LMath for DMath users

Reference Guide to New and Changed Functions and Behavior

Viatcheslav V. Nesterov

September 25, 2024

Contents

1 General Notes

1.1 Purpose of this manual

LMath is a further development of DMath library by Jean Debord. Most of available functions are described in the manual for DMath. However, LMath names or behavior of some DMath functions are changed in LMath; some new ones were added. This document contains brief description of such changes and additions. More formal definitions can be found in the Reference Guide for LMath.

1.2 Installation and use of the library

First, download LMath_and_Components_06_1.zip from the official site of the program. You can download and use trunk of SVN repository to study and research the latest development, but I do not guarantee that code in the trunk was properly tested or even that it is a working code at all.

LMath is organized as a set of Lazarus packages. Their complete list is located in the Reference Guide. To install, copy directory structure contained in LMath_and_Components_06_1.zip on your hard drive and compile packages (.lpk files) in the following order:
lmGenMath; lmMathUtil; lmIntegrals.lpk; lmLinearAlgebra; lmMathStat; lmRandoms; lmOptimum; lmNonLinEq; lmPolynoms; lmPlotter; lmRegression;
lmSpecRegress, lmDSP. Finally, compile LMath.lpk.

For maximally broad compatibility, all LPK files have low level of optimization in options. Change them before compilation of final version of your program to the settings, optimal for your system. Try optimization levels 3 or 4. For modern systems you may want to use -OpCOREAVX2 and -CfAVX2 options. Consider also use of -OoFASTMATH and -OoLOOPUNROLL.

Conversely, if you get weird errors, inspect options of the packages and make sure that they are indeed conservative enough.

See Free Pascal Wiki for more details about optimizations.

There is no need to install packages from LMath in the lazarus IDE. If you plan to use LMComponents, LMComponents.lpk must be installed, and it installs rather large subset of LMath. To use the library in your programs, add needed packages to the requirements of your project and, of course, necessary units in your `uses` clause. Alternative to adding every needed package, you can add to requirements the general LMath package.

Before LMath ver. 0.5, names of both packages and units began with "u", since 0.5 names for packages begin with "lm".

In contrast to DMath, no special means for compilation of LMath as a shared library is provided. I find it unlikely that anybody would use this library in a language other then pascal, and in pascal it is always more convenient to have a single executable than an executable and required external libraries.

2 Note about array indexing in DMath and LMath

It must be noted here that many functions in DMath/LMath library were translated or adapted from Fortran which has two notable differences from Pascal. First, it keeps arrays in the order "columns first", while Pascal keeps them as "rows first". Consequently, many algorithms included in this library are optimized for "columns first" order which seriously slows down the Pascal implementation. We hope to address this issue, at least partially, in later versions of the library.

Second, in Fortran array indexing begins from "1", while in Pascal from "0". In DMath/LMath, many internal procedures are written in Fortran manner, beginning numeration with "1" which in Pascal implementation means of course that a first element of an array, indexed "0", remains unused. This is a small price which we pay for the possibility to use algorithms developed in Fortran without modifications and introducing new bugs.

However, it would be somewhat awkward to force the users of the library adopt "1"-based indexing. This is one of the reasons that in most cases when an array is passed as a parameter to a routine of the library, there are additional parameters `Lb` and `Ub` which stands for "Low bound" and "Upper bound" and which define a slice of an array on which the operation must be performed. Typically, `Lb` can be "0" or "1" and `Ub` can be `High(Passed_array)` although of course they may take any values in between provided that $Lb < Ub$. This convention makes our library more flexible, although, admittedly, at the expense of more complicated syntax. When "0"-based or "1"-based array is mentioned in this manual, it means that actual data begin from index 0 or 1; technically, all dynamic arrays in Object Pascal are 0-based.

3 Using open array parameters: LMath 0.6 and later.

`Lb, Ub` convention adopted by DMath and LMath is rather flexible, but has two drawbacks: first, the calls are rather complicated with a lot of parameters passed and, second, passing TVector as a parameter excludes use of static arrays. 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. Therefore, beginning from LMath 0.6.0, many functions and procedures with open array parameters instead of `TVector, Lb, Ub` are developed. Older versions are kept as well.

Now, one important note to be kept in mind. Passing for example
SomeFunc(V,2,5):TVector in the form `TVector, Lb, Ub` and
SomeFunc(V[2..5], R):TVector in the form `Open Array`
would not be the same, because in the first case the whole array is passed to the function and returned, but whatever the function does is applied to a slice `[2..5]`, while in the second case only part of the array is passed and returned. This is the reason why in some cases, when this difference is important, functions were converted to procedures.

4 Changes since version 0.4 which may require modification of calling code

1. In `function CriticalPoints`, unit `uCrtPtPol`, allocation of `CrtPoints` and `PointTypes` prior to function call is not necessary. The function can allocate them itself.
2. Indexing of `Minima` and `Maxima` returned by `FindSplineExtremums` begins now from 1 and not from 0. Reason: internal consistency of the library.
3. Various `Dim*Vector` and `Dim*Matrix` functions were replaced by overloaded universal `DimVector` and `DimMatrix`.
4. Package lmLineAlgebra was renamed to lmLinearAlgebra.

5 Error handling

Mechanisms of error handling in DMath and LMath are defined in `uErrors` unit. In DMath, success or failure of a function was defined by a numeric error code, which could be checked by call of `MathErr` function. However, numeric error code is not particularly informative, especially for end users. Therefore, in LMath numeric error codes are supplemented with text error messages. With this purpose, array of strings, corresponding to error codes was defined in `uErrors` unit. List of standard error codes and messages can be found in the *uErrors* section in LMath Reference Guide. Additionally, a user can define own error codes and messages and set them by call of

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

If `EMessage` parameter remains empty and ErrCode is a standard error code, then corresponding message from the standard list is set. Otherwise, error message is set to `EMessage`. Error code can be retrieved by call of

```
function MathErr : Integer;
```

and text message with

```
function MathErrMessage :  string;
```

Note that both functions must be called immediately after the call to mathematical function, before they could be rewritten by a call of another function. This may be changed in later versions.

6 NAN and tests for approximate equality

Due to rounding errors, testing floating numbers for exact equality can be unreliable. Therefore, unit `uTypes` in LMath defines following functions for approximate equality:

```
function IsZero(F: Float; Epsilon: Float = -1): Boolean;
function SameValue(A, B: Float; epsilon:float): Boolean; overload;
function SameValue(A,B:Float):boolean; overload;
```

`IsZero` returns true if $|F| < epsilon$. If parameter `epsilon` is not used, it is set by default to `MachEp/8` where MachEp is floating point precision, see Reference Guide for LMath. Default value of `epsilon` for `SetZero` can be changed with the

help of
`procedure SetZeroEpsilon(AZeroEpsilon:float)`.

First of `SameValue` functions returns true if $|A - B| < epsilon$. Third function is most sophisticated, it uses comparison with epsilon scaled to the scale of compared numbers.

Besides that, constants `NAN` (not a number), `Infinity` and `NegInfinity` are defined and function `IsNAN`. `NAN` and `IsNAN` are used in LMath to represent and check missing values in some statistical functions, see Section 15.

7 General purpose functions and procedures

In DMath, `uTypes` unit contained a family of functions `Dim*Vector` and `Dim*Matrix` for allocation of arrays of different base types. In LMath, procedures `DimVector` and `DimMatrix` are overloaded for different types:

```
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);

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);
```

Similarly, following general procedures for scalars are defined:

```
function Min(X, Y : Float) : Float;
function Min(X, Y : Integer) : Integer;
function Max(X, Y : Float) : Float;
function Max(X, Y : Integer) : Integer;
function Sgn(X : Float) : Integer;
function Sgn(X : integer) : integer;
function Sgn0(X : Float) : Integer;
function Sgn0(X : integer) : integer;
function DSgn(A, B : Float) : Float;
function Sign(X: Float):integer; inline;
function IsNegative(X: float):boolean;
function IsNegative(X: Integer):boolean;
function IsPositive(X: float):boolean;
function IsPositive(X: Integer):boolean;
procedure Swap(var X, Y : Float);
procedure Swap(var X, Y : Integer);
```

`Function Sign` has the same semantic as `Sgn0` and was introduced for compatibility with Math unit.

8 New Operators

Free Pascal allows to overload operators. LMath took advantage of this feature. It defines operator `**` for exponentiation for `Float` in `integer` degree and `Float` in `Float` degree.

For complex numbers (type `complex`), unit `uComplex` in LMath defines operators `+,-,*,/, =` (comparison) for `complex` and `float` in all meaningful combinations. This part is largely based on ideas from uComplex unit by Pierre Müller.

Operators defined for work with other structural types will be discussed in the next chapter where corresponding types are discussed.

9 TInterval

Type `TInterval` defined in `uInterval` unit describes an interval on a numeric axis:

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

Function `Length` returns the length of this interval, that is, $Hi - Lo$. This type is useful for example in the tasks of signal processing where it can represent a window in finite impulse response filters. Another use is presentation of a fragment of axis shown on a screen or printout.

Following functions and procedures are defined over this type:

`function IntervalsIntersect(Lo1, Hi1, Lo2, Hi2:Float):boolean; overload;`
Returns true if intervals [Lo1; Hi1] and [Lo2; Hi2] intersect.

`function IntervalsIntersect(Lo1, Hi1, Lo2, Hi2:Integer):boolean;`
`function IntervalsIntersect(Interval1, Interval2:TInterval):boolean;`
Returns true if intervals [Lo1; Hi1] and [Lo2; Hi2] intersect.

`function Contained(ContainedInterval,ContainingInterval:TInterval):boolean;`
Returns true if interval ContainedInterval is located completely inside ContainingInterval:
`(ContainedInterval.Lo > ContainingInterval.Lo) and`
`(ContainedInterval.Hi < ContainingInterval.Hi).`

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

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

`function Inside(V:Float; AInterval:TInterval):boolean; overload;`
True if V is inside `AInterval`.

`function Inside(V:float; ALo, AHi:float):boolean; overload;`
True if `V` is inside `(ALo, AHi)` (similar to `Math.InRange`)

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

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

`function DefineInterval(ALo,AHi:Float):TInterval;`
Constructs `TInterval` from `ALo` and `AHi`.

`procedure MoveInterval(V:float; var AInterval:TInterval);`
Move interval by a value (it is added to both `Lo` and `Hi`)

`procedure MoveIntervalTo(V:Float; var AInterval:TInterval);`
Move interval to a value (`Lo` is set to this value, `Hi` adjusted such that length remains constant).

10 TRealPoint and TRealPointVector

`TRealPoint` represents simply point on a Cartesian plane, `PRealPoint` is a typed pointer on `TRealPoint` and `TRealPointVector` is an array of `TRealPoint` which is often useful as a representation of a signal or function in a tabbed form.

```
PRealPoint = ^TRealPoint;
TRealPoint = record
  X: Float;
  Y: Float;
end;
TRealPointVector =  array of TRealPoint;
```

Unit `uRealPoints` defines several functions and operators over `TRealPoint` treating it as a vector in 2-dimentional space.

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

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`.

`function rpPoint(AX, AY:float):TRealPoint;`
Constructs a `TRealPoint` from two floats.

`function rpSum(P1,P2:TRealPoint):TRealPoint;`
Sum of `TRealPoint`.

`function rpSubtr(P1, P2:TRealPoint):TRealPoint;`
Subtraction of `TRealPoint`.

`function rpMul(P:TRealPoint; S:Float):TRealPoint;`
Multiplication of `TRealPoint` by Scalar.

`function rpDot(P1, P2:TRealPoint):Float;`
Dot product of `TRealPoint`.

`function rpLength(P:TRealPoint):Float;`
Length of vector, represented by `TRealPoint`.

`function Distance(P1, P2:TRealPoint):Float;`
Distance between two `TRealPoint` on cartesian plane.

These operations may be invoked also in operator form:

`P1 + P2` where `P1` and `P2` are `TRealPoint`: summation (equivalent of `rpSum`).
`P1 - P2` is substraction (equivalent of `rpSubstr`).
`P * S` or `S * P` where P is TRealPoint and `S` is scalar is multiplication by scalar (equivalent of `rpMul`).
`P1 * P2` where `P1` and `P2` are `TRealPoint` is dot product (equivalent of `rpDot`).

11 TIntegerPoint

`TIntegerPoint` similarly to `TRealPoint` represents a point on a Cartesian plane, but with integer coordinates, mainly pixel coordinates on a screen or canvas.

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

Unit `uIntPoints` defines few functions and operators over `TIntegerPoint`.

`function ipPoint(AX, AY:integer):TIntegerPoint;`
Constructs of TIntegerPoint from two integers

`function ipSum(P1,P2:TIntegerPoint):TIntegerPoint;`
Summation of TIntegerPoint.

`function ipSubtr(P1, P2:TIntegerPoint):TIntegerPoint;`
Subtraction of TIntegerPoint

`function ipMul(P:TIntegerPoint; S:integer):TIntegerPoint;`
Multiplication of TIntegerPoint by Integer.

Same operations may be invoked in operator form:
`P1 + P2` or `P1 - P2` where P1 and P2 are `TIntegerPoint`.
`P * S` or `S * P` where P is `TIntegerPoint` and `S` is integer.

12 Derivative and Critical Points of a Polynom

Unit `ucrtptpol` defines two routines needed for investigation of polynomial functions.

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

finds an expression for a derivative of input polynomial of degree `Deg`. This derivative is, obviously, polynomial of `Deg-1` degree. Input parameters: `Coef` is 0-based array of coefficients of the input polynomial, beginning from free member, such that array index is equal to the degree of corresponding member. `Deg` is degree of the input polynomial. Output: DCoef: coefficients of derivative polynomial, DDeg is degree of derivative polynomial. For example, to find a derivative of polynomial

$$2x^4 + 3x^3 + x - 2 :$$

```
uses uTypes, uVectorHelper, ucrtptpol;
const
  Deg = 4;
var
```

```
  Coefs, DCoefs : TVector;
  DDeg : integer;
begin
  Coefs.FillWithArr(0,[-2,1,0,3,2]);
  DerivPolynom(Coefs, Deg, DCoefs, DDeg);
end;
```

To find critical points, where derivative is zero, use

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

`Coef` and `Deg` have