var Det : Float);`

Description Minimization of a function of several variables by the Newton-Raphson method.

Input parameters: Func = objective function; HessGrad = procedure to compute hessian and gradient; X = initial guess minimum coordinates; Lb, Ub = indices of first and last variables in X; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined minimum coordinates; F_min = function value at minimum; G = gradient vector; H_inv = inverse hessian matrix; Det = determinant of hessian.

Possible results: OptOk = no error OptNonConv = non-convergence OptSing = singular hessian matrix.

8.7 Unit umarq

8.7.1 Description

Minimization of a function of several variables by Marquardt's method

8.7.2 Overview

`SaveMarquardt`

`Marquardt`

8.7.3 Functions and Procedures

SaveMarquardt

Declaration `procedure SaveMarquardt(FileName : string);`

Description Save Marquardt iterations in a file

Marquardt

Declaration `procedure Marquardt(Func : TFuncNVar; HessGrad : THessGrad; X : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; out F_min : Float; G : TVector; H_inv : TMatrix; out Det : Float);`

Description Minimization of a function of several variables by Marquardt's method.

Input parameters. : Func = objective function; HessGrad = procedure to compute hessian and gradient; X = initial guess minimum coordinates; Lb, Ub = indices of first and last variables in X; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined minimum coordinates; F_min = function value at minimum; G = gradient vector; H_inv = inverse hessian matrix; Det = determinant of hessian.

Possible results: OptOk = no error; OptNonConv = non-convergence; OptSing = singular hessian matrix; OptBigLambda = too high Marquardt parameter Lambda.

8.8 Unit ulinmin

8.8.1 Description

Minimization of a function of several variables along a line.

8.8.2 Overview

`LinMin`

8.8.3 Functions and Procedures

LinMin

Declaration `procedure LinMin(Func : TFuncNVar; X, DeltaX : TVector; Lb, Ub : Integer; var R : Float; MaxIter : Integer; Tol : Float; var F_min : Float);`

Description Minimizes function Func from point X in the direction specified by DeltaX.

Input parameters: Func:TFuncNVar = objective function; X = initial minimum coordinates; DeltaX = direction in which minimum is searched; Lb, Ub = indices of first and last variables; R = initial step, in fraction of $|DeltaX|$; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined minimum coordinates; R = step corresponding to the minimum; F_min = function value at minimum.

Possible results: OptOk, OptNonConv.

8.9 Unit ugenalg

8.9.1 Description

Optimization by Genetic Algorithm
Ref.: E. Perrin, A. Mandrille, M. Oumoun, C. Fonteix & I. Marc Optimisation globale par strategie d'evolution Technique utilisant la genetique des individus diploides Recherche operationnelle / Operations Research 1997, 31, 161-201.
Thanks to Magali Camut for her contribution.

8.9.2 Functions and Procedures

InitGAParams

Declaration `procedure InitGAParams(NP, NG : Integer; SR, MR, HR : Float);`

Description Initialize Genetic Algorithm parameters.

NP: Population size; NG: Max number of generations; SR: Survival rate; MR: Mutation rate; HR: Proportion of homozygotes.

GA_CreateLogFile

Declaration `procedure GA_CreateLogFile(LogFileName : String);`

Description Initialize log file.

GenAlg

Declaration `procedure GenAlg(Func : TFuncNVar; X, Xmin, Xmax : TVector; Lb, Ub : Integer; var F_min : Float);`

Description Minimization of a function of several variables by genetic algorithm.

Input parameters: Func = objective function to be minimized; X = initial minimum coordinates; Xmin = minimum value of X; Xmax = maximum value of X; Lb, Ub = array bounds.

Output parameters: X = refined minimum coordinates; F_min = function value at minimum.

8.10 Unit umcmc

8.10.1 Description

Simulation by Markov Chain Monte Carlo (MCMC) with the Metropolis-Hastings algorithm.

This algorithm simulates the probability density function (pdf) of a vector X. The pdf P(X) is written as:

$$P(X) = C\mathrm{e}^{\frac{-F(X)}{T}}$$

Simulating P by the Metropolis-Hastings algorithm is equivalent to minimizing F by simulated annealing at the constant temperature T. The constant C is not used in the simulation.

The series of random vectors generated during the annealing step constitutes a Markov chain which tends towards the pdf to be simulated.

It is possible to run several cycles of the algorithm. The variance-covariance matrix of the simulated distribution is re-evaluated at the end of each cycle and used for the next cycle.

8.10.2 Functions and Procedures

InitMHParams

Declaration `procedure InitMHParams(NCycles, MaxSim, SavedSim : Integer);`

Description Initializes Metropolis-Hastings parameters.

GetMHParams

Declaration `procedure GetMHParams(out NCycles, MaxSim, SavedSim : Integer);`

Description Returns Metropolis-Hastings parameters.

Hastings

Declaration `procedure Hastings(Func : TFuncNVar; T : Float; X : TVector; V : TMatrix; Lb, Ub : Integer; Xmat : TMatrix; X_min : TVector; var F_min : Float);`

Description Simulation of a probability density function by the Metropolis-Hastings algorithm.

Input parameters: Func = Function such that the pdf is

$$P(X) = C\mathrm{e}^{\frac{-\mathrm{Func}(X)}{T}}$$

T = Temperature; X = Initial mean vector; V = Initial variance-covariance matrix; Lb, Ub = Indices of first and last variables.

Output parameters: Xmat = Matrix of simulated vectors, stored row-wise, i.e. Xmat[1..MH_SavedSim, Lb..Ub]; X = Mean of distribution; V = Variance-covariance matrix of distribution; X_min = Coordinates of minimum of F(X) (mode of the distribution); F_min = Value of F(X) at minimum.

Possible results: MatOk: No error; MatNotPD: The variance-covariance matrix is not positive definite.

8.11 Unit usimann

8.11.1 Description

Optimization by Simulated Annealing
Adapted from Fortran program SIMANN by Bill Goffe:
http://www.netlib.org/opt/simann.f

8.11.2 Functions and Procedures

InitSAParams

Declaration `procedure InitSAParams(NT, NS, NCycles : Integer; RT : Float);`

Description Initialize simulated annealing parameters

NT: Number of loops at constant temperature; NS: Number of loops before step adjustment; NCycles: Number of cycles; RT: Temperature reduction factor.

SA_CreateLogFile

Declaration `procedure SA_CreateLogFile(FileName : String);`

Description Initialize log file

SimAnn

Declaration `procedure SimAnn(Func : TFuncNVar; X, Xmin, Xmax : TVector; Lb, Ub : Integer; var F_min : Float);`

Description Minimization of a function of several variables by simulated annealing.

Input parameters: Func:TFuncNVar = objective function to be minimized; X = initial guess minimum coordinates; Xmin = minimum value of X; Xmax = maximum value of X; Lb, Ub = indices of first and last variables.

Output parameter: X = refined minimum coordinates; F_min = function value at minimum.

8.12 Unit ueval

8.12.1 Description

Simple Expression Evaluator, Version: 1.1. Author : Aleksandar Ruzicic (admin@krcko.net) File: fbeval.bas BIG thanks goes to Jack W. Crenshaw for his "LET'S BUILD A COMPILER!" text series
(http://compilers.iecc.com/crenshaw/)

Pascal version by Jean Debord for use with DMath, modified by V. Nesterov for LMath.

Following functions and operators are defined:

Operators: `+`, `-`, `*`, `/`, \ (integer division), `%` (modulus), `^` and `**` (exponentiation).

Bitwise: `>` shift right, `<` shift left, `&` and, `|` or, `$` xor, `!` not, `@` imp, `=` EQV.

Precedence: `!`, `&`, `|`, `$`, `=`, `@`, ⁏ `*` and `/`, `\`, `%`, `<` and `>`, `+` and `-`.
Parenthesis may be used to override the precedence.

In DMath library, only 26 variables could be defined and only first letter of a variable name was meaningful; number of functions was limited to 100. In LMath beginning from version 0.3, number of functions and variables is not limited and identifiers can have unlimited length.

Operator `**` is added to `^` for exponentiation. It may be necessary in some environments where `^` may have a special meaning.

Special variable `_Last` contains result of the last evaluation. Variables `Pi` and `Euler` are predefined, containing corresponding constants.

8.12.2 Functions and Procedures

InitEval

Declaration `function InitEval : Integer;`

Description Initializes expression evaluation system. Must be called before first call to `eval`. Returns number of defined functions.

SetVariable

Declaration `procedure SetVariable(VarName : String; Value : Float);` LMath

Description Defines variable `VarName` and initializes it with `Value`. Change in LMath: `VarName` is a string, may have arbitrary length and unlimited number of variables is possible.

SetFunction

Declaration `procedure SetFunction(FuncName : String; Wrapper : TWrapper);`

Description Defines a new function. `FuncName` is its name; `wrapper` is a function of TWrapper which will be actually called. Paramters of `FuncName` in the expression are copied into the vector of parameters for `Wrapper`.

Eval

Declaration `function Eval(ExpressionString : String) : Float;`

Description Actually evaluates expression in `ExpressionString` and returns result.

DoneEval

Declaration `procedure DoneEval;` LMath

Description Removes functions and variables. Call it after the end of session to free memory.

8.12.3 Variables

ParsingError

LMath

Declaration `ParsingError: boolean;`

Description Returns true if an error of expression parsing occurred, which means invalid expression. In LMath the variable was made public.

8.13 Unit ulinminq

8.13.1 Description

Minimization of a sum of squared functions along a line (Used internally by equation solvers)

8.13.2 Functions and Procedures

LinMinEq

Declaration `procedure LinMinEq(Equations : TEquations; X, DeltaX, F : TVector; Lb, Ub : Integer; R : Float; MaxIter : Integer; Tol : Float);`

Description Minimizes a sum of squared functions from point X in the direction specified by DeltaX, using golden search as the minimization algo.

Input parameters: SysFunc = system of functions; X = starting point; DeltaX = search direction; Lb, Ub = bounds of X; R = initial step, in fraction of $|DeltaX|$; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined minimum coordinates; F = function values at minimum; R = step corresponding to the minimum.

Possible results: OptOk = no error; OptNonConv = non-convergence.

8.14 Unit uCobyla. Constrained optimization by linear approximation

LMath 0.5

8.14.1 Description

This unit defines a procedure implementing the Constrained optimization by linear approximation (COBYLA) algorithm, initially developed by Michael J. D. Powell and adapted from Fortran 77 for LMath by V. Nesterov. Source code in Fortran can be found here. COBYLA is an optimization method for constrained problems which does not require knowing a derivative of the objective function. The procedure minimizes an objective function F(X) subject to M inequality constraints on X, where X is a vector of variables that has N components. Constrain expressions must be nonnegative. The algorithm employs linear approximations to the objective and constraint functions, the approximations being formed by linear interpolation at N+1 points in the space of the variables. These interpolation points are regarded as vertices of a simplex. The parameter RHO controls the size of the simplex and it is reduced automatically from RHOBEG to RHOEND. For each RHO the procedure tries to achieve a good vector of variables for the current size, and then RHO is reduced until the value RHOEND is reached. Therefore RHOBEG and RHOEND should be set to reasonable initial values and the required accuracy in the variables respectively, but this accuracy should be viewed as a subject for experimentation because it is not guaranteed.

8.14.2 Procedures and functions

COBYLA

Declaration procedure COBYLA(N: integer; M: integer; X: TVector;
out F: float; out MaxCV: float; RHOBEG: float; RHOEND: float;
var MaxFun: integer; CalcFC: TCobylaObjectProc);

Description

N: integer Input. Number of variables to optimize, residing in `X[N]` array.

M: integer Input. Number of inequality constrains.

X: TVector Array of variables to be optimized. Guess values on input, optimal values on output.

out F: float Output. Objective function value upon minimization.

out MaxCV: float Output. Maximal constraint violation after optimization.

RHOBEG: float Input. Initial size of simplex. Must be set by user. It is a subject of experimentation and depends on the size of the parameters.

RHOEND: float Input. End size of simplex: desired precision of objective function and constrain satisfaction.

var MaxFun: integer On input: Limit on the number of calls of CALCFC user-supplied function. On output: number of actual calls CalcFC: TCobylaObjectProc.

CalcFC: TCobylaObjectProc Objective function, type TCobylaObjectProc. The function receives vector of variables X as input and calculates the value of objective function as well as values of constraint expressions. After the optimizaion constraint expressions must be nonnegative, but, importantly, these constraints can be violated during the execution of the procedure.

8.15 unit uTrsTlp

LMath 0.5

8.15.1 Description

Unit uTrsTlp impplements linear optimization procedure uTrsTlp initially written by Michael J. D. Powell in Fortran 77 and adapted for LMath by V. Nesterov. This procedure is used by COBYLA algorithm, implemented in uCOBYLA unit.

This procedure calculates an N-component vector DX by applying the following two stages. In the first stage DX is set to the shortest vector that minimizes the greatest violation of the constraints

$$A[1,K]\cdot DX[1]+A[2,K]\cdot DX[2]+\cdots+A[N,K]\cdot DX[N]\geq[K], K=2,3,...,M,$$

subject to the Euclidean length of DX being at most RHO. If its length is strictly less than RHO, then we use the resultant freedom in DX to minimize the objective function

$$-A[1, M+1] \cdot DX[1] - A[2, M+1] \cdot DX[2] - \cdots - A[N, M+1] \cdot DX[N]$$

subject to no increase in any greatest constraint violation. This notation allows the gradient of the objective function to be regarded as the gradient of a constraint. Therefore the two stages are distinguished by $MCON = M$ and $MCON > M$ respectively. It is possible that a degeneracy may prevent DX from attaining the target length RHO. Then the value $IFULL = 0$ would be set, but usually $IFULL = 1$ on return.

In general NACT is the number of constraints in the active set and

$$IACT[1], \ldots, IACT[NACT]$$

are their indices, while the remainder of IACT contains a permutation of the remaining constraint indices. Further, Z is an orthogonal matrix whose first NACT columns can be regarded as the result of Gram-Schmidt applied to the active constraint gradients. For $J = 1, 2, ..., NACT$, the number ZDOTA[J] is the scalar product of the J-th column of Z with the gradient of the J-th active constraint. DX is the current vector of variables and here the residuals of the active constraints should be zero. Further, the active constraints have nonnegative Lagrange mulpliers that are held at the beginning of VMUltc. The remainder of this vector holds the residuals of the inactive constraints at DX, the ordering of the components of vmultc being in agreement with the permutation of the indices of the constraints that is in IACT. All these residuals are nonnegative, which is achieved by the shift RESMAX that makes the lest residual zero.

8.15.2 Procedure

TrsTlp

Declaration procedure TrsTlp(N, M : integer; A : TMatrix; B : TVector; RHO : float; DX : TVector; out IFULL : integer);

8.16 Unit uLinSimplex. Linear Programming

LMath 0.5

8.16.1 Description

Simplex method for linear programming. Adapted from Fortran 90, *Numerical Recipes.* Detailed description of principle and input and output parameters may be found in the "Numeric recipes in Fortran 77", pages 423-435.

Briefly, task of the linear programming is to maximize objective function

$$z = a_1x_1 + a_2x_2 + \cdots + a_Nx_N \tag{8.1}$$

with the primary constrains

$$x_1 \geq 0, \cdots, x_N \geq 0$$

and $M = m1 + m2 + m3$ secondary constrains in form:

$$a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \le b_i \quad (b_i \ge 0), \qquad i = 1, \ldots, m_1 \tag{8.2}$$

$$a_{j1}x_1 + a_{j2}x_2 + \cdots + a_{jN}x_N \ge b_j \quad (b_j \ge 0), \qquad j = m_1 + 1, \ldots, m_1 + m_2 \tag{8.3}$$

$$a_{k1}x_1 + a_{k2}x_2 + \cdots + a_{kN}x_N = b_k \quad (b_k \ge 0), \quad k = m1 + m2 + 1, \ldots, m_1 + m_2 + m_3 \tag{8.4}$$

a_{ji} coefficients can be positive, negative, or zero.

8.16.2 Functions and procedures

LinProgSolve

Declaration procedure LinProgSolve(var A : TMatrix; N, M1, M2, M3 : integer; out iCase: integer; out FuncVal : float; out SolVector :TVector);

Description LinProgSolve is a convenience wrapper around main procedure LinSimplex. Here A is a matrix containing tableau which describes the problem to be solved. It must have dimentions A[M+2,N+1].

In the first line, A[1,1] = 0, A[1,2] to A[1,N+1] contain coefficients for objective function, a_1 to a_N.
Lines A[2]..A[M+1] contain coefficients for constraints, in the order m1, m2, m3. That is:

"$\le$" constraints corresponding to Equation 8.2 are in A[2] to A[m1+1];
"$\ge$" constraints (Equation 8.3) in A[m1+2] to A[m1+m2+1];
"=" constraints (Equation 8.4) in A[m1+m2+2] to A[M+1].

First column A[2,1] to A[M+1] is occupied by free members (b_i, b_j and b_k in Equations 8.2 to 8.4). Cells A[2,2] to A[M+1,N+1] contain coefficients of constrains, a_{i1} to $a_{i_N}, i = 1, \ldots, M$.

Line A[M+2] is used internally for auxiliary function.

N is the number of coefficients in the objective function z (8.1), M1, M2 and M3 are the number of constrains in form 8.2, 8.3, and 8.4, correspondingly.

On output, iCase is a flag of an outcome of the calculation: icase = 0 means that finite solution was found; icase = 1: objective function is unbounded; icase = -1: no solution exists (constrains are internally contradictory). FuncVal is the value of the objective function after optimization, and SolVector contains the solution vector, X_1 in SolVector[1] etc.

LinSimplex

Declaration procedure LinSimplex(var A: TMatrix; N, M1, M2, M3 : integer; out icase : integer; out izrov, iposv : TIntVector);

Description In most cases there is no need to call `LinSimplex` directly, rather use `LinProgSolve`.

On input, parameters A, N, M1, M2, and M3 have the same meaning as in `LinProgSolve`. But, importantly, for LinProgSolve, sign of coefficients in constrain equations must be changed!

Output: `A` is revized Tableu; `A[1,1]` is objective function value. iposv and izrov are arrays indexing revized A. `High(iposv) = M`, `High(izrov) = N` where $M = M1 + M2 + M3$. `iposv[j]` ($j \in [1..M]$) contains index i of original variable $x[i]$, represented now by row $j + 1$ in A. If $iposv[j] > N$, then row `A[j+1]` represents a slack variable. First row in A is row of objective function.

izrov[k], $k \in [1..N]$, contains index i of a variable x[i] represented by column 1 to I. All these X are "0" in the solution, if $izrov[k] > N$, it represents a slack variable.

Chapter 9

Package lmNonLinEq: Units for Finding Roots of Non-Linear Equations

9.1 Unit ubisect

9.1.1 Description

Bisection method for nonlinear equation. Equation may be defined either as TFunc (`function Func(X : Float) : Float`) or as TParamFunc (`function Func(X : Float; Params:Pointer) : Float`) where Params may be pointer to any structure used by the target function.

9.1.2 Functions and Procedures

RootBrack

Declaration `procedure RootBrack(Func : TFunc; var X, Y, FX, FY : Float); overload;`
`procedure RootBrack(Func : TParamFunc; Params:Pointer; var X, Y, FX, FY : Float); overload;`

Description Expands the interval [X,Y] until it contains a root of Func, i. e. Func(X) and Func(Y) have opposite signs. The corresponding function values are returned in FX and FY;

Bisect

Declaration `procedure Bisect(Func : TFunc; var X, Y : Float; MaxIter : Integer; Tol : Float; out F : Float); overload;`
`procedure Bisect(Func : TParamFunc; Params: Pointer; var X, Y : Float; MaxIter : Integer; Tol : Float; out F : Float); overload;`

Description `Func` is a target function, `TFunc` or `TParamFunc`; the maximum number of iterations `MaxIter`; Initial values X; Y; the tolerance `Tol` with which the root must be located.

9.2 Unit ubroyden

9.2.1 Description

Broyden method for system of nonlinear equations

9.2.2 Functions and Procedures

Broyden

Declaration `procedure Broyden(Equations : TEquations; X, F : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float);`

Description Solves a system of nonlinear equations by Broyden's method.

Input parameters: Equations = subroutine to compute equations; X = initial guess values for roots; Lb, Ub = bounds of X; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined roots; F = function values.

Possible results: OptOk = no error; OptNonConv = non-convergence.

9.3 Unit unewteq

9.3.1 Description

Newton-Raphson solver for nonlinear equation.

9.3.2 Functions and Procedures

NewtEq

Declaration `procedure NewtEq (Func, Deriv : TFunc; var X : Float; MaxIter : Integer; Tol : Float; var F : Float);`

Description Solves a nonlinear equation by Newton's method.

Input parameters: Func = function to be solved; Deriv = derivative; X = initial root; MaxIter = maximum number of iterations; Tol = required precision.

Output parameters: X = refined root; F = function value.

Possible results: OptOk = no error; OptNonConv = non-convergence; OptSing = singularity (null derivative).

9.4 Unit unewteqs

9.4.1 Description

Newton-Raphson solver for system of nonlinear equations.

9.4.2 Functions and Procedures

NewtEqs

Declaration `procedure NewtEqs(Equations: TEquations; Jacobian : TJacobian; X, F : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float);`

Description Solves a system of nonlinear equations by Newton's method.

Input parameters: Equations = subroutine to compute equations; Jacobian = subroutine to compute Jacobian; X = initial root; MaxIter = maximum number of iterations; Tol = required precision

Output parameters: X = refined root; F = function values.

Possible results: OptOk = no error; OptNonConv = non-convergence; OptSing = singular jacobian matrix.

9.5 Unit usecant

9.5.1 Description

Secant method for nonlinear equation.

9.5.2 Functions and Procedures

Secant

Declaration `procedure Secant (Func : TFunc; var X, Y : Float; MaxIter : Integer; Tol : Float; out F : Float);`

Description function Func(X : Float) : Float; Initial values of X and Y; the maximum number of iterations MaxIter; the tolerance Tol with which the root must be located.

Chapter 10

Package lmDSP: Digital Signal Processing

10.1 Description

This package contains common routines for digital signal processing: Fast Fourier Transform (unit uFFT), Digital Fourier Transform, which does not require that input signal has length of power of two (unit uDFT; note however that this unit is distributed under GPL license, and not LGPL), and several signal filters (unit uFilters). Unit uFFT was moved here from lmRegression package, others were written for LMath ver 0.6.0.

10.2 Unit uConvolutions

LMath 0.6

10.2.1 Description

This unit defines a single function `Convolve` for convolution of two signals in time domain.

10.2.2 Procedures and functions

Declaration `function Convolve(constref Signal:array of float; constref FIR:array of float; Ziel : TVector = nil):TVector;`

Description `Function Convolve` convolves array `Signal` with array `FIR`. If `Ziel` is provided, result is placed into it beginning from Ziel[0], and `Ziel` is returned as result of function. Procedure is optimized for a case when `Signal` is longer than `FIR`, but works in the opposite case as well, only slightly slower.

10.3 Unit ufft

10.3.1 Description

Direct and inverse Fast Fourier Transform.

Note on API changes in LMath 0.6.0. In previous versions, `FFT`, `IFFT` and `FFT_Integer` were procedures and accepted both `InArray` and `OutArray` as `TCompVector`. In ver. 0.6.0 they were made functions, parameter `OutArray` is optional, and `InArray` is now open array of `Complex` instead of `TCompVector`. This makes these functions much more flexible, but supports backward compatibility, because when formal parameter has open array type, `TCompVector` can be passed as well.

Call of `FFT_Integer_Cleanup` is not necessary anymore. This function does nothing and is supported only for backward compatibility.

Modified from Don Cross:
http://groovit.disjunkt.com/analog/time-domain/fft.html

10.3.2 Functions and Procedures

FFT

Declaration `function FFT(NumSamples : Integer; constref InArray : array of Complex; OutArray : TCompVector = nil):TCompVector;`

Description Calculates the Fast Fourier Transform of the array of complex numbers ('InArray') and returns result of transform. `NumSamples` is number of samples; length of `InArray` must be `NumSamples`. If `OutArray` is assigned, its dimensions must be `[0..NumSamples-1]`. In this case, transform result is put into `OutArray` and pointer to it is returned, if `OutArray = nil`, `Result` is allocated by the function itself.

IFFT

Declaration `function IFFT( NumSamples : Integer; constref InArray : array of Complex; OutArray : TCompVector = nil) : TCompVector;`

Description Calculates the Inverse Fast Fourier Transform of the array of complex numbers ('InArray') and returns result of transform. `NumSamples` is number of samples; length of `InArray` must be `NumSamples`. If `OutArray` is assigned, its dimensions must be `[0..NumSamples-1]`. In this case, transform result is put into `OutArray` and pointer to it is returned, if `OutArray = nil`, `Result` is allocated by function itself.

FFT_Integer

Declaration `function FFT_Integer(NumSamples : Integer; constref RealIn, ImagIn : array of Integer; OutArray : TCompVector = nil) : TCompVector;`

Description Same as procedure FFT, but takes as input two open arrays of Integer (One for real, one for imaginary components) instead of one complex array. Similarly, lengths of RealIn and ImagIn must be equal to NumSampleslength of `InArray` must be `NumSamples`. If `OutArray` is assigned, its dimensions must be `[0..NumSamples-1]`. In this case, transform result is put into `OutArray` and pointer to it is returned, if `OutArray = nil`, `Result` is allocated by function itself.

FFT_Integer_Cleanup

Declaration `procedure FFT_Integer_Cleanup; deprecated;`

Description Call of this procedure is not necessary anymore, it is kept only for backward compatibility.

CalcFrequency

Declaration `function CalcFrequency(NumSamples, FrequencyIndex : Integer; InArray : TCompVector) : Complex;`

Description This function returns the complex frequency sample at a given index directly. Use this instead of 'FFT' when you only need one or two frequency samples, not the whole spectrum.

It is also useful for calculating the Discrete Fourier Transform (DFT) of a number of data which is not an integer power of 2. For example, you could calculate the DFT of 100 points instead of rounding up to 128 and padding the extra 28 array slots with zeroes.

10.4 Unit uDFT

LMath 0.6

10.4.1 Description

Unit uDFT includes procedures for direct and inverse digital Fourier transform of arrays with length different from degrees of two. Unit was contributed by David Chen, who translated it from C# version of AlgLib library by Sergey Bochkanov, Cooley-Tukey and Bluestein's algorithms are used. Important: this unit is distributed under GPL license, not LGPL as other parts of `LMath`.

10.4.2 Procedures and Functions

FFTC1D

Declaration `Procedure FFTC1D(var A: TCompVector; Lb, Ub: Integer);`

Description 1-dimensional complex DFT. Input parameters: `A` is `Complex` array to be transformed; `Lb, Ub` are lower and upper bounds of array slice to be transformed. `Lb` must be less or equal to `Ub`, otherwise `MatErrDim` error is set and no transform is done. Typically, but not necessary, `Lb` is 0 or 1 and `Ub` is `High(A)`. The problem size N, $N = Ub - Lb + 1$, could be any natural number. Output parameters: upon successful call,`A[Lb..Ub]` contains result of DFT transform. If the input data need to be preserved, back it up before calling.

FFTC1DInv

Declaration `Procedure FFTC1DInv(var A: TCompVector; Lb, Ub: Integer);`

Description 1-dimensional complex inverse FFT. Input parameters: `A` is an array of `Complex` to be transformed; `Lb, Ub` are lower and upper bounds of the array slice to be transformed. `Lb` must be less or equal to `Ub`, otherwise `MatErrDim` error is set and no transform is done. Typically, but not necessarily, `Lb` is 0 or 1 and `Ub` is `High(A)`. The problem size N, $N = Ub - Lb + 1$, could be any natural number. Output parameters: upon successful call,`A[Lb..Ub]` contains result of inverse DFT transform. If the input data need to be preserved, back it up before calling.

FFTR1D

Declaration Procedure FFTR1D(const A: TVector; Lb, Ub: Integer; out F: TCompVector);

Description 1-dimensional real FFT. Input parameters: A is array of `Float` to be transformed; `Lb, Ub` are Lower and upper bounds of the array slice to be transformed. `Lb` must be less or equal to `Ub`, otherwise `MatErrDim` error is set and no transform done. Typically, but not necessary, `Lb` is 0 or 1 and `Ub` is `High(A)`. The problem size N, $N = Ub - Lb + 1$, could be any natural number. Output parameters: `F` is the result of DFT of `A`. `F` is a complex array with range `[0..(Ub-Lb)]`. Note: F[] satisfies symmetry property $F[k] = \text{conj}(F[N - k])$, so just one half of array is usually needed. But for convenience, subroutine returns full complex array (with frequencies above N/2), so its result may be used by other FFT-related subroutines. $N = Ub - Lb$.

FFTR1DInv

Declaration `Procedure FFTR1DInv(const F: TCompVector; Lb, Ub: Integer; var A: TVector);`

Description 1-dimensional real inverse FFT. Input parameters: `F` array of Complex containing frequencies from forward real FFT. `Lb, Ub` are bounds of the array slice to be transformed. `Lb` must be less or equal to `Ub`, otherwise `MatErrDim` error is set and no transform done. Typically, but not always, `Lb` is 0 or 1 and `Ub` is `High(F)`. The problem size N, which equals $Ub - Lb + 1$, could be any natural number. Output parameters: `A` is result of Inverse DFT of the input array. Its dimensions are [0..(Ub-Lb)]. Note: F[] should satisfy symmetry property $F[k] = \text{conj}(F[N-k])$. Only one half of frequencies array F, namely, elements from Lb to `floor(Lb + N/2)`, is used. However, this function doesn't check the symmetry of F[Lb..Ub], so the caller have to make sure it before use. F[0] is always real. If `N` is even, F[floor(N/2)] is real too. If `N` is odd, then F[floor(N/2)] has no special properties. $N = Ub - Lb$.

10.5 Unit uFilters

LMath 0.6

10.5.1 Description

This unit implements several digital filters: low pass moving average and gaussian filters, one-pole stop-band (notch) and pass band filters, one-pole high-pass filter, Chebyshev filter, which can be used both as low- and high-pass; with ripple = 0 setting it is a Butterworth filter; median filter. Besides that, the unit contains several functions for calculation of filter parameters. `GaussCascadeFreq` allows to find effective cut-off frequency for a case when two Gaussian filters were used one after another (and, because cascade of gaussian filters is also gaussian filters, repeatitive use of this function allows to find effective frequency of a cascade of any arbitrary length). `GaussRiseTime` allows to find a rise time of a step response of gaussian filter with a given cut-off frequency. Rise time is defined as time of the rize from 10% to 90% of complete amplitude. `MoveAvRiseTime` finds a rise time of moving average filter with a given window length. `MoveAvCutOffFreq` finds the cut-off frequency of Moving Average Filter with a given window length. `MoveAvFindWindow` solves the opposite task: finds window length for a given cut-off frequency. Cut-off frequency of all filters is defined as a frequency at which power of signal is reduced to 50% of initial value, which corresponds to amplitude reduction by $\sqrt{2}/2 \approx 0.7$. Most of filter procedures require sampling rate and cut-off frequency. If sampling frequency is not defined in your allipcation, you may set it to 1 and define cut-off frequency as its fraction. Note also that typically cut-off frequency cannot be higher than half of sampling rate.

Gaussian and moving average filters efficiently remove high frequency noise preserving form of signal in time domain and serve as excellent smoothing filters. However, they have relatively poor performance in the frequency domain. In contrast, Chebyshev filter has a steep frequency response, but causes large ringing in time domain. Therefore it may be an excellent analytical frequency domain filter, but is not recommended for curve smoothing for data viewed in time domain. Gaussian filter is implemented as described in:
Young I.T., L.J. van Vliet. Recursive implementation of the Gaussian Filter. //

Signal Processing, 44 (1995) 139-151.
Chebyshev filter and one-pole recursive filters implement algorithms from
Smith, S. W. The Scientist and Engineer's Guide to Digital Signal Processing

10.5.2 Functions and Procedures

GaussFilter

Declaration `procedure GaussFilter(var Data:array of float; ASamplingRate: Float; ACutFreq: Float);`

Description Implements low-pass gaussian filter. Open array `Data` on input contains initial signal, on output, filtered signal. `ASampling rate` is sampling rate, `ACutFreq` is cut-off frequency. Errors: `lmTooHighFreqError` is set if `ACutFreq` > `ASamlingRate/2` and `Data` remains unchanged. See 10.5.1 for definition of cut-off frequency.

MovingAverageFilter

Declaration procedure MovingAverageFilter(var Data:array of float; WinLength:integer);

Description Performs smoothing of `Data` array of Float with moving average filter. `Data` on input contains initial signal, on output, filtered signal. `WinLength` is length of averaged window. Errors: lmDSPFilterWinError is set if `WinLength` > `length(Data)` and `Data` remains unchanged. If you want to define moving average filter in terms of cut-off frequency, desired window length may be found with a call to `MovAvFindWindow` (??). Conversly, to find cut-off frequency corresponding to known window length, use `MoveAvCutOffFreq`, 10.5.2.11. See 10.5.1 for definition of cut-off frequency.

MedianFilter

Declaration procedure MedianFilter(var Data:array of float; WinLength:integer);

Description Median filter is ideal to remove short spikes preserving sharp edges. Window length must be set to spike length + 1. `Data` on input contains initial signal, on output, filtered signal. `WinLength` is length of averaged window. Errors: lmDSPFilterWinError is set if `WinLength` > `length(Data)` and `Data` remains unchanged.

NotchFilter

Declaration procedure NotchFilter(var Data:array of float; ASamplingRate: Float; AFreqReject: Float; ABW: Float);

Description Notch filter rejects `AFreqReject`. It may be useful for example to remove hum, originating from a power network 50 (in USA 60) Hz frequency. `ABW` is rejected bandwidth, measured at 0.5 power (0.7 amplitude). Increasing this value increases amplitude and decreases length of ringing after steps in filtered signal. `Data` contains initial signal on input and filtered on output. Errors: `lmTooHighFreqError` is set if ACutFreq > ASamlingRate/2 and `Data` remains unchanged. See 10.5.1 for definition of cut-off frequency.

BandPassFilter

Declaration procedure BandPassFilter(var Data:array of float; ASamplingRate: Float; AFreqPass: Float; ABW: Float);

Description Opposite to `NotchFilter`. Passes `AFreqPass` and rejects everything else. `Data` contains initial signal on input and filtered on output. `ABW` is rejected the width of passed band, measured at 0.5 power (0.7 amplitude). Errors: `lmTooHighFreqError` is set if ACutFreq > `ASamlingRate/2` and `Data` remains unchanged.

HighPassFilter

Declaration procedure HighPassFilter(var Data:array of float; ASamplingRate: Float; ACutFreq: Float);

Description One pole highpass filter. `Data` contains initial signal on input and filtered on output. `ASampling rate` is sampling rate, `ACutFreq` is cut-off frequency. See 10.5.1 for definition of cut-off frequency. Errors: if `ACutFreq` > `ASamplingRate/2`, lmTooHighFreqError is set and `Data` remains unchanged.

ChebyshevFilter

Declaration procedure ChebyshevFilter(var Data:array of float; ASamplingRate: Float; ACutFreq: Float; NPoles: integer; PRipple: float; AHighPass:boolean);

Description Defines Chebyshev filter, which may be high or low pass. Chebyshev filters have steep frequency response, but distort data shape in time domain. Increase of poles number makes the frequency response steeper, but decreases filter stability and of course increases calculation load. Typically, it is good idea to keep $NPoles \leq 6$ if $Float = Single$. `NPoles` cannot be greater than 10. Generally, Chebyshev filters have ripple in pass band; higher is this ripple, steeper is the signal attenuation. Value of ripple as % of passband amplitude is controlled by `PRipple` parameter; $0 \leq PRipple \leq 29.0$. Setting `PRipple` to 0 converts Chebyshev filter to Butterworth filter.

On input: `Data` is initial signal; `ASamplingRate` is sampling rate; `ACutFreq` is cut-off frequency; `NPoles` is number of poles. It must be positive even, less or equal to 10. `PRipple` is allowed value of ripple in the passband as %. $0 \geq PRipple \geq 29$. AHighPass: if true, highpass filtering will be done, lowpass otherwise. Output: filtered array in `Data`. Errors: `lmTooHighFreqError` is set if `ACutFreq` > `ASamlingRate/2`, `lmPolesNumError` is set if `NPoles` is not in [2,4,6,8,10] and `lmFFTBadRipple` if $PRipple < 0$ or $PRipple > 29$. In all these cases `Data` remains unchanged.

GaussCascadeFreq

Declaration function GaussCascadeFreq(Freq1, Freq2:Float):Float;

Description Returns effective cut-off frequency of cascade of 2 gaussian filters, each with cut-off frequencies `Freq1` and `Freq2`. If `Freq1 = 0`, `Freq2` is returned and *vice versa*. Errors: if any of `Freq1` or `Freq2` is negative, `FDomain` error is set and 0 is returned.

GaussRiseTime

Declaration GaussRiseTime(Freq: Float): Float;

Description Calculates Rise time of step response for gaussian filter with cut-off frequency `Freq`. Rise time is defined as time of rise from 10% of step amplitude to 90%. If $Freq = 0$, `MaxNum` is returned. Errors: if $Freq < 0$, FDomain is set and 0 is returned.

MovAvRiseTime

Declaration function MovAvRiseTime(SamplingRate:Float; WLength:integer):Float;

Description Calculates rise time from 0 to 100% of step response for moving average filter with window length `WLength` and sampling rate `SamplingRate`. Errors: if `SamplingRate` ≤ 0 or `WLength` < 1, `FDomain` error is set and 0 is returned.

MoveAvCutOffFreq

Declaration function MoveAvCutOffFreq(SamplingRate:Float; WLength:integer):Float;

Description Calculates cut-off frequency of moving average filter with SamplingRate `SamplingRate` and window length `WLength`. Errors: Errors: if `SamplingRate` ≤ 0 or `WLength` < 1, `FDomain` error is set and 0 is returned.

MoveAvFindWindow

Declaration function MoveAvFindWindow(SamplingRate, CutOffFreq:Float):Integer;

Description Calculates required window length from desired cut-off frequency (`CutOffFreq`) and given sampling rate (`SamplingRate`). Errors: if `SamplingRate` ≤ 0 or `CutOffFreq` ≤ 0, `FDomain` error is set; if `CutOffFreq` $>$ `ASamplingRate/2`, lmTooHighFreqError is set. In all these cases 0 is returned.

Chapter 11

Package lmRegression: Linear and Non-Linear Regression and Curve Fitting

Procedures for non-linear regression and data fitting with some general models are collected in this package. Units uLinFit, uMultFit and uSVDfit include procedures for linear and multple linear regression; unit unlfit contains algorithms for general non-linear regression; other units contain a library of common regression models. In addition, unit uSpline provides spline interpolation of experimental data and procedures for exploration of the spline function.

11.1 Unit ulinfit

11.1.1 Description

Linear regression:

$$Y = B_0 + B_1 X$$

11.1.2 Functions and Procedures

LinFit

Declaration `procedure LinFit(X, Y : TVector; Lb, Ub : Integer; B : TVector; V : TMatrix);`

Description Unweighted linear regression. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds. Output parameters: B = regression parameters; V = inverse matrix, [0..2,0..2].

WLinFit

Declaration `procedure WLinFit(X, Y, S : TVector; Lb, Ub : Integer; B : TVector; V : TMatrix);`

Description Weighted linear regression. Additional input parameter: S = standard deviations of observations.

SVDLinFit

Declaration `procedure SVDLinFit(X, Y : TVector; Lb, Ub : Integer; SVDTol : Float; B : TVector; V : TMatrix);`

Description Unweighted linear regression by singular value decomposition. SVDTol = tolerance on singular values.

WSVDLinFit

Declaration `procedure WSVDLinFit(X, Y, S : TVector; Lb, Ub : Integer; SVDTol : Float; B : TVector; V : TMatrix);`

Description Weighted linear regression by singular value decomposition.

11.2 Unit umulfit

11.2.1 Description

Multiple linear regression (Gauss-Jordan method)

$$Y = B_0 + B_1 X_1 + B_2 X_2 + \cdots + B_d X_d$$

11.2.2 Functions and Procedures

MulFit

Declaration `procedure MulFit(X : TMatrix; Y : TVector; Lb, Ub, Nvar : Integer; ConsTerm : Boolean; B : TVector; V : TMatrix);`

Description Input parameters:

- X = matrix of independent variables, [Lb..Ub,1..NVar];
- Y = vector of dependent variable;
- Lb, Ub = array bounds;
- NVar = number of independent variables;
- ConsTerm = presence of constant term B(0).

Output parameters: B = regression parameters; V = inverse matrix, [0..d,0..d].

WMulFit

Declaration `procedure WMulFit(X : TMatrix; Y, S : TVector; Lb, Ub, Nvar : Integer; ConsTerm : Boolean; B : TVector; V : TMatrix);`

Description Weighted multiple linear regression. S = standard deviations of observations, other parameters as in MulFit.

11.3 Unit usvdfit

11.3.1 Description

Multiple linear regression (Singular Value Decomposition)

11.3.2 Functions and Procedures

SVDFit

Declaration `procedure SVDFit(X : TMatrix; Y : TVector; Lb, Ub, Nvar : Integer; ConsTerm : Boolean; SVDTol : Float; B : TVector; V : TMatrix);`

Description Input parameters: X = matrix of independent variables; Y = vector of dependent variable; Lb, Ub = array bounds; Nvar = number of independent variables; ConsTerm = presence of constant term B(0). SVDTol = tolerance on singular values. Output parameters: B = regression parameters; V = inverse matrix.

WSVDFit

Declaration `procedure WSVDFit(X : TMatrix; Y, S : TVector; Lb, Ub, Nvar : Integer; ConsTerm : Boolean; SVDTol : Float; B : TVector; V : TMatrix);`

Description Weighted multiple linear regression. S = standard deviations of observations. Other parameters as in SVDFit.

11.4 Unit unlfit

11.4.1 Description

Nonlinear regression. This unit defines generic procedures for non-linear regression which are used further in all non-linear models in the library.

11.4.2 Functions and Procedures

SetOptAlgo

Declaration `procedure SetOptAlgo(Algo : TOptAlgo);`

Description Sets the optimization algorithm according to Algo:TOptAlgo, which must be NL_MARQ, NL_SIMP, NL_BFGS, NL_SA, NL_GA. Default is NL_MARQ.

GetOptAlgo

Declaration `function GetOptAlgo : TOptAlgo;`

Description Returns the optimization algorithm.

SetMaxParam

Declaration `procedure SetMaxParam(N : Byte);`

Description Sets the maximum number of regression parameters.

GetMaxParam

Declaration `function GetMaxParam : Byte;`

Description Returns the maximum number of regression parameters.

SetParamBounds

Declaration `procedure SetParamBounds(I : Byte; ParamMin, ParamMax : Float);`

Description Sets the bounds on the I-th regression parameter.

GetParamBounds

Declaration `procedure GetParamBounds(I : Byte; var ParamMin, ParamMax : Float);`

Description Returns the bounds on the I-th regression parameter.

NullParam

Declaration `function NullParam(B : TVector; Lb, Ub : Integer) : Boolean;`

Description Checks if a regression parameter is equal to zero.

NLFit

Declaration `procedure NLFit(RegFunc : TRegFunc; DerivProc : TDerivProc; X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; FirstPar, LastPar : Integer; V : TMatrix);`

Description Unweighted nonlinear regression. Input parameters: RegFunc:TRegFunc = regression function to be modeled; DerivProc = procedure to compute derivatives; X, Y = point coordinates; Lb, Ub = array bounds; MaxIter = max. number of iterations; Tol = tolerance on parameters; B = initial parameter values; FirstPar = index of first regression parameter; LastPar = index of last regression parameter; Output parameters: B = fitted regression parameters; V = inverse matrix. Its dimentions must be [Lb..Ub, Lb..Ub], or [0..Ub,0..Ub]. The matrix must be allocated, but does not require any initialization.

WNLFit

Declaration `procedure WNLFit(RegFunc : TRegFunc; DerivProc : TDerivProc; X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; FirstPar, LastPar : Integer; V : TMatrix);`

Description Weighted nonlinear regression. S = standard deviations of observations. Other parameters as in NLFit.

SetMCFile

Declaration `procedure SetMCFile(FileName : String);`

Description Set file for saving MCMC simulations

SimFit

Declaration `procedure SimFit(RegFunc : TRegFunc; X, Y : TVector; Lb, Ub : Integer; B : TVector; FirstPar, LastPar : Integer; V : TMatrix);`

Description Simulation of unweighted nonlinear regression by Markov chain Monte Carlo (MCMC) method.

WSimFit

Declaration `procedure WSimFit(RegFunc : TRegFunc; X, Y, S : TVector; Lb, Ub : Integer; B : TVector; FirstPar, LastPar : Integer; V : TMatrix);`

Description Simulation of weighted nonlinear regression by MCMC.

11.5 Unit uConstrNLFit

LMath 0.5

11.5.1 Description

This unit implements fit of a data with non-linear regression models including arbitrary constrains on the model variables and their relations. Fitting procedure uses the COBYLA algorithm, implemented in the unit uCOBYLA, see 8.14.

11.5.2 Types

TConstrainsProc

Declaration TConstrainsProc = procedure(MaxCon: integer; B, Con : TVector);

Description This is function type for calculation of constraint expressions supplied to COBYLA algorithm. It calculates constraint functions and puts results of calculation into `Con` array. `B` is vector of model parameters as in ConstrNLFit. Constraint calculation results are placed from `Con[1]` and ending with `Con[LastCon]`. At the end they must be nonnegative. `Con[0]` is not used, Fortran inheritance. `Con` is allocated by the fitting procedure.

11.5.3 Procedures and Functions

ConstrNLFit

Declaration procedure ConstrNLFit(
RegFunc: TRegFunc; ConstProc: TConstrainsProc; X, Y: TVector;
Lb, Ub: Integer; var MaxFun: Integer; var Tol: Float; B: TVector;
LastPar: Integer; LastCon: Integer; out MaxCV: float);

Description Non-linear regression with constrains. Parameters: RegFunc:`TRegFunc` = regression function fitting the data; `ConstProc:` `TConstrainsProc` = procedure calculating constrain expressions, which must be non-negative; `X, Y` = point coordinates of the data to be fitted; `Lb, Ub` = bounds of `X` and `Y` arrays; `MaxFun` on input is maximal number of calls to objective function, to avoid endless looping; on output, actual number of calls; `Tol` = tolerance of fit (RhoEnd); `B` = vector of parameters, guesses in input, fitted values on output; `LastPar` = number of parameters in `B`. First parameter is placed in in `B[1]`, last in `B[LastParam]`; `LastCon` = number of constraints; `MaxCV` = maximal constraint violation.

GetCFFittedData

Declaration function GetCFFittedData: TVector;

Description Function which returns calculated values of regression function, corresponding to values from `X` array.

GetCFResiduals

Declaration function GetCFResiduals: TVector;

Description Returns residuals (differences between data values supplied in Y array and calculated values, as returned by GetCFFittedData.)

11.5.4 Variables

RhoBeg

Declaration RhoBeg : float = 1.0;

Description Variable which defines RhoBeg parameter for the call of COBYLA algorithm. Set it before call to ConstrNLFit, or leave default value if you have no idea about an optimal one. Must be scaled according to the scale of the variables and their uncertainty and is a subject of experimentation.

11.6 Unit uevalfit

11.6.1 Description

Fitting of a user-defined function.

11.6.2 Functions and Procedures

InitEvalFit

Declaration `procedure InitEvalFit(ExpressionString : String);`

Description Defines a regression model from ExpressionList. The independent variable is denoted by 'x'. The regression parameters are denoted by single-character symbols, from 'a' to 'w'. Example: `InitEvalFit('a * exp(-k * x)')`

FuncName

Declaration `function FuncName : String;`

Description Returns the name of the regression function (= ExpressionString).

LastParam

Declaration `function LastParam : Integer;`

Description Returns the index of the last regression parameter

ParamName

Declaration `function ParamName(I : Integer) : String;`

Description Returns the name of the I-th regression parameter.

EvalFit

Declaration `procedure EvalFit(X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of the function defined by InitEvalFit

Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters; V = inverse matrix.

WEvalFit

Declaration `procedure WEvalFit(X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of the function defined by InitEvalFit. Additional input parameter: S = standard deviations of observations.

EvalFit_Func

Declaration `function EvalFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

11.7 Unit uiexpfit

11.7.1 Description

This unit fits the increasing exponential :

$$y = Y_{min} + A(1 - \exp(-kx))$$

11.7.2 Functions and Procedures

IncExpFit

Declaration `procedure IncExpFit(X, Y : TVector; Lb, Ub : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of the model. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; ConsTerm = flag of the presence of the constant term (Ymin). MaxIter = max. number of iterations; Tol = tolerance on the parameters. Output parameters: B = regression parameters, such that: B[0] = Y_{min}, B[1] = A, B[2] = k; V = inverse matrix, [0..2,0..2].

WIncExpFit

Declaration `procedure WIncExpFit(X, Y, S : TVector; Lb, Ub : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of the model. Additional input parameter: S = standard deviations of the observations.

IncExpFit_Func

Declaration `function IncExpFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

11.8 Unit uexpfit

11.8.1 Description

This unit fits a sum of decreasing exponentials:

$$y = Y_{min} + A_1 \exp(-a_1 x) + A_2 \exp(-a_2 x) + A_3 \exp(-a_3 x) + \cdots + A_d \exp(-a_d x)$$

11.8.2 Functions and Procedures

ExpFit

Declaration `procedure ExpFit(X, Y : TVector; Lb, Ub, Nexp : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of the sum of exponentials. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; Nexp = number of exponentials; ConsTerm = presence of constant term B(0); MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters; V = inverse matrix, $[0..2N_{exp}, 0..2N_{exp}]$.

Regression parameters: $B[0] = Y_{min},\ B[1] = A_1,\ B[2] = a_1,\ B[2i-1] = A_i,\ B[2i] = a_i;\ i = 1..N_{exp}$.

WExpFit

Declaration `procedure WExpFit(X, Y, S : TVector; Lb, Ub, Nexp : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of sum of exponentials.

Additional input parameter: S = standard deviations of observations.

ExpFit_Func

Declaration `function ExpFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

11.9 Unit uexlfit

11.9.1 Description

This unit fits the "exponential + linear" model:

$$y = A(1 - \exp(-kx)) + Bx$$

11.9.2 Functions and Procedures

ExpLinFit

Declaration `procedure ExpLinFit(X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of model

Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that:

B[0] = A, B[1] = k, B[2] = B;

V = inverse matrix, [0..2,0..2].

WExpLinFit

Declaration `procedure WExpLinFit(X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of model.

Additional input parameter: S = standard deviations of observations.

ExpLinFit_Func

Declaration `function ExpLinFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

11.10 Unit upolfit

11.10.1 Description

Polynomial regression :

$$Y = B_0 + B_1X + B_2X^2 + ... + B_dX^d$$

11.10.2 Functions and Procedures

PolFit

Declaration `procedure PolFit(X, Y : TVector; Lb, Ub, Deg : Integer; B : TVector; V : TMatrix);`

Description Unweighted polynomial regression. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; Deg = degree of polynomial. Output parameters: B = regression parameters; V = inverse matrix, [0..Deg, 0..Deg].

WPolFit

Declaration `procedure WPolFit(X, Y, S : TVector; Lb, Ub, Deg : Integer; B : TVector; V : TMatrix);`

Description Weighted polynomial regression. Additional input parameter: S = standard deviations of observations.

SVDPolFit

Declaration `procedure SVDPolFit(X, Y : TVector; Lb, Ub, Deg : Integer; SVDTol : Float; B : TVector; V : TMatrix);`

Description Unweighted polynomial regression by singular value decomposition. SVDTol = tolerance on singular values.

WSVDPolFit

Declaration `procedure WSVDPolFit(X, Y, S : TVector; Lb, Ub, Deg : Integer; SVDTol : Float; B : TVector; V : TMatrix);`

Description Weighted polynomial regression by singular value decomposition.

11.11 Unit ufracfit

11.11.1 Description

This unit fits a rational fraction:

$$y = \frac{p_0 + p_1 x + p_2 x^2 + ... + q_{d_1} x^{d_1}}{q_0 + q_1 x + q_2 x^2 + ... + q_{d_2} x^{d_2}}$$

11.11.2 Functions and Procedures

FracFit

Declaration `procedure FracFit(X, Y : TVector; Lb, Ub : Integer; Deg1, Deg2 : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of rational fraction. Input parameters: X, Y = point coordinate; Lb, Ub = array bounds; Deg1, Deg2 = degrees of numerator and denominator; ConsTerm = presence of constant term p_0; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that :

$B[0] = p_0,\ B[1] = p_1,\ B[2] = p_2,\ \ldots, B[Deg1] = p_d1$

$B[Deg1 + 1] = q_0,\ B[Deg1 + 2] = q_1,\ \ldots, B[Deg1 + Deg2 + 1] = p_d2;$

V = inverse matrix, $[0..Deg1 + Deg2 + 1, 0..Deg1 + Deg2 + 1]$.

WFracFit

Declaration `procedure WFracFit(X, Y, S : TVector; Lb, Ub : Integer; Deg1, Deg2 : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of rational fraction. Additional input parameter: S = standard deviations of observations.

FracFit_Func

Declaration `function FracFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

11.12 Unit ugamfit

11.12.1 Description

This unit fits the gamma variate regression model:

$$y = a(x - b)^c \exp\left(-\frac{x - b}{d}\right)$$

11.12.2 Functions and Procedures

GammaFit

Declaration `procedure GammaFit(X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of model. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that:

B[1] = a, B[2] = b, B[3] = c, B[4] = d;

V = inverse matrix, [0..4,0..4].

WGammaFit

Declaration `procedure WGammaFit(X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of model. Additional input parameter: S = standard deviations of observations.

GammaFit_Func

Declaration `function GammaFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

11.13 Unit ulogifit

11.13.1 Description

This unit fits the logistic function :

$$y = A + \frac{B - A}{1 + \exp(-\alpha x + \beta)}$$

and the generalized logistic function :

$$y = A + \frac{B - A}{(1 + \exp(\alpha x + \beta))^n}$$

11.13.2 Functions and Procedures

LogiFit

Declaration `procedure LogiFit(X, Y : TVector; Lb, Ub : Integer; ConsTerm : Boolean; General : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of logistic function. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; ConsTerm = presence of constant term A; General = generalized logistic; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that: B[0] = A; B[1] = B; B[2] = α; B[3] = β; B[4] = n.

V = inverse matrix, [0..4,0..4].

WLogiFit

Declaration `procedure WLogiFit(X, Y, S : TVector; Lb, Ub : Integer; ConsTerm : Boolean; General : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of logistic function. Additional input parameter: S = standard deviations of observations.

LogiFit_Func

Declaration `function LogiFit_Func(X : Float; B : TVector) : Float;`

Description Computes the regression function at point X. B is the vector of parameters.

11.14 Unit upowfit

11.14.1 Description

This unit fits a power function :

$$y = Ax^n$$

11.14.2 Functions and Procedures

PowFit

Declaration `procedure PowFit(X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of model. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that :

B[0] = A, B[1] = n;

V = inverse matrix, [0..1,0..1].

WPowFit

Declaration `procedure WPowFit(X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of model. Additional input parameter: S = standard deviations of observations.

PowFit_Func

Declaration `function PowFit_Func(X : Float; B : TVector) : Float;`

Description Computes the regression function at point X. B is the vector of parameters, such that:

B[0] = A; B[1] = n.

11.15 Unit uregtest

11.15.1 Description

Goodness of fit tests

11.15.2 Functions and Procedures

RegTest

Declaration `procedure RegTest(Y, Ycalc : TVector; LbY, UbY : Integer; V : TMatrix; LbV, UbV : Integer; out Test : TRegTest);`

Description Test of unweighted regression. Input parameters: Y, Ycalc = observed and calculated Y values; LbY, UbY = bounds of Y and Ycalc; V = inverse matrix, as returned by regression roitine; LbV, UbV = bounds of V; Output parameters: V = variance-covariance matrix; Test = test results.

WRegTest

Declaration `procedure WRegTest(Y, Ycalc, S : TVector; LbY, UbY : Integer; V : TMatrix; LbV, UbV : Integer; out Test : TRegTest);`

Description Test of weighted regression. Additional input parameter: S = standard deviations of observations.

11.16 Unit uSpline

LMath

11.16.1 Functions and Procedures

InitSpline

Declaration `procedure InitSpline(Xv, Yv:TVector; out Ydv:TVector; Lb,Ub:integer);`

Description Input parameters: `Xv`, `Yv` are data points; `Lb`, `Ub`: array bounds; Output: `Ydv`: cubic spline values used in calls to `Splint`. Vector `Ydv` is allocated by `InitSpline` and after it has length `Ub - Lb + 1`. This procedure must be called before actual drawing with `Splint` function or spline investigation with `SplDeriv` and `FindSplineExtremums`.

SplInt

Declaration `function SplInt(X:Float; Xv, Yv, Ydv: TVector; Lb,Ub:integer):Float;`

Description After preparing to draw by `InitSpline`, this function returns `Y` value at `X`. Input parameters: `Xv, Yv`: data points, same as at a call of `InitSpline`; `Ydv`: vector of spline data as returned by `InitSpline`; `Lb,Ub`: array bounds; `X`: independent variable. Returns spline value at `X`.

SplDeriv

Declaration `function SplDeriv(X:Float; Xv, Yv, Ydv: TVector; Lb, Ub:integer):float;`

Description Returns first derivative to spline function at any given point.

FindSplineExtremums

Declaration
```
procedure FindSplineExtremums(Xv,Yv,Ydv:TVector;
Lb,Ub:integer; out Minima, Maxima:TRealPointVector; out NMin,
NMax:  integer; ResLb :  integer = 1);
```

Description Finds all local minima and maxima of a spline between points `Lb` and `Ub`. Input parameters are the same as for `Splint`, except `X`. Output: found maximums in `Maxima`; minimums are in `Minima`; `Minima[j].X` is abscissa and `Minima[j].Y` an ordinate of extremum. Numbering of output arrays begins from `ResLb`, default value is 1. `Minima` and `Maxima` are allocated by `FindSplineExtremums`. Number of found minima is returned in `NMin`; of maxima, in `NMax`.

Chapter 12

Package lmSpecRegress: Specialized Regression Models

12.1 Description

This package contains collection of some regression models, specific for paricular fields of knowledge. Currently it includes equation of enzyme kinetics (Michaelis-Menten equation, Hill equation), chemistry (acid-base titration curve), electrophysiology (Goldman-Hodgkin-Katz Equation for current), and some statistic distributions.

12.2 Unit uDistribs

LMath

12.2.1 Description

This unit defines several distributions and instruments to model experimental data with these distributions. Defined are binomial, exponential, hypoexponential and hyperexponential distributions.

12.2.2 Functions and Procedures

dBinom

Declaration `function dBinom(k,n:integer;q:Float):Float;`

Description Returns binomial probability density for value k in test with n trials and q probability of success in one trial. If k>n returns 0.

ExponentialDistribution

Declaration `function ExponentialDistribution(beta, X:float):float;`

Description Evaluates exponential probability density with $\beta = beta$ for given X. If $X \leq 0$ returns 0.

HyperExponentialDistribution

Declaration `function HyperExponentialDistribution(N:integer; var Params:TVector; X:float):float;`

Description N defines number of phases; Params: zero-based TVector[2*N] contains pairs of parameters for each phase: probability and time (not rate!) constant. Sum of all probabilities must be 1.

Fit2HyperExponents

Declaration `procedure Fit2HyperExponents(var Xs, Ys:TVector; Ub:integer; var P1, beta1, beta2:float);`

Description Estimates parameters of hyperexponential distribution with 2 phases using Marqardt algorithm

HypoExponentialDistribution2

Declaration `function HypoExponentialDistribution2(beta1, beta2, X:float):float;`

Description PDF of the hypoexponential distribution with 2 phases; beta1, beta2 are time constants (not rate constants!)

EstimateHypoExponentialDistribution

Declaration `procedure EstimateHypoExponentialDistribution(M,CV:float; out beta1, beta2:float);`

Description Analytic estimate of the parameters of hypoexponential distribution

Fit2Hypoexponents

Declaration `procedure Fit2Hypoexponents(var Xs, Ys: TVector; Ub:integer; var beta1, beta2:float);`

Description Iterative fit of hypoexponential distribution.

12.2.3 Types

TBinomialDistribFunction

Declaration `TBinomialDistribFunction = function(k, n:integer; q:Float):Float;`

12.3 Unit uGauss

LMath

12.3.1 Description

This unit fits experimental data with a multigaussian distribution which is a sum of several gaussian distributions; each has own mathematical expectancy and probability to occur, while variance is the same for all:

$$\begin{cases} pdf(x) = p_0 g(x, \mu_0, \sigma_0) + \sum_{i=1}^{n} p_i g(x, \mu_i, \sigma) \\ \sum_{i=0}^{n} p_i = 1 \end{cases} \tag{12.1}$$

Such distributions occur in patch-clamp experiments (distribution of current values over time in a recording with several active channels) and in chromatography (several peaks).

12.3.2 Classes, Interfaces, Objects and Records

ENoSigma Class

Hierarchy

ENoSigma > Exception

12.3.3 Functions and Procedures

FindSigma

Declaration `function FindSigma(var XArray, YArray:TVector; TheLength, MuPos :integer):Float;`

Description Quick and rough estimate of sigma for normal distribution, if μ is known and upper part of the empiric probability density curve is known. uses SigmaArray. Used internally to get guess value for the fit.

ScaledGaussian

Declaration `function ScaledGaussian(mu, sigma, ScF, X:float):float;`

Description Evaluates pdf of a gaussian distribution with mathematical expectance μ and variation σ, scaled by factor ScF.

SumGaussians

Declaration `function SumGaussians(X:Float; Params:TVector):float;`

Description Evaluates sum of gaussians where σ is the same for all of them. X is independent variable; Params is vector of parameters, such that

Params[1] = σ, Params[2]..Params[N+1]: scaling factors (in other words, probabilities of all gaussians); Params[N+2..2*N+2]: μ_i, $0 \le i \le N$, where N is number of gaussians. This function is used internally as RegFunc for fitting the sum of gaussians, but may be used also for evaluating a ready model.

SumGaussiansS0

Declaration `function SumGaussiansS0(X:Float; Params:TVector):float;`

Description Evaluates sum of gaussians where σ_0 for the first gaussian defined as $g(\mu_0, \sigma_0)$ (See Equation 12.1) may be different from σ for gaussians $[g_1..g_N]$ and is fitted separately. X is independent variable; Params is vector of parameters such that:

Params[1] = σ_0, Params[2] = σ, its value is shared among gaussians $g_1..g_N$; Params[3]..Params[N+2] are ScF_i (scaling factors); Params[N+3]..[2*N+2] are μ_i for N gaussians.

SetGaussFit

Declaration `procedure SetGaussFit(ANumberOfGaussians:integer; AUseSigma0, AFitMeans: boolean);`

Description Set model parameters: ANumberOfGaussians: How many gaussians form the distribution; AUseSigma0 : if Sigma0 may be different from others; AFitMeans: if means of all gaussians are fitted or they are fixed and only sigmas and scale factors (which give probabilities for every gaussian) are fitted. This procedure must be called before `SumGaussFit` .

SumGaussFit

Declaration `procedure SumGaussFit(var AMathExpect: TVector; var ASigma, ASigma0:Float; var ScFs : TVector; const AXV, AYV:TVector; Observ:integer);`

Description Actual fit of the model. AMathExpect: as input, guess values for means; as output, fitted means; ASigma, ASigma0: guessed and then found Sigma for all gaussians and, if needed, for first one; AXV, AYV: experimental data for X and for Y (observed probability distribution density); Observ: number of observations (High bound of AXV and AYV).

12.4 Unit uGaussf

LMath

12.4.1 Description

This unit is largely similar to `uGauss`, but in this model difference $\mu_{i+1} - \mu_i = \delta_\mu$ is constant. Such distributions arise often in patch-clamp experiments. Consequently, fitted are $[\sigma_0, \sigma, \mu_0, \delta_\mu, SCF_i]$.

12.4.2 Functions and Procedures

SumGaussiansF

Declaration `function SumGaussiansF(X:Float; Params:TVector):float;`

Description X is independent variable; Params is vector of parameters: Params[1] is σ, Params[2]..Params[N+1] are ScF_i (scaling factors); Params[N+2] is μ_0; Params[N+3] is δ_μ. This function is used as RegFunc for fitting of sum of gaussian.

SumGaussiansFS0

Declaration `function SumGaussiansFS0(X:Float; Params:TVector):float;`

Description Similar to `SumGaussiansS0`, this function evaluates multigaussian distribution with a separate σ_0. X is independent variable; Params is vector of parameters: Params[1] is σ_0, Params[2] is σ, Params[3]..Params[N+2] are ScF_i (scaling factors). Params[N+3] is μ_0, Params[N+4] is δ_μ.

SetGaussFitF

Declaration `procedure SetGaussFitF(ANumberOfGaussians:integer; AUseSigma0:boolean);`

Description Sets parameters of the model. `ANumberOfGaussians` is the number of gaussians which form the multigaussian distribution; `AUseSigma0` defines if σ_0 can be different from σ for other distributions. This procedure must be called before `DeltaFitGauss`.

DeltaFitGaussians

Declaration `procedure DeltaFitGaussians(var ASigma, ASigma0, ADelta, AMu0: Float; var ScFs: TVector; const AXV, AYV: TVector; Observ: integer);`

Description This procedure executes actual fit of the multigaussian model with the costant δ_μ. `ASigma, ASigma0, ADelta, AMu0` before call must contain guess values for respective parameters of the model; after call, the found refined values. `ScFs[1..N]` is vector of scaling factors, guess values before the call and refined values upon return. `AXV[1..Observ]` and `AVY[1..Observ]` are vectors of independent variable (X) and corresponding distribution density (Y) in the observed histogram; `Observ` is number of observations, or bins in the histogram.

12.5 Unit ugoldman

LMath

12.5.1 Description

This unit defines and fits Goldman-Hodgkin-Katz equation for current:

$$I = P\frac{z^2F^2V_m}{RT} \cdot \frac{C_i - C_e\,\mathrm{e}^{\frac{-zFV_m}{RT}}}{1-\mathrm{e}^{\frac{-zFV_m}{RT}}} \tag{12.2}$$

where I is current (Amp) or current density (Amp/m^2); P is specific permeability (Mol/m^2) if current density is calculated or permeability (Mol/m^3) if current is calculated. z is valence of permeated ion; F is Faraday constant; R is gas constant; T is absolute temperature, K; V_m is transmembrane voltage, V; C_i and C_e are intracellular and extracellular concentrations of permeated ion, respectively.

Note, that at a call to function, temperature is expressed in °C and voltage in mV; all conversions are done by the function itself.

12.5.2 Functions and Procedures

GHK

Declaration `function GHK(P, z, Cin, Cout, Vm, TC: float):float;`

Description Returns current, Amp, calculated according to Goldman-Hodgkin-Katz equation (12.2) at a given transmembrane potential, mV.

Parameters: P: permeability constant. Classically, current density (Amp/m^2) is calculated and P is in m/s. If we are interested in absolute value of current and not density, P is m^3/s. z is ion charge (-1 for Cl$^-$; 2 for Ca^{++} etc). It is float since for non-selective channels apparent valence of permeated ion may be non-integer. Cin is intracellular concentration of ion, Mol/m^3 or mM/l; $Cout$ is extracellular concentration. Vm is transmembrane voltage, mV. TC is temperature, °C.

FitGHK

Declaration `procedure FitGHK(CinFixed : boolean; az, aCout, aTC : float; var Cin, P : float; Voltages, Currents : TVector; Lb, Ub:integer);`

Description Fits data with Goldman-Hodgkin-Katz equation.

Input parameters: $CinFixed$: flag that intracellular concentration is known and fixed; only permeability constant must be fitted. If false, both C_{in} and P are fitted. az, $aCout$ are valence and extracellular concentration of permeated ion; aTC is temperature, °C; Cin and P are initial (guess) values for intracellular concentration and permeability; $Voltages$ and $Currents$ are vectors of observed data; Lb and Ub are array bounds.

Output: Fitted permeability is returned in P and, if not $CinFixed$, then fitted intracellular concentration in Cin, otherwise Cin keeps its initial value.

GOutMax

Declaration `function GOutMax(P,TC,Cin,Cout,z:float):float;`

Description When $V_m \to \pm\infty$, Goldman-Hodgkin-Katz voltage-current dependance tends to linear and slope conductance (G_s)tends to constant:

$$\lim_{V_m \to +\infty} G_s = GOutMax$$

GInMax

Declaration `function GInMax(P,TC,Cin,Cout,z:float):float;`

Description Similar to GoutMax, but finds limit for $-\infty$:

$$\lim_{V_m \to -\infty} G_s = GInMax$$

ERev

Declaration `function ERev(CIn, COut, z, TC:float):float;`

Description Returns reverse potential by Nernst equation, mV. TC : temperature, °C.

Intracellular

Declaration `function Intracellular(Cout, z, TC, ERev:float):float;`

Description Returns intracellular concentration from C_{out}, valence, temperature (°C), E_{Rev} (mV)

GSlope

Declaration `function GSlope(Cin,Cout,z,TC,Vm,P:float):float;`

Description Returns slope conductance at any V_m.

PfromSlope

Declaration `function PfromSlope(dI,dV,z,C,TC:float):float;`

Description Returns permeability for linear voltage-current relations. dI and dV are current and corresponding voltage.

12.6 Unit uhillfit

12.6.1 Description

This unit fits the Hill equation:

$$y = A + \frac{B - A}{1 + (K/x)^n}$$

$n > 0$ for an increasing curve;
$n < 0$ for a decreasing curve.

12.6.2 Functions and Procedures

HillFit

Declaration `procedure HillFit(X, Y : TVector; Lb, Ub : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of model. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; ConsTerm = presence of constant term A; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that:

B[0] = A, B[1] = B, B[2] = K, B[3] = n;

V = inverse matrix, [0..3,0..3].

WHillFit

Declaration `procedure WHillFit(X, Y, S : TVector; Lb, Ub : Integer; ConsTerm : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of model. Additional input parameter: S = standard deviations of observations.

HillFit_Func

Declaration `function HillFit_Func(X : Float; B : TVector) : Float;`

Description Computes the regression function at point X. B is the vector of parameters.

12.7 Unit umichfit

12.7.1 Description

This unit fits the Michaelis equation:

$$y = \frac{Y_{max} X}{K_m + X}$$

12.7.2 Functions and Procedures

MichFit

Declaration `procedure MichFit(X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of model. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; MaxIter = max. number of iterations; Tol = tolerance on parameters. Output parameters: B = regression parameters, such that:

B[0] = Y_{max}, B[1] = K_m;

V = inverse matrix, [0..1,0..1].

WMichFit

Declaration `procedure WMichFit(X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of model. Additional input parameter: S = standard deviations of observations.

MichFit_Func

Declaration `function MichFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

12.8 Unit umintfit

12.8.1 Description

This unit fits the Integrated Michaelis-Menten equation:

$$y = S_0 - K_m \cdot \mathrm{W}\left(\frac{S_0}{K_m} \cdot \exp\left(\frac{S_0 - k_{cat}\,E_0\,t}{K_m}\right)\right)$$

y = product concentration at time t; S_0 = initial substrate concentration; K_m = Michaelis constant; k_{cat} = catalytic constant; E_0 = total enzyme concentration.

W is Lambert's function (reciprocal of x * exp(x)).

The independent variable x may be:

- $t \implies$ fitted parameters: S_0 (optional), K_m, $V_{max} = k_{cat}\,E_0$;
- $S_0 \implies$ fitted parameters: K_m, $(Vmax \cdot t)$;
- $E_0 \implies$ fitted parameters: S_0 (optional), K_m, $(k_{cat} \cdot t)$.

Optional parameter is placed in `B[0]`, others in following elements of `B` array.

12.8.2 Functions and Procedures

MintFit

Declaration `procedure MintFit(X, Y : TVector; Lb, Ub : Integer; MintVar : TMintVar; Fit_S0 : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Unweighted fit of model. Input parameters: X, Y = point coordinates; Lb, Ub = array bounds; MintVar = independant variable, possible values: (Var_T, Var_S, Var_E). Fit_S0 indicates if S_0 must be fitted (for Var_T or Var_E only); MaxIter = max. number of iterations; Tol = tolerance on parameters; B[0] = initial value of S0. Output parameters: B = regression parameters; V = inverse matrix, [0..2, 0..2].

WMintFit

Declaration `procedure WMintFit(X, Y, S : TVector; Lb, Ub : Integer; MintVar : TMintVar; Fit_S0 : Boolean; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

Description Weighted fit of model. Additional input parameter: S = standard deviations of observations.

MintFit_Func

Declaration `function MintFit_Func(X : Float; B : TVector) : Float;`

Description Returns the value of the regression function at point X.

12.9 Unit upkfit

12.9.1 Description

This unit fits the acid/base titration function :

$$y = A + \frac{B - A}{1 + 10^{(pK_a - x)}}$$

where x is pH, y is some property (e.g. absorbance) which depends on the ratio of the acidic and basic forms of the compound. A is the property for the pure acidic form, B is the property for the pure basic form. pK_a is the acidity constant.

12.9.2 Functions and Procedures

PKFit

Declaration `procedure PKFit(X, Y : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

WPKFit

Declaration `procedure WPKFit(X, Y, S : TVector; Lb, Ub : Integer; MaxIter : Integer; Tol : Float; B : TVector; V : TMatrix);`

PKFit_Func

Declaration `function PKFit_Func(X : Float; B : TVector) : Float;`

Description Computes the regression function at point X. B is the vector of parameters, such that : B[0] = A; B[1] = B; B[2] = pKa.

12.10 Unit uModels

12.10.1 Description

Sets and returns properties of regression models.

12.10.2 Types

TRegType

Declaration `TRegType = (...);`

Description

Values `REG_LIN` Linear

`REG_MULT` Multiple linear

`REG_POL` Polynom

`REG_FRAC` Rational fraction

`REG_EXPO` Sum of exponentials

`REG_IEXPO` Increasing exponential

`REG_EXLIN` Exponential + linear

`REG_LOGIS` Logistic

`REG_POWER` Power

`REG_GAMMA` Gamma distribution

`REG_MICH` Michaelis equation

`REG_MINT` Integrated Michaelis equation

`REG_HILL` Hill equation

`REG_PK` Acid-base titration curve

`REG_EVAL` User-defined function

TModel

Declaration
```
TModel = record
case RegType : TRegType of
REG_MULT   : (Mult_ConsTerm : Boolean; Nvar : Integer);
REG_POL    : (Deg : Integer);
REG_FRAC   : (Frac_ConsTerm : Boolean; Deg1, Deg2 : Integer);
REG_EXPO   : (Expo_ConsTerm : Boolean; Nexp : Integer);
REG_IEXPO  : (IExpo_ConsTerm : Boolean);
REG_LOGIS  : (Logis_ConsTerm, Logis_General : Boolean);
REG_MINT   : (MintVar : TMintVar; Fit_S0 : Boolean);
REG_Hill   : (Hill_ConsTerm : Boolean);
end;
```

Description This record defines type and parameters of a model to be fitted with a call of FitModel or WFitModel. `RegType` is `TRegType` introduced above; other fields correspond to call parameters of specific functions and can be found in their descriptions.

12.10.3 Functions and Procedures

FirstParam

Declaration `function FirstParam(Model : TModel) : Integer;`

Description Returns the index of the first regression parameter.

LastParam

Declaration `function LastParam(Model : TModel) : Integer;`

Description Returns the index of the last regression parameter.

FuncName

Declaration `function FuncName(Model : TModel) : String;`

Description Returns the name (formula) of the regression function.

ParamName

Declaration `function ParamName(Model : TModel; I : Integer) : String;`

Description Returns the name of the I-th parameter

RegFunc

Declaration `function RegFunc(Model : TModel; X : Float; B : TVector) : Float;`

Description Returns the regression function.

FitModel

Declaration `procedure FitModel(Model : TModel; X, Y, Ycalc : TVector; U : TMatrix; Lb, Ub : Integer; MaxIter : Integer; Tol, SVDTol : Float; B : TVector; V : TMatrix; var Test : TRegTest);`

Description Unweighted fit of model. Input:
`Model`:`TModel`: type and parameters of the model to be fitted;
X,Y: point coordinates;
U: matrix of independent variables for multilinear regression;
Lb, Ub: bounds for X and Y arrays;
MaxIter: maximal number of iterations;
Tol, SVDTol: tolerance on regression parameters;

Output:
YCalc[Lb..Ub] contains predicted Y values for each X from X array. Before

call it must be allocated but does not require initialization;
B: vector of regression parameters, length depends on a model;
V: inverse matrix, dimentions depend on the model;
Test: results of goodness of fit test.

WFitModel

Declaration `procedure WFitModel(Model : TModel; X, Y, S : TVector; Ycalc : TVector; U : TMatrix; Lb, Ub : Integer; MaxIter : Integer; Tol, SVDTol : Float; B : TVector; V : TMatrix; var Test : TRegTest);`

Description Weighted fit of model. Additional input parameter S: vector of standard deviations, [Lb,Ub].

Chapter 13

Package lmMathUtil. Various utility functions.

13.1 Description

Package lmMathUtil includes several units which do not belong *sensu stricto* to the field of numeric analysis, but can be useful for scientific programming. Unit lmUnitsFormat allows to output values with units in conveniently formatted form using prefixes as pico-, nano- and so on. Unit lmSorting implements several sorting algorithms for arrays of Float and of TRealPoint; the latter ones may be sorted both for X and for Y; units uStrings and uWinStr define several handy functions over strings.

13.2 Unit uSearchTrees

LMath 0.5

13.2.1 Description

This unit defines object type TStringTreeNode as a named element of a binary search tree and implements a procedure of a search within it. Old-type object is used instead of class to save space. I don't want to use a huge `classes` unit in LMath.

13.2.2 Types and objects

TStringTreeNode

Declaration

```
TStringTreeNode = object
  Name : string; {Name of the object. Function Finds searches for it}
  Left : PStringTreeNode; {Link to left (lesser) element}
  Right: PStringTreeNode; {Link to right (greater) element}
  constructor Init(AName:string);
  destructor Done;
  function Find(AName:string; out Comparison:integer):PStringTreeNode;
end;
```

Methods

Init

Declaration constructor Init(AName:TString)

Description Creates the object, initializes `Name` field with `AName`, `Left` and `Right` with `nil`.

Done

Declaration destructor Done;

Description Disposes the item and all its children. To dispose a whole tree beginning with `TreeRoot:TStringTreeNode`, call `dispose(TreeRoot, done)` for its root.

Find

Declaration
```
function Find(AName:string;
                out Comparison:integer):PStringTreeNode;
```

Description Searches self and children for a member with `Name = AName`. returns either found item (`Comparison = 0` in this case) or, if the tree does not contain an item which meets condition, then returns an item where a new item with `AName` must be inserted. If `AName < Name` and, consequently, the new Item must be inserted as `Find.Left`, then `Comparison < 0`, if `AName > Name`, then `Comparison > 0`.

13.3 Unit uUnitsFormat

LMath

13.3.1 Description

This unit formats a value with exponent prefixes (milli, pico etc) such that value in output is in the range 1..1000 and adds provided string at the end. For example, FormatUnits(1.2E-12,S) will return "1.2 pS"

13.3.2 Functions and Procedures

FormatUnits

Declaration
```
function FormatUnits(Val:float; UnitsStr:string;
long:boolean=false):string;
```

Description Formats a value Val and SI units name UnitStr with SI decimal prefix such that numeric value in the output string is in [-999..999] range and corresponding prefix is used. E.g.: FormatUnits(12000, "Hz") returns "1.2 kHz". One letter prefix from "UnixArray" is used when `long = false` ('k'), long prefix from "UnitPrefixLong" ('kilo') otherwise.

FindPrefixForExponent

Declaration
```
function FindPrefixForExponent(E:integer;
long:boolean=false):string):string;
```

Description `E` is an integer from a set [-18, -15 ..., 18]. The function returns a corresponding prefix, one letter if `Long = False` ('a', 'f', 'p', ..., 'E'), or complete prefix ('atto','femto','pico', ..., 'Exa'). If supplied value is not in the range, string 'Too high or low value' is returned.

13.3.3 Constants

DefFormat

Declaration
```
DefFormat = '####0.000';
```

UnitExponents

Declaration
```
UnitExponents : array[0..12] of Integer = (-18, -15, -12, -9,
-6, -3, 0, 3, 6, 9, 12, 15, 18);
```

UnitFactors

Declaration `UnitFactors : array[0..12] of Float = (1E-18, 1E-15, 1E-12, 1E-9, 1E-6, 1E-3, 1, 1E3, 1E6, 1E9, 1E12, 1E15, 1E18);`

UnitPrefix

Declaration `UnitPrefix : array[0..12] of string = ('a','f','p','n','µ','m','','K','M','G','T','P','E');`

UnitPrefixLong

Declaration `UnitPrefixLong : array[0..12] of String = ('atto', 'femto', 'pico', 'nano', 'micro', 'milli','', 'Kilo', 'Mega', 'Giga', 'Tera', 'Peta', 'Exa');`

13.4 Unit usorting

LMath

13.4.1 Description

Quicksort, InsertSort and HeapSort algorithms are implemented for sorting of arrays of float, of TRealPoint for X and for TRealPoint for Y. In all these procedures, Vector or Points is an array to be sorted; Lb, Ub are low and upper bounds of the sorted array; if desc is true, the array is sorted in descending order, otherwise in ascending.

13.4.2 Functions and Procedures

QuickSort

Declaration `procedure QuickSort(Vector : TVector; Lb,Ub:integer; desc:boolean);`

QuickSortX

Declaration `procedure QuickSortX(Points : TRealPointVector; Lb,Ub:integer; desc:boolean);`

QuickSortY

Declaration `procedure QuickSortY(Points : TRealPointVector; Lb,Ub:integer; desc:boolean);`

InsertSort

Declaration `procedure InsertSort(Vector : TVector; Lb,Ub:integer; desc:boolean);`

InsertSortX

Declaration `procedure InsertSortX(Points : TRealPointVector; Lb,Ub:integer; desc:boolean);`

InsertSortY

Declaration `procedure InsertSortY(Points : TRealPointVector; Lb,Ub:integer; desc:boolean);`

Heapsort

Declaration `procedure Heapsort(Vector:TVector; Lb, Ub : integer; desc:boolean);`

HeapSortX

Declaration `procedure HeapSortX(Points:TRealPointVector; Lb, Ub : integer; desc:boolean);`

HeapSortY

Declaration `procedure HeapSortY(Points:TRealPointVector; Lb, Ub : integer; desc:boolean);`

13.5 Unit uVecUtils

LMath 0.5

13.5.1 Types

TMatCoords

Declaration

```
TMatCoords = record
Row, Col :integer;
end;
```

Description Record, representing coordinates of a matrix element.

tmCoords

Declaration `function tmCoords(ARow,ACol:integer):TMatCoords;`

Description Constructor of TMatCoords from two integers.

13.5.2 Procedures and Functions

Apply

Declaration

```
procedure Apply(V:TVector; Lb, Ub: integer; Func:TFunc);

procedure Apply(M:TMatrix; LRow, URow, LCol, UCol: integer; Func:TFunc);

procedure Apply(V:TIntVector; Lb, Ub: integer; Func:TIntFunc);

procedure Apply(M:TIntMatrix; LRow, URow, LCol, UCol: integer; Func:TIntFunc);

procedure Apply(V:TVector; Mask:TIntVector; MaskLb:integer; Func:TFunc);

procedure Apply(V:TIntVector; Mask:TIntVector; MaskLb:integer; Func:TIntFunc);
```

`procedure Apply(var V:array of Float; Func:TFunc);`

`procedure Apply(var V:array of Integer; Func:TIntFunc);`

Description These procedures apply `TFunc` or `TIntFunc` to every element of `V` or `M` in the slice defined by `Lb` and `Ub` variables. Last two versions are "masked" functions which apply the functions only to elements whose indices are included into `Mask` array. For example, if `Mask = (3,5,7)` then the function will be applied only to `V[3],V[5] and V[7]`. The `Mask` array may be formed for example with the help of `SelectElements` function. Last two versions apply the function to every element of open array. At a function call, subarray may be passed as a parameter.

ApplyRecursive

Declaration function ApplyRecursive(InitValues:array of Float; Lb, Ub : integer; Func:TFloatArrayFunc; Ziel:TVector = nil):TVector;
function ApplyRecursive(InitValues:array of Integer; Lb, Ub : integer; Func:TIntArrayIntFunc; Ziel:TIntVector = nil):TIntVector;

Description `Function ApplyRecursive` is used to initialize an array with a recursive sequence, for example, Fibonacci numbers. `Ziel` is the vector where result is placed. If at the call it is `nil`, `Ziel` is allocated by the function. `InitValues` defines a "seed" of the sequence with some length L. First L elements of `Ziel`, beginning from `Lb` are filled with `InitValues`; `Ziel[element` $Lb + L + 1$`]` is equal to `Func(InitValues)`, `Ziel[Lb+1+2]` will be equal to Func(Ziel[Lb+1..Lb+L+1]) and so on. For example, to create an array containing first 20 numbers of Fibonacci, write:

```
uses uTypes, uVecUtils;
type
  Fibonacci : TIntVector;

function FibFunc(InitArr:array of Integer):integer;
begin
  Result := InitArr[0] + initArr[1];
end;

Fibonacci := ApplyRecursive(FibFunc, [0,1],0,19,nil);
```

CompVec

Declaration `function CompVec(X, Xref : TVector; Lb, Ub : Integer; Tol : Float) : Boolean;`

`function CompVec(constref X, Xref : array of float; Tol : Float) : Boolean;`

Description Checks if every component of vector X is within a fraction Tol of the corresponding component of the reference vector Xref. In this case, the function returns True, otherwise it returns False. Subarray can be passed to a version with open array. This latter function has all functionality as a version with TVector, but has more simple calling convention and is more flexible, allowing to use both dynamic and static arrays and subarrays. Hence, we suggest that it is used in all new code with the old version left for backward compatibility.

Any

Declaration `function Any(Vector : TVector; Lb, Ub : integer; Test : TTestFunc) : boolean;`

`function Any(M : TMatrix; LRow, URow, LCol, UCol : integer; Test : TTestFunc) : boolean;`

`function Any(Vector : TIntVector; Lb, Ub : integer; Test : TIntTestFunc) : boolean;`

`function Any(M : TIntMatrix; LRow, URow, LCol, UCol : integer; Test : TIntTestFunc) : boolean;`

`function Any(constref Vector:array of Float; Test:TTestFunc):boolean; overload;`

`function Any(constref Vector:array of integer; Test:TIntTestFunc):boolean; overload;`

Description Applies `Test` function of type TTestFunc to every element in Vector[Lb..Ub] or M[LCol..UCol,LRow..URow], or to every element of open array and returns `true` if for any of them `Test` returns `true`.

FirstElement

Declaration `function FirstElement(Vector : TVector; Lb, Ub : integer; Ref : float; Comparator : TComparator) : integer;`

`function FirstElement(M : TMatrix; LRow, URow, LCol, UCol : integer; Ref : float; Comparator : TComparator) : TMatCoords;`

`function FirstElement(Vector : TVector; Lb, Ub : integer; Test : TTestFunc) : integer;`

`function FirstElement(M : TMatrix; LRow, URow, LCol, UCol : integer; Test : TTestFunc) : TMatCoords;`

`function FirstElement(Vector : TIntVector; Lb, Ub : integer; Ref : integer; Comparator : TIntComparator) : integer;`

`function FirstElement(M : TIntMatrix; LRow, URow, LCol, UCol : integer; Ref : integer; Comparator : TIntComparator) : TMatCoords;`

`function FirstElement(Vector : TVector; Lb, Ub : integer; Ref : float; CompType : TCompOperator) : integer;`

`function FirstElement(M : TMatrix; LRow, URow, LCol, UCol : integer; Ref : float; CompType : TCompOperator) : TMatCoords;`

```
function FirstElement(Vector : TIntVector; Lb, Ub : integer;
Ref : integer; CompType : TCompOperator) : integer;
```

```
function FirstElement(M : TIntMatrix; LRow, URow, LCol, UCol :
integer; Ref : integer; CompType:TCompOperator) :
TMatCoords;
```

Description `FirstElement` tests every element of an array in a slice defined by `Lb` and `Ub`, or by `LRow, URow, LCol, UCol` and returns index (or indices, as tmCoords, when applied to a matrix) of the first element which meets a condition. There are three ways to set the condition. First, define and pass a test function of type TTestFunc; second, pass a reference value and a comparator function, type TComparator. FirstElement passes to the comparator an array element as a first parameter and `Ref` as a second. Third way is to pass a reference value and type of comparison as a parameter `CompType` of type TCompOperator. FirstElement will return an index (or indices, for matrix) of an element which compares to `Ref` according to `CompType`.

MaxLoc, MinLoc

Declaration
```
function MaxLoc(Vector:TVector; Lb, Ub:integer):integer;
```

```
function MaxLoc(M:TMatrix;
LRow,URow,LCol,UCol:integer):TMatCoords;
```

```
function MaxLoc(Vector:TIntVector; Lb, Ub:integer):integer;
```

```
function MaxLoc(M:TIntMatrix;
LRow,URow,LCol,UCol:integer):TMatCoords;
```

```
function MinLoc(Vector:TVector; Lb, Ub:integer):integer;
```

```
function MinLoc(M:TMatrix;
LRow,URow,LCol,UCol:integer):TMatCoords;
```

```
function MinLoc(Vector:TIntVector; Lb, Ub:integer):integer;
```

```
function MinLoc(M:TIntMatrix;
LRow,URow,LCol,UCol:integer):TMatCoords;
```

Description Functions `MaxLoc` and `MinLoc` return the index or indices, in latter case as tmCoords record, of a maximal or minimal, respectively, element of a slice of array where array is `M` or `Vector` and slice is defined by `Lb` and `Ub` or by `LRow, URow, LCol, UCol` parameters.

Seq

Declaration
```
function Seq(Lb, Ub : integer; first, increment:Float;
Vector:TVector = nil):TVector;
```

```
function ISeq(Lb, Ub : integer; first, increment:integer;
Vector:TIntVector = nil):TIntVector;
```

Description Generates arithmetic progression,
`V[Lb] = First;`
`V[Lb+N] = First + N * Increment`
up to `V[Ub]`.

SelElements

Declaration function SelElements(Vector:TVector; Lb, Ub, ResLb : integer; Ref: float; CompType:TCompOperator):TIntVector;

function SelElements(Vector:TVector; Lb, Ub, ResLb : integer; Ref:float; Comparator:TComparator):TIntVector;

function SelElements(Vector:TIntVector; Lb, Ub, ResLb : integer; Ref: Integer; CompType:TCompOperator):TIntVector;

function SelElements(Vector:TIntVector; Lb, Ub, ResLb : integer; Ref:Integer; Comparator:TIntComparator):TIntVector;

Description Functions of `SelElements` family select elements from a source array, which can be of `TVector` or `TIntVector` type and place their indices into a result array of `TIntVector` type. Similar to FirstElement, selection criteria may be defined with a reference value and type of comparison (TCompOperator type) or with a reference value and a comparator function of TIntComparator or TComparator type. Selected indices are copied to Result array beginning from ResLb. Resulting “mask” array can be used with masked versions of Apply procedure. Besides, all corresponding elements can be extracted in a new array using ExtractElements function (see below).

ExtractElements

Declaration function ExtractElements(Vector:TVector; Mask:TIntVector; Lb:integer):TVector;

Description Function `ExtractElements` allows to extract selected elements into a separate TVector. Elements of `Vector` whose indices are contained in `Mask` array are copied into `Result` beginning from `Lb`.

13.6 uVecFunc

LMath 0.5

13.6.1 Description

Unit `uVecFunc` defines several standard functions over elements of arrays (vectors and matrices).

13.6.2 Functions

VecAbs

Declaration procedure VecAbs(V : TVector; Lb, Ub : integer);

procedure VecAbs(V : TIntVector; Lb, Ub : integer);

Description Calculates absolute value over all elements of a vector or integer vector from V[Lb] to V[Ub].

MatAbs

Declaration procedure MatAbs(M : TMatrix; Lb1, Ub1, Lb2, Ub2 : integer);

procedure MatAbs(M : TIntMatrix; Lb1, Ub1, Lb2, Ub2 : integer);

Description Calculates absolute value of all elements in M[Lb1,Lb2] to M[Lb2,Ub2].

VecSqr

Declaration procedure VecSqr(V : TVector; Lb, Ub : integer);

procedure VecSqr(V : TIntVector; Lb, Ub : integer);

Description Calculates square of all elements in of a vector or integer vector from V[Lb] to V[Ub].

MatSqr

Declaration procedure MatSqr(M : TMatrix; Lb1, Ub1, Lb2, Ub2 : integer);

procedure MatSqr(M : TIntMatrix; Lb1, Ub1, Lb2, Ub2 : integer);

Description Calculates square of all elements in M[Lb1,Lb2] to M[Lb2,Ub2].

VecSqrt

Declaration procedure VecSqrt(V : TVector; Lb, Ub : integer);

Description Calculates square root of all elements in of a vector from V[Lb] to V[Ub].

MatSqrt

Declaration procedure MatSqrt(V : TVector; Lb, Ub : integer);

Description Calculates square root of all elements in M[Lb1,Lb2] to M[Lb2,Ub2].

13.7 Unit uVectorHelpers

LMath

13.7.1 Description

This unit defines type helpers for TVector and TIntVector.

13.7.2 Types

TVectorHelper

Declaration
```
TVectorHelper = type helper for TVector
     procedure Insert(value:Float; index:integer);
     procedure Remove(index:integer);
     procedure Swap(ind1,ind2:integer);
     procedure Clear;
     procedure Fill(Lb, Ub : integer; Val:Float);
     procedure FillWithArr(Lb : integer; Vals:array of Float);
     procedure Sort(Descending:boolean);
     procedure InsertFrom(
      Source:TVector; Lb, Ub: integer; ind:integer);
     procedure InsertFrom(constref source: array of float;
        ind: integer); overload;
     function ToString(Index:integer):string;
     function ToStrings(Dest:TStrings; First, Last:integer;
     Indices:boolean; Delimiter: char):integer;
end;
```

TIntVectorHelper

Declaration
```
TIntVectorHelper = type helper for TIntVector
     procedure Insert(value:Integer; index:integer);
     procedure Remove(index:integer);
     procedure Swap(ind1,ind2:integer);
     procedure Clear;
     procedure Fill(St, En : integer; Val:Integer);
     procedure FillWithArr(Lb : integer; Vals:array of Integer);
     procedure InsertFrom(
      Source:TIntVector; Lb, Ub: integer; ind:integer);
     procedure InsertFrom(constref source: array of integer;
        ind: integer); overload;
     function ToString(Index:integer):string;
     function ToStrings(Dest:TStrings; First, Last:integer;
     Indices:boolean; Delimiter: char):integer;
end;
```

13.7.3 Methods

Most of the methods of TVectorHelper and TVIntVectorHelper are identical with the exception of array element type, the following descriptions are relevant for both. The only exception is `sort` which is defined for TVectorHelper only.

Insert

Declaration `procedure Insert(value:Float; index:integer);`

`procedure Insert(value:Integer; index:integer);`

Description Inserts `value` in the position `index`. Following elements are shifted to the right. Self[High(self)] is lost.

Remove

Declaration `procedure Remove(index:integer);`

Description Element in the position `index` is removed; succeeding elements are shifted to the left. Self[High(self)] is set to 0.

Swap

Declaration `procedure Swap(ind1,ind2:integer);`

Description Swaps elements in positions ind1, ind2. No range check is performed.

Declaration `procedure Clear;`

Description Fills `Self` with zeros.

Fill

Declaration `procedure Fill(Lb, Ub : integer; Val:Float);`

`procedure Fill(Lb, Ub : integer; Val:integer);`

Description Sets all elements from `self[Lb]` to `self[Ub]` to `Val`. If `Self` is empty, it is allocated, otherwise it id filled to min(Ub,high(self)). Length of `Self` remains unchanged.

FillWithArr

Declaration `procedure FillWithArr(Lb : integer; Vals:array of Float);`

`procedure FillWithArr(Lb : integer; Vals:array of Integer);`

Description Copies all elements from `Vals` into `self` beginning from `self[Lb]`. If `Self` is empty, it is allocated, otherwise filled up to
min(Lb+length(Vals),High(Self)).
This procedure is designed mainly for convenient initialization of arrays in statements like
MyNewVector.FillWithArr(1,[3,4,5,9,12]);

InsertFrom

Declaration `procedure InsertFrom(Source:TVector; Lb, Ub: integer; ind:integer);`

`procedure InsertFrom(Source:TIntVector; Lb, Ub: integer; ind:integer);`

`procedure InsertFrom(constref source: array of float; ind: integer); overload;`

`procedure InsertFrom(constref source: array of integer; ind: integer); overload;`

Description This method exists in two forms: older one where `Sourse` is TVector, kept for backward compatibility, and a new one where sourse is an open array, which gives greater flexibility.

Elements of Source are inserted into `Self` beginning from `Self[ind]`. Elements of target array beginning from `Ind` are shifted to the right. Length of target array remains unchanged, last elements of it are lost. In the older form, elements beginning from `Source[Lb]` to `Source[Ub]` are insterted, in the new one, the whole array is inserted. If you need to insert a subarray, you may define it at a calling the method:
`MyArr.InsertFrom(MyOtherArr[5..12])`.
If length of passed array is too long, the array is truncated respectively, such that length of a target array remains unchanged: if $Ind + Ub - Lb > High(Self)$, only $High(Self) - Ind$ elements are inserted.

Sort (TVectorHelper only)

Declaration procedure Sort(Descending:boolean);

Description Sorts `Self` using Heap sort algorithm, `Descending` defines order. Note that Quick sort and insert sort algorithms are umplemented in uSorting unit.

ToString

Declaration function ToString(Index:integer):string;

Description Returns FloatStr(Self[Index]) or IntStr(Self[Index]). Output format is defined with uStrings.SetFormat function.

ToStrings

Declaration function ToStrings(Dest : TStrings; First, Last: integer; Indices: boolean; Delimiter: char):integer;

Description Sends string representation of the subarray `self[Lb]` to `self[Ub]` to `Dest`. If `Indices` is true, every string (for TVectorHelper) is formed as `IntStr(Index)+' '+Delimiter+FloatToStr(self[Index])`.

13.8 Unit uVecFileUtils

LMath 0.5

13.8.1 Desccription

This unit defines four routines for saving `TMatrix` or `TVector` to a file or reading from a file.

13.8.2 Functions and Procedures

SaveVecToText

Declaration `procedure SaveVecToText(FileName:string; V:TVector; Lb, Ub:integer);`

Description Saves `V:TVector` to a file with name `FileName`, beginning from `V[Lb]` up to `V[Ub]` as a column of numbers. If $Ub > High(V)$, the to the end is saved.

Errors: if $Lb > Ub$ or $Lb < 0$, MatErrDim is set and no file written. If an error occurs at an opening or writing the file, lmFileError is set with additional information in a text message which can be retrieved with `MathErrMessage` function.

LoadVecFromText

Declaration `function LoadVecFromText(FileName:string; Lb:integer; out HighLoaded : integer) : TVector;`

Description This is a counterpart to `SaveVecToText`. This function reads a column of numbers from a file into `TVector`.
Input: `FileName`, name of a file to be read; `Lb`, index, from which the file contents is placed into `TVector`, typically 0 or 1.
The function counts number of lines in the file (N) and allocates `TVector` with the length $Lb + N$ which is returned by the function. Further, it reads the values from the file. If a line does not contain valid number, it is silently skipped. Number of actually assigned values is returned in `HighLoaded`. Hence, the last meaningful element is $Lb + HighLoad$.

Errors: if $Lb < 0$, MatErrDim is set `nil` is returned. If an error occurs at an opening or writing the file, lmFileError is set with additional information in a text message which can be retrieved with `MathErrMessage` function. If nothing was read before an error, `nil is returned`, otherwise partially filled vector returned with the number of actually read data in `HighLoaded`.

SaveMatToText

Declaration `procedure SaveMatToText(FileName: string; M: TMatrix; delimiter: char; FirstCol, LastCol, FirstRow, LastRow : integer);`

Description Saves slice of a matrix `M[FirstCol..LastCol,FirstRow...LastRow]` into a file as a delimited text (for example, CSV).
Input: `FileName:string`, name of a file to be saved;`M:TMatrix`, a matrix to be saved; `delimiter:char` is a delimiter between values in a line of a file. It is not recommended to use a space as a delimiter, see LoadMatFromText for explanation. `FirstCol, LastCol, FirstRow, LastRow : integer` are border indices of a slice to be saved. If `LastCol` is higher than the length of rows or `LastRow` is bigger than length of columns, they are adjusted accordingly such that the rows and/or the columns are saved completely.

Errors: if indices given in `FirstRow, LastRow, FirstColumn, LastColumn` are impossible, MatErrDim is set and no file written. If an error occurs at an opening or writing the file, lmFileError is set with additional information in a text message which can be retrieved with `MathErrMessage` function.

LoadMatFromText

Declaration function LoadMatFromText(FileName: string; delimiter: char; Lb:integer; MD:Float): TMatrix;

Description Allocates and loads a matrix from text file. Number of rows allocated is equal to the number of lines in the file; number of columns is found as a maximal number of delimiters in a line plus 1. One potential problem with this approach is that if a delimiter is space char, and spaces are used for formatting and alignment, then too many elements are allocated. Therefore, it is not recommended to use space as a delimiter. However, if there is only one space between values, everything works fine.
Input: `FileName: string`, name of a file to be read. Delimiter: char, symbol which delimits values within a line of a file (and, correspondingly, in a row of a matrix). `Lb:integer`, first row which is used to load the values and first index in each row, typically 0 or 1. If $Lb \neq 0$, rows and columns with indices $< Lb$ remain empty (filled with 0). `MD:Float`, code for missed value.
If a substring between two delimiters in a line of the file cannot be parsed (for example, it is an empty line), corresponding position in a matrix is filled with `MD`. If a line contains no valid values, it is supposed to be title or subtitle and is silently skipped.
Example:
If there is a file `example.csv` with content:

```
V1; V2; V3; V4; V5
3.1; 5.6; 7.4;;2.3
5.8; 9.6; 11.1;
7.6; 4.5; 45.2; 3.9; 7.5
```

then call
`LoadMatFromText('example.csv',';',1,-10000);`
would produce following matrix:

ind	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	3.1	5.6	7.4	-10000	2.3
2	0	5.8	9.6	11.1	-10000	-10000
3	0	7.6	4.5	45.2	3.9	7.5

`Lb = 1` causes column and row with indices “0” to remain empty, the matrix is filled from indices “1”. Maximal number of delimiters ';' per line is 4, which defines 5 values per line. Empty string between 3rd and 4th delimiters in the second line is not a valid float, it is substituted with `MD = -10000`; missing values at the end of next line are also substituted with `MD`. First line contains no valid floats at all, it is skipped.

Errors: if `Lb` is negative, MatErrDim error is set; on file read failure, lmFileError is set.

13.9 Unit uCompVecUtils

LMath 0.6

13.9.1 Description

Unit uCompVectUtils defines several utility functions and procedures to make more convenient work with arrays of complex numbers, largely similar to what was done for arrays of `Float` in the unit `uVecUtils` (13.5).

13.9.2 Types

TComplexTestFunc

Declaration `TComplexTestFunc = function(Val:complex):boolean;`

Description General function for testing complex value for a condition

TComplexFunc

Declaration `TComplexFunc = function(Arg:Complex):complex;`

Description General complex function of real argument

TIntComplexFunc

Declaration `TIntComplexFunc = function(Arg:Integer):complex;`

Description Complex function of an integer argument. This type is intended mostly for the use with function Apply (13.9.3.10) where index of an element is passed to the function.

TComplexComparator

Declaration `TComplexComparator = function(Val, Ref:complex):boolean;`

Description General function for comparison of complex numbers. Functions `FirstElement` and `SelElements` pass array elements to `Val` and user-supplied `Ref` value to `Ref`.

13.9.3 Procedures and Functions

ExtractReal

Declaration `function ExtractReal(const CVec:array of complex):TVector;`

Description Extracts real components from all elements of `CVec` and puts them into result, which is allocated by the function.

ExtractImaginary

Declaration `function ExtractImaginary(const CVec:array of complex):TVector;`

Description Extracts imaginary components from all elements of `CVec` and puts them into result, which is allocated by the function.

CombineCompVec

Declaration `function CombineCompVec(const VecRe, VecIm:array of float):TCompVector;`

Description Opposite to `ExtractImaginary` and `ExtractReal`: takes two arrays of `Float` and combines them into `TCompVector`. Elements of `VecRe` become real components; elements of `VecIm` become imaginary. Lengths of `VecRe` and `VecIm` must be equal, otherwise `MaterrDim` error is set and `nil` is returned.

CMakePolar

Declaration `function CMakePolar(const V:array of complex):TCompVector;`

Description Converts all elements of array of complex `V` from rectangular to polar form and puts them into `TCompVector` which is returned as result.

CMakeRectangular

Declaration `function CMakeRectangular(const V:array of complex):TCompVector;`

Description Converts all elements of array of complex `V` from polar to rectangular form and puts them into `TCompVector` which is returned as result.

MaxReLoc

Declaration `function MaxReLoc(CVec: TCompVector; Lb, Ub: integer):integer;`

Description Returns position in `CVec:TCompVector)` beginning from `Lb` ending with `Ub` of an element with maximal value of real component. If $Lb > Ub$, `MatErrDim` error is set and -1 is returned.

MaxImLoc

Declaration `function MaxImLoc(CVec: TCompVector; Lb, Ub: integer):integer;`

Description Returns position in `CVec:TCompVector)` beginning from `Lb` ending with `Ub` of an element with maximal value of imaginary component. If $Lb > Ub$, `MatErrDim` error is set and -1 is returned.

MinReLoc

Declaration `function MinReLoc(CVec: TCompVector; Lb, Ub: integer):integer;`

Description Returns position in `CVec:TCompVector)` beginning from `Lb` ending with `Ub` of an element with minimal value of real component. If $Lb > Ub$, `MatErrDim` error is set and -1 is returned.

MinImLoc

Declaration `function MinImLoc(CVec: TCompVector; Lb, Ub: integer):integer;`

Description Returns position in `CVec:TCompVector)` beginning from `Lb` ending with `Ub` of an element with minimal value of imaginary component. If $Lb > Ub$, `MatErrDim` error is set and -1 is returned.

Apply

Declaration
```
procedure Apply(var V:array of Complex; Func:TComplexFunc);
; procedure Apply(var V:array of Complex;
Func:TIntComplexFunc);
procedure Apply(V:TCompVector; Mask:TIntVector;
MaskLb:integer; Func:TComplexFunc);
```

Description These functions apply functions passed as `Func` parameter to every element of array `V` and assign result of the function to this element. When `Func` is of `TComplexFunc` type (13.9.2.2), element itself is passed to the function, otherwise if `Func` is `TIntComplexFunc` (13.9.2.3), index of the element is passed to the function. Thus, `Apply` in this form is suitable for primary initialization of arrays of `complex`. Third version of `Apply` applies `Func` only to the elements whose indices are contained in the `Mask`, beginning from `MaskLb`. The mask array can be generated for example by call to `Selelements` (13.9.3.14) prior to using `Apply` function.

CompareCompVec

Declaration `function CompareCompVec(const X, Xref : array of Complex; Tol : Float) : Boolean;`

Description Returns `True` if both vectors have same length and all elements are equal to the `MachEp` (2.2.3.19)accuracy.

Any

Declaration `function Any(const Vector:array of Complex; Test:TComplexTestFunc):boolean;`

Description Calls function `Test` with every element of array `Vector`. Returns `True` if `Test` returns `True` for at least one of the elements, otherwize returns `False`.

FirstElement

Declaration `function FirstElement(Vector: TCompVector; Lb, Ub: integer; Ref: complex; Comparator: TComplexComparator): integer;`

Description Function `FirstElement` for `TCompVector` calls `Comparator` of `TComplexComparator` type (13.9.2.4) with each element of the vector, beginning from `Lb` ending with `Ub`. Elements of array are passed as `Val` parameter, user-supplied `Ref` value as a second parameter. Function `FirstElement` returns index of the first element for which `Comparator` returns `True`. If $Lb > Ub$, `MatErrDim` error is set and -1 is returned.

SelElements

Declaration `function SelElements(Vector:TCompVector; Lb, Ub, ResLb : integer; Ref:complex; Comparator:TComplexComparator):TIntVector;`

Description Function `SelElements is largely similar to FirstElement, but it does not stop after finding a first element which satisfies the condition of Comparator, but finds all such elements and places them into result of TInteger type, beginning from ResLb, which typically is 0 or 1. Later this array of indices may be used for example with ExtractElements (see below) or masked version of Apply (13.9.3.10) function.` If $Lb > Ub$, `MatErrDim` error is set and `nil` is returned.

ExtractElements

Declaration `function ExtractElements(Vector:TCompVector; Mask:TIntVector; Lb:integer):TCompVector;`

Description Function `ExtractElements` for `TCompVector` places all elements of input array `Vector` whose indicis are found in the `Mask`

ComplexSeq

Declaration function ComplexSeq(Lb, Ub : integer; FirstRe, FirstIm, IncrementRe, IncrementIm:Float; Vector:TCompVector = nil):TCompVector;

Description Function `ComplexSec` is intended for easy initialization of `TCompVector`. This function generates arithmetic progression

$$FirstRe + I\dot{I}ncrementRe$$

where I is consecutive number of progression element. This progression is placed into the real components of output vector beginning from Lb. Similarly, progression formed as

$$FirstIm + I\dot{I}ncrementIm$$

is placed into imaginary components of the same elements. If output array `Input` at a call is `nil`, it is allocated by `ComplexSec`, otherwise the user-supplied array is used. If $Lb > Ub$, `MatErrDim` error is set and `nil` is returned.

13.10 Unit uVecMatPrn

LMath

13.10.1 Description

This small unit introduces procedures for printout of a vector or matrix in a console, useful mostly for test and demonstration purposes. See program LinProgTest as an example of usage.

13.10.2 Constants

lmFmtStr

Declaration lmFmtStr : string = '%8.3f';

Description Default format for printout of a float point number, using `Format` function.

lmFmtStr

Declaration lmIntFmtStr : string = '%4d';

Description Default format for printout of an integer number, using `Format` function.

13.10.3 Variables

LB

Declaration LB : integer = 1;

Description Defines a lower bound of a matrix printout. Default value is 1 (Fortran inheritance). PrintVector do not use this variable, use subarrays at call instead.

13.10.4 Procedures And Functions

PrintVector

Declaration procedure PrintVector(V:array of Integer);
procedure PrintVector(V:array of Float);

Description Prints TVector or TIntVector on one line in a console window.

PrintMatrix

Declaration procedure PrintMatrix(A:TMatrix);

Description Prints TMatrix as a table in a console window.

13.11 Unit ustrings

13.11.1 Description

Pascal string routines

13.11.2 Functions and Procedures

LTrim

Declaration `function LTrim(S : String) : String;`

Description Removes leading blanks

RTrim

Declaration `function RTrim(S : String) : String;`

Description Removes trailing blanks

Trim

Declaration `function Trim(S : String) : String;`

Description Removes leading and trailing blanks

StrChar

Declaration `function StrChar(N : Byte; C : Char) : String;`

Description Returns a string made of character C repeated N times

RFill

Declaration `function RFill(S : String; L : Byte) : String;`

Description Completes string S with trailing blanks for a total length L

LFill

Declaration `function LFill(S : String; L : Byte) : String;`

Description Completes string S with leading blanks for a total length L

CFill

Declaration `function CFill(S : String; L : Byte) : String;`

Description Completes string S with leading blanks to center the string on a total length L

Replace

Declaration `function Replace(S : String; C1, C2 : Char) : String;`

Description Replaces in string S all the occurences of character C1 by character C2

Extract

Declaration `function Extract(S : String; var Index : Byte; Delim : Char) : String;`

Description Extracts a field from a string. Index is the position of the first character of the field. Delim is the character used to separate fields (e.g. blank, comma or tabulation). Blanks immediately following Delim are ignored. Index is updated to the position of the next field.

Parse

Declaration `procedure Parse(S : String; Delim : Char; Field : TStrVector; var N : Byte);`

Description Parses a string into its constitutive fields. Delim is the field separator. The number of fields is returned in N. The fields are returned in Field[0]..Field[N - 1]. Field must be dimensioned in the calling program.

SetFormat

Declaration `procedure SetFormat(NumLength, MaxDec : Integer; FloatPoint, NSZero : Boolean);`

Description Sets the numeric format NumLength = Length of numeric field MaxDec = Max. number of decimal places FloatPoint = True for floating point notation NSZero = True to write non significant zero's

FloatStr

Declaration `function FloatStr(X : Float) : String;`

Description Converts a real to a string according to the numeric format

IntStr

Declaration `function IntStr(N : LongInt) : String;`

Description Converts an integer to a string

CompStr

Declaration `function CompStr(Z : Complex) : String;`

Description Converts a complex number to a string

ArrayToStr

LMath

Declaration
```
function ArrayToString(A:array of float; ADelimiter:
string):string;
function ArrayToString(A:array of integer; ADelimiter:
string):string;
```

Description Outputs an array into a string, inserting `ADelimiter` between the elements.

ArrayToStrMD

LMath

Declaration `function ArrayToStringMD(A:array of float; ADelimiter: string; MissingCode:float):string;`
`function ArrayToStringMD(A:array of integer; ADelimiter: string; MissingCode:integer):string;`

Description Outputs an array with missing values, coded by `MissingCode` into a string, inserting `ADelimiter` between the elements and a space instead of `MissingCode`.

13.12 Unit uwinstr

13.12.1 Description

String routines for DELPHI

13.12.2 Functions and Procedures

StrDec

Declaration `function StrDec(S : String) : String;`

Description Replaces commas or decimal points by the decimal separator defined in SysUtils

IsNumeric

Declaration `function IsNumeric(var S : String; out X : Float) : Boolean;`

Description Replaces in string S the decimal comma by a point, tests if the resulting string represents a number. If so, returns this number in X

ReadNumFromEdit

Declaration `function ReadNumFromEdit(Edit : TEdit) : Float;`

Description Reads a floating point number from an Edit control

WriteNumToFile

Declaration `procedure WriteNumToFile(var F : Text; X : Float);`

Description Writes a floating point number in a text file, forcing the use of a decimal point

Chapter 14

Package lmPlotter: Plotting of Mathematics

14.1 Unit uhsvrgb

14.1.1 Description

HSV / RGB conversion.
Adapted from http://www.cs.rit.edu/ ncs/color/t_convert.html
R, G, B values are from 0 to 255 H = [0..360), S = [0..1], V = [0..1] if S = 0, then H is undefined.

14.1.2 Functions and Procedures

HSVtoRGB

Declaration `procedure HSVtoRGB(H, S, V : Float; var R, G, B : Byte);`

RGBtoHSV

Declaration `procedure RGBtoHSV(R, G, B : Byte; var H, S, V : Float);`

14.2 Unit uplot

14.2.1 Description

Plotting routines for BGI graphics (based on the Graph unit)

14.2.2 Functions and Procedures

InitGraphics

Declaration `function InitGraphics(Pilot, Mode : Integer; BGIPath : String) : Boolean;`

Description Enters graphic mode

SetWindow

Declaration `procedure SetWindow(X1, X2, Y1, Y2 : Integer; GraphBorder : Boolean);`

Description Sets the graphic window. X1, X2, Y1, Y2: Window coordinates in % of maximum. GraphBorder: Flag for drawing the window border.

SetOxScale

Declaration `procedure SetOxScale(Scale : TScale; OxMin, OxMax, OxStep : Float);`

Description Sets the scale on the Ox axis.

SetOyScale

Declaration `procedure SetOyScale(Scale : TScale; OyMin, OyMax, OyStep : Float);`

Description Sets the scale on the Oy axis.

GetOxScale

Declaration `procedure GetOxScale(var Scale : TScale; var OxMin, OxMax, OxStep : Float);`

Description Returns the scale on the Ox axis.

GetOyScale

Declaration `procedure GetOyScale(var Scale : TScale; var OyMin, OyMax, OyStep : Float);`

Description Returns the scale on the Oy axis.

SetGraphTitle

Declaration `procedure SetGraphTitle(Title : String);`

Description Sets the title for the graph.

SetOxTitle

Declaration `procedure SetOxTitle(Title : String);`

Description Sets the title for the Ox axis

SetOyTitle

Declaration `procedure SetOyTitle(Title : String);`

Description Sets the title for the Oy axis.

GetGraphTitle

Declaration `function GetGraphTitle : String;`

Description Returns the title for the graph

GetOxTitle

Declaration `function GetOxTitle : String;`

Description Returns the title for the Ox axis.

GetOyTitle

Declaration `function GetOyTitle : String;`

Description Returns the title for the Oy axis.

SetTitleFont

Declaration `procedure SetTitleFont(FontIndex, Width, Height : Integer);`

Description Sets the font for the main graph title.

SetOxFont

Declaration `procedure SetOxFont(FontIndex, Width, Height : Integer);`

Description Sets the font for the Ox axis (title and labels).

SetOyFont

Declaration `procedure SetOyFont(FontIndex, Width, Height : Integer);`

Description Sets the font for the Oy axis (title and labels).

SetLgdFont

Declaration `procedure SetLgdFont(FontIndex, Width, Height : Integer);`

Description Sets the font for the legends.

PlotOxAxis

Declaration `procedure PlotOxAxis;`

Description Plots the horizontal axis.

PlotOyAxis

Declaration `procedure PlotOyAxis;`

Description Plots the vertical axis.

PlotGrid

Declaration `procedure PlotGrid(Grid : TGrid);`

Description Plots a grid on the graph.

WriteGraphTitle

Declaration `procedure WriteGraphTitle;`

Description Writes the title of the graph.

SetClipping

Declaration `procedure SetClipping(Clip : Boolean);`

Description Determines whether drawings are clipped at the current viewport boundaries, according to the value of the Boolean parameter Clip.

SetMaxCurv

Declaration `function SetMaxCurv(NCurv : Byte) : Boolean;`

Description Sets the maximum number of curves. Returns False if the needed memory is not available.

SetPointParam

Declaration `procedure SetPointParam(CurvIndex, Symbol, Size, Color : Integer);`

Description Sets the point parameters for curve # CurvIndex.

SetLineParam

Declaration `procedure SetLineParam(CurvIndex, Style, Width, Color : Integer);`

Description Sets the line parameters for curve # CurvIndex.

SetCurvLegend

Declaration `procedure SetCurvLegend(CurvIndex : Integer; Legend : String);`

Description Sets the legend for curve # CurvIndex.

SetCurvStep

Declaration `procedure SetCurvStep(CurvIndex, Step : Integer);`

Description Sets the step for curve # CurvIndex.

GetMaxCurv

Declaration `function GetMaxCurv : Byte;`

Description Returns the maximum number of curves.

GetPointParam

Declaration `procedure GetPointParam( CurvIndex : Integer; var Symbol, Size, Color : Integer);`

Description Returns the point parameters for curve # CurvIndex.

GetLineParam

Declaration `procedure GetLineParam( CurvIndex : Integer; var Style, Width, Color : Integer);`

Description Returns the line parameters for curve # CurvIndex.

GetCurvLegend

Declaration `function GetCurvLegend(CurvIndex : Integer) : String;`

Description Returns the legend for curve # CurvIndex.

GetCurvStep

Declaration `function GetCurvStep(CurvIndex : Integer) : Integer;`

Description Returns the step for curve # CurvIndex.

PlotPoint

Declaration `procedure PlotPoint(Xp, Yp, CurvIndex : Integer);`

Description Plots a point on the screen Input parameters : Xp, Yp = point coordinates in pixels CurvIndex = index of curve parameters (Symbol, Size, Color).

PlotCurve

Declaration `procedure PlotCurve(X, Y : TVector; Lb, Ub, CurvIndex : Integer);`

Description Plots a curve Input parameters : X, Y = point coordinates Lb, Ub = indices of first and last points CurvIndex = index of curve parameters.

PlotCurveWithErrorBars

Declaration `procedure PlotCurveWithErrorBars(X, Y, S : TVector; Ns, Lb, Ub, CurvIndex : Integer);`

Description Plots a curve with error bars. Input parameters: X, Y = point coordinates; S = errors; Lb, Ub = indices of first and last points; CurvIndex = index of curve parameters.

PlotFunc

Declaration `procedure PlotFunc(Func : TFunc; X1, X2 : Float; CurvIndex : Integer);`

Description Plots a function.

Input parameters: Func = function to be plotted; X1, X2 = abscissae of 1st and last point to plot; CurvIndex = index of curve parameters (Width, Style, Color).

The function must be programmed as: function Func(X : Float) : Float;

WriteLegend

Declaration `procedure WriteLegend(NCurv : Integer; ShowPoints, ShowLines : Boolean);`

Description Writes legends for all curves.

NCurv: number of curves (1 to MaxCurv); ShowPoints: for displaying points; ShowLines: for displaying lines.

WriteLegendSelect

Declaration `procedure WriteLegendSelect(NSelect : Integer; Select : TIntVector; ShowPoints, ShowLines : Boolean);`

Description Writes legends for selected curves.

NSelect: number of selected curves; Select indices of selected curves.

ConRec

Declaration `procedure ConRec(Nx, Ny, Nc : Integer; X, Y, Z : TVector; F : TMatrix);`

Description Contour plot. Adapted from Paul Bourke, Byte, June 1987 http://paulbourke.net/papers/conrec/.

Input parameters: Nx, Ny = number of steps on Ox and Oy; Nc = number of contour levels; X[0..Nx], Y[0..Ny] = point coordinates; Z[0..(Nc - 1)] = contour levels in increasing order; F[0..Nx, 0..Ny] = function values, such that F[I,J] is the function value at (X[I], Y[J]).

Xpixel

Declaration `function Xpixel(X : Float) : Integer;`

Description Converts user abscissa X to screen coordinate.

Ypixel

Declaration `function Ypixel(Y : Float) : Integer;`

Description Converts user ordinate Y to screen coordinate.

Xuser

Declaration `function Xuser(X : Integer) : Float;`

Description Converts screen coordinate X to user abscissa.

Yuser

Declaration `function Yuser(Y : Integer) : Float;`

Description Converts screen coordinate Y to user ordinate.

LeaveGraphics

Declaration `procedure LeaveGraphics;`

Description Quits graphic mode.

14.3 Unit utexplot

14.3.1 Description

Plotting routines for LaTeX/PSTricks

14.3.2 Functions and Procedures

TeX_InitGraphics

Declaration `function TeX_InitGraphics(FileName : String; PgWidth, PgHeight : Integer; Header : Boolean) : Boolean;`

Description Initializes the LaTeX file.

FileName = Name of LaTeX file (e. g. 'figure.tex'); PgWidth, PgHeight = Page width and height in cm; Header = True to write the preamble in the file.

TeX_SetWindow

Declaration `procedure TeX_SetWindow(X1, X2, Y1, Y2 : Integer; GraphBorder : Boolean);`

Description Sets the graphic window.

X1, X2, Y1, Y2: Window coordinates in % of maximum; GraphBorder: Flag for drawing the window border.

TeX_LeaveGraphics

Declaration `procedure TeX_LeaveGraphics(Footer : Boolean);`

Description Close the LaTeX file.

Footer = Flag for writing the 'end of document' section.

TeX_SetOxScale

Declaration `procedure TeX_SetOxScale(Scale : TScale; OxMin, OxMax, OxStep : Float);`

Description Sets the scale on the Ox axis.

TeX_SetOyScale

Declaration `procedure TeX_SetOyScale(Scale : TScale; OyMin, OyMax, OyStep : Float);`

Description Sets the scale on the Oy axis

TeX_SetGraphTitle

Declaration `procedure TeX_SetGraphTitle(Title : String);`

Description Sets the title for the graph.

TeX_SetOxTitle

Declaration `procedure TeX_SetOxTitle(Title : String);`

Description Sets the title for the Ox axis.

TeX_SetOyTitle

Declaration `procedure TeX_SetOyTitle(Title : String);`

Description Sets the title for the Oy axis.

TeX_PlotOxAxis

Declaration `procedure TeX_PlotOxAxis;`

Description Plots the horizontal axis

TeX_PlotOyAxis

Declaration `procedure TeX_PlotOyAxis;`

Description Plots the vertical axis

TeX_PlotGrid

Declaration `procedure TeX_PlotGrid(Grid : TGrid);`

Description Plots a grid on the graph.

TeX_WriteGraphTitle

Declaration `procedure TeX_WriteGraphTitle;`

Description Writes the title of the graph.

TeX_SetMaxCurv

Declaration `function TeX_SetMaxCurv(NCurv : Byte) : Boolean;`

Description Sets the maximum number of curves and re-initializes their parameters.

TeX_SetPointParam

Declaration `procedure TeX_SetPointParam(CurvIndex, Symbol, Size : Integer);`

Description Sets the point parameters for curve # CurvIndex.

TeX_SetLineParam

Declaration `procedure TeX_SetLineParam(CurvIndex, Style : Integer; Width : Float; Smooth : Boolean);`

Description Sets the line parameters for curve # CurvIndex.

TeX_SetCurvLegend

Declaration `procedure TeX_SetCurvLegend(CurvIndex : Integer; Legend : String);`

Description Sets the legend for curve # CurvIndex.

TeX_SetCurvStep

Declaration `procedure TeX_SetCurvStep(CurvIndex, Step : Integer);`

Description Sets the step for curve # CurvIndex.

TeX_PlotCurve

Declaration `procedure TeX_PlotCurve(X, Y : TVector; Lb, Ub, CurvIndex : Integer);`

Description Plots a curve.

Input parameters: X, Y = point coordinates; Lb, Ub = indices of first and last points; CurvIndex = index of curve parameters.

TeX_PlotCurveWithErrorBars

Declaration `procedure TeX_PlotCurveWithErrorBars(X, Y, S : TVector; Ns, Lb, Ub, CurvIndex : Integer);`

Description Plots a curve with error bars.

Input parameters: X, Y = point coordinates; S = errors; Lb, Ub = indices of first and last points; CurvIndex = index of curve parameters.

TeX_PlotFunc

Declaration `procedure TeX_PlotFunc(Func : TFunc; X1, X2 : Float; Npt : Integer; CurvIndex : Integer);`

Description Plots a function.

Input parameters: Func = function to be plotted; X1, X2 = abscissae of 1st and last point to plot; Npt = number of points; CurvIndex = index of curve parameters (Width, Style, Smooth).

The function must be programmed as : function Func(X : Float) : Float;

TeX_WriteLegend

Declaration `procedure TeX_WriteLegend(NCurv : Integer; ShowPoints, ShowLines : Boolean);`

Description Writes legends for all curves.

NCurv: number of curves (1 to MaxCurv); ShowPoints: for displaying points; ShowLines: for displaying lines.

TeX_WriteLegendSelect

Declaration `procedure TeX_WriteLegendSelect(NSelect : Integer; Select : TIntVector; ShowPoints, ShowLines : Boolean);`

Description Writes legends for selected curves.

NSelect : number of selected curves Select : indices of selected curves

TeX_ConRec

Declaration `procedure TeX_ConRec(Nx, Ny, Nc : Integer; X, Y, Z : TVector; F : TMatrix);`

Description Contour plot Adapted from Paul Bourke, Byte, June 1987
http://paulbourke.net/papers/conrec/

Input parameters: Nx, Ny = number of steps on Ox and Oy; Nc = number of contour levels; X[0..Nx], Y[0..Ny] = point coordinates; Z[0..(Nc - 1)] = contour levels in increasing order; F[0..Nx, 0..Ny] = function values, such that F[I,J] is the function value at (X[I], Y[I]).

Xcm

Declaration `function Xcm(X : Float) : Float;`

Description Converts user coordinate X to cm.

Ycm

Declaration `function Ycm(Y : Float) : Float;`

Description Converts user coordinate Y to cm.

14.4 Unit uwinplot

14.4.1 Description

lotting routines for Delphi

14.4.2 Functions and Procedures

InitGraphics

Declaration `function InitGraphics(Width, Height : Integer) : Boolean;`

Description Enters graphic mode.

The parameters Width and Height refer to the object on which the graphic is plotted.

Examples:

To draw on a TImage object: InitGraph(Image1.Width, Image1.Height)

To print the graphic: InitGraph(Printer.PageWidth, Printer.PageHeight)

SetWindow

Declaration `procedure SetWindow(Canvas : TCanvas; X1, X2, Y1, Y2 : Integer; GraphBorder : Boolean);`

Description Sets the graphic window.

X1, X2, Y1, Y2 : Window coordinates in % of maximum GraphBorder : Flag for drawing the window border.

SetOxScale

Declaration `procedure SetOxScale(Scale : TScale; OxMin, OxMax, OxStep : Float);`

Description Sets the scale on the Ox axis.

SetOyScale

Declaration `procedure SetOyScale(Scale : TScale; OyMin, OyMax, OyStep : Float);`

Description Sets the scale on the Oy axis.

GetOxScale

Declaration `procedure GetOxScale(var Scale : TScale; var OxMin, OxMax, OxStep : Float);`

Description Returns the scale on the Ox axis.

GetOyScale

Declaration `procedure GetOyScale(var Scale : TScale; var OyMin, OyMax, OyStep : Float);`

Description Returns the scale on the Oy axis.

SetGraphTitle

Declaration `procedure SetGraphTitle(Title : String);`

Description Sets the title for the graph.

SetOxTitle

Declaration `procedure SetOxTitle(Title : String);`

Description Sets the title for the Ox axis.

SetOyTitle

Declaration `procedure SetOyTitle(Title : String);`

Description Sets the title for the Oy axis.

GetGraphTitle

Declaration `function GetGraphTitle : String;`

Description Returns the title for the graph.

GetOxTitle

Declaration `function GetOxTitle : String;`

Description Returns the title for the Ox axis.

GetOyTitle

Declaration `function GetOyTitle : String;`

Description Returns the title for the Oy axis.

PlotOxAxis

Declaration `procedure PlotOxAxis(Canvas : TCanvas);`

Description Plots the horizontal axis.

PlotOyAxis

Declaration `procedure PlotOyAxis(Canvas : TCanvas);`

Description Plots the vertical axis.

PlotGrid

Declaration `procedure PlotGrid(Canvas : TCanvas; Grid : TGrid);`

Description Plots a grid on the graph.

WriteGraphTitle

Declaration `procedure WriteGraphTitle(Canvas : TCanvas);`

Description Writes the title of the graph.

SetMaxCurv

Declaration `function SetMaxCurv(NCurv : Byte) : Boolean;`

Description Sets the maximum number of curves. Returns False if the needed memory is not available.

SetPointParam

Declaration `procedure SetPointParam(CurvIndex, Symbol, Size : Integer; Color : TColor);`

Description Sets the point parameters for curve # CurvIndex.

SetLineParam

Declaration `procedure SetLineParam(CurvIndex : Integer; Style : TPenStyle; Width : Integer; Color : TColor);`

Description Sets the line parameters for curve # CurvIndex.

SetCurvLegend

Declaration `procedure SetCurvLegend(CurvIndex : Integer; Legend : String);`

Description Sets the legend for curve # CurvIndex.

SetCurvStep

Declaration `procedure SetCurvStep(CurvIndex, Step : Integer);`

Description Sets the step for curve # CurvIndex.

GetMaxCurv

Declaration `function GetMaxCurv : Byte;`

Description Returns the maximum number of curves.

GetPointParam

Declaration `procedure GetPointParam( CurvIndex : Integer; var Symbol, Size : Integer; var Color : TColor);`

Description Returns the point parameters for curve # CurvIndex.

GetLineParam

Declaration `procedure GetLineParam( CurvIndex : Integer; var Style : TPenStyle; var Width : Integer; var Color : TColor);`

Description Returns the line parameters for curve # CurvIndex.

GetCurvLegend

Declaration `function GetCurvLegend(CurvIndex : Integer) : String;`

Description Returns the legend for curve # CurvIndex.

GetCurvStep

Declaration `function GetCurvStep(CurvIndex : Integer) : Integer;`

Description Returns the step for curve # CurvIndex.

PlotPoint

Declaration `procedure PlotPoint(Canvas : TCanvas; X, Y : Float; CurvIndex : Integer);`

Description Plots a point on the screen. Input parameters : X, Y = point coordinates; CurvIndex = index of curve parameters (Symbol, Size, Color).

PlotCurve

Declaration `procedure PlotCurve(Canvas : TCanvas; X, Y : TVector; Lb, Ub, CurvIndex : Integer);`

Description Plots a curve Input parameters: X, Y = point coordinates; Lb, Ub = indices of first and last points; CurvIndex = index of curve parameters.

PlotCurveWithErrorBars

Declaration `procedure PlotCurveWithErrorBars(Canvas : TCanvas; X, Y, S : TVector; Ns, Lb, Ub, CurvIndex : Integer);`

Description Plots a curve with error bars. Input parameters: X, Y = point coordinates; S = errors; Ns = number of SD to be plotted; Lb, Ub = indices of first and last points; CurvIndex = index of curve parameters.

PlotFunc

Declaration `procedure PlotFunc(Canvas : TCanvas; Func : TFunc; Xmin, Xmax : Float; Npt, CurvIndex : Integer);`

Description Plots a function. Input parameters: Func = function to be plotted; Xmin, Xmax = abscissae of 1st and last point to plot; Npt = number of points; CurvIndex = index of curve parameters (Width, Style, Color).

The function must be programmed as : function Func(X : Float) : Float;

WriteLegend

Declaration `procedure WriteLegend(Canvas : TCanvas; NCurv : Integer; ShowPoints, ShowLines : Boolean);`

Description Writes legends for all curves.

NCurv: number of curves (1 to MaxCurv) ShowPoints: for displaying points ShowLines: for displaying lines.

WriteLegendSelect

Declaration `procedure WriteLegendSelect(Canvas : TCanvas; NSelect : Integer; Select : TIntVector; ShowPoints, ShowLines : Boolean);`

Description Writes legends for selected curves.

NSelect: number of selected curves; Select : indices of selected curves.

ConRec

Declaration `procedure ConRec(Canvas : TCanvas; Nx, Ny, Nc : Integer; X, Y, Z : TVector; F : TMatrix);`

Description Contour plot. Adapted from Paul Bourke, Byte, June 1987. http://paulbourke.net/papers/conrec/

Input parameters: Nx, Ny = number of steps on Ox and Oy; Nc = number of contour levels; X[0..Nx], Y[0..Ny] = point coordinates; Z[0..(Nc - 1)] = contour levels in increasing order; F[0..Nx, 0..Ny] = function values, such that F[I,J] is the function value at (X[I], Y[J]).

Xpixel

Declaration `function Xpixel(X : Float) : Integer;`

Description Converts user abscissa X to screen coordinate.

Ypixel

Declaration `function Ypixel(Y : Float) : Integer;`

Description Converts user ordinate Y to screen coordinate.

Xuser

Declaration `function Xuser(X : Integer) : Float;`

Description Converts screen coordinate X to user abscissa.

Yuser

Declaration `function Yuser(Y : Integer) : Float;`

Description Converts screen coordinate Y to user ordinate.

Chapter 15

Changes in LMath and LMComponents

Ver. 0.6.1

LMath

1. Version with open arrays was added for function `ApplyRecursive` (see 13.5.2.2).

2. FirstElement: -1 is returned if nothing is found. See 13.5.2.5. Previous behaviour: `Ub+1` was returned. Reason for change: -1 is definitely impossible value, therefore it is easier to organize testing.

3. CompVec (13.5.2.3): if compared arrays have different length, the function returns normally and FALSE is returned. Previous behaviour: incompatible dimensions error was generated. Reason for the change: vectors of diffrent length are indeed not equal.

4. Apply, ApplyRecursive, InitWithFunc, CompVec, Any, FirstElement, MaxLoc, MinLoc, Seq, ISeq, SelElement: check for $Lb < 0$ added. ExtractElements: check for errors in border definition added.

5. In the unit uMeanSD_MD (7.2) functions `LastDefined`, `StdErr Defined` and `Undefined` for array element added,; for many functions variants with open arrays added or default values `Lb = 0` and `Ub = -1` added, where -1 means to use High(A) for Ub. These changes allow to omit Lb, Ub by function call.

6. In the unit uStrings (13.11) functions `ArrayToStr` and `ArrayToStrMD` added

7. In the unit uMinMax (2.4) function `InRange` added

LMComponents

LMFilters Now the unit uses `uVectorHelper` and `uFilters` from LMath library. Hence, functions `GaussCascadeFreq`, `GaussRiseTime`, `MovAvRiseTime`, `MoveAvCutoffFreq`, `MoveAvFindWindow` were removed, because they are contained in `uFilters`.

For TMoveAvFilter and TMedianFilter, possibility of a filtering in real time, during data aquisition, is added.

In `TGaussFilter` new event, `OnInputBackward`, was added. TGaussFilter algorihtm fulfils filtering in two steps, first from beginning to end (forward filtering), and this once filtered dataset is subsequently filtered from the end to the beginning. If the calling program keeps initial dataset unmodified and places the filtered data to a new one, you must define the `OnInputBackward` event such that it reads data from the new, filtered, dataset. If you filter data *in situ*, modifying initial data, defining this event is not necessary; filter will use the usual `OnInput` event both for forward and backward filtering.

lmRecursFilters This is a new unit, which defines several new filters:
`THighPassFilter`, which passes high frequency and stops low;
`TNotchFilter`, which filters out a narrow band around a selected frequency and may be useful for example to filter out 50 Hz hum;

`TBandPassFilter`, which passes only slected band;
`TChebyshevFilter`, which implements a Chebyshev type filter.

Ver. 0.6.0

1. Package `lmDSP` (chapter 10) created, which contains functions for digital signal processing. It includes `uFFT` unit (10.3) for fast Fourier transform, moved from `lmOptimum` package, and new units: `uConvolutions` (10.2), for convolution of two arrays of `Float`, `uDFT` unit, for digital Fourier transform of signals with length other then degrees of two (10.4). Importantly, this unit is distributed under GPL license, not LGPL, as the rest of the library. Unit `uFilters` (10.5) contains procedures for various data filtering: low pass, high pass, bandstop and bandpass.

2. New unit `uCompVecUtils` (13.9) in `lmMathUtil` package. The unit implements several utility routines for work with arrays of Complex numbers, somewhat similar to `uVecUtils` for `Float`.

3. Call convention of many functions and procedures working with arrays is revisited. Mostly for historical reasons, related in part to Fortran inheritance, in part to limitations of old Pascal implementations, standard API of DMath/LMath includes passing of a dynamic array (`TVector, TMatrix`, etc) and bounds of a slice which is actually processed by a function (`Lb,Ub`).

   `Function Apply(V:TVector;Lb,Ub:integer; Func:TFunc)`

   is an example of such call. This approach has an important limitation that only dynamic, but not static, arrays may be used with LMath. Besides, mandatory passing of bounds, even if the whole array must be processed, makes the call too complicated. Modern Pascal implementations have open array parameter mechanism and even allow to pass slices of arrays, both static and dynamic. Therefore, as an experimental feature, LMath 0.6 introduced new form of function calls where instead of `V:TVector, Lb, Ub:integer` open array is passed. `Apply` in this new form looks like:

   ] `Apply(var V:array of Float, Func:TFunc);`.

   In most cases, old forms are kept for backward compatibility as overloaded functions. Operators over arrays defined in `uMatrix` unit were also redefined for open arrays. This makes them much more flexible. For example, it is possible now to write:

   `V := MyArr[4..9]+AnotherArr[7..12]`,

   which will create `V:TVector` with length 6, containing sums of elements `MyArr[4]+AnotherArr[7]`, etc.

   Work with new call conventions is still in progress and this feature is considered experimental. For now, functions and procedures from `uMatrix` (3.2), `uVecUtils` (13.8), `uCompVecUtils` (13.9), `uMedian` (7.3), `uMeanSD` (7.1), `uFFT` (10.3) got these new forms. New units have only form with open arrays. Unfortunately, there is no way to define open two-dimentional arrays, therefore all operations with `TMatrix` keep old form only.

   It must be noted, that in some cases subtle differences in implementation of functions in these two forms exist. For example, function Median(V:TVector,Lb,Ub:integer)

gets `V` by reference, as always with dynamic arrays and rearranges it in the process of median search. In contrast,

`function Median(V:array of Float)`

gets it by value, hence, old array remains unchanged. This is done deliberately to avoid unwanted side effect. In most other cases open arrays are passed by reference, either as `var` or `constref` parameters.

4. Few bugs were fixed.

Modifications of DMath Code

These are not numerous.

1. In many cases, I have changed `var` descriptor to `out` for output parameters in function calls, to avoid getting tons of unmeaningful warnings. So, now a user must take seriously compiler complaints on possibly uninitialized variables. However, I do not guarantee that I did it everywhere.

2. Instead of many various `Dim*Vector` and `Dim*Matrix`, overloaded DimVector and DimMatrix procedures for all types of vectors and matrices were defined.

3. Similarly, changed `FSwap` and `ISwap` to universal Swap; Min and Max are overloaded as well.

4. In uEval unit, ParsingError variable was made public, to enable a calling program to react adequately if an invalid expression was passed for evaluation. Names of the variables may be of any length and, unlike DMath, the whole name is meaningful. Previously, only first letter was meaningful for expression parser.

5. Converted into Lazarus and recompiled GUI and BGI demo programs.

Additions before and including ver 0.5.1

1. Added TRealPoint type which represents a point on a cartesian plane and defined several procedures and operators over it (see 2.18).

2. Defined IsZero, IsNAN and SameValue functions together with accompanying SetEpsilon and SetZeroEpsilon functions.

3. Improvements in error handling: SetErrCode function allows to set not only numeric error code, but text message as well. Consequently, MathErrMessage function allows to read it. Standard error messages for standard error defined. See 2.3.1 for details.

4. In the unit `uMinMax`, function `Sign` was added with the same semantics as in Math unit, for compatibility reasons.

5. In `uMath`, operator `**` was defined.

6. In `uCompex` unit, opreators over complex numbers defined (inspired by `uComplex` unit from Free Pascal RTL).
7. uIntervals unit was written where TInterval type is defined which represents an interval between two real numbers, several procedures with it are defined. See 2.17.1 for details.
8. Unit uMatrix (3.2) was written which defines some basic operation of linear algebra.
9. uCrtPtPol unit written, with the procedures to find critical points of a polynomial, see 4.8.
10. Unit uMeanSD_MD written which defines functions of descriptive statistics over array containing missed values. By default, missed value is represented as NAN, but any code can be defined.
11. In the unit uNormal, function DGaussian added to evaluate a normal distribution with arbitrary μ and σ.
12. Unit uSpline written, for cubic spline interpolation of a set of points and to investigate the resulting spline function, finding roots and extremums.
13. Goldman-Hodgkin-Katz equation for current and Sum of Gaussians distribution were added to the library of fitting models (see 12.3, 12.4 and 12.5).
14. Estimates and models of exponential, hyperexponential and hypoexponential distributions added (see 12.2).
15. Optimization algorithm COBYLA, innitially written in FORTRAN by Michael J. D. Powell was implemented and used for constrained non-linear regression (see 8.14 and 11.5).
16. Unit uLinSimplex which solves linear programming problems with simplex method was written (8.16).
17. Unit lmSorting (13.4) written which contains procedures for quick sort, insert sort and heap sort of Float and TRealPoint arrays.
18. Unit lmUnitsFormat written, for formatting values with units using SI prefixes (femto-, pico-, . . . , Tera-, Peta-).
19. Units uVecUtils (13.5), uVectorHelpers (13.7), uVecFunc (13.6) and uVecMatPrn (13.10) were written which define utility functions and type helpers for easier work with vectors and matrices.
20. Unit uVecFileUtils written, which contains procedures for saving `TMatrix` or `TVector` to a file and read them from a file.

Units

Types

Constants

Procedures and Functions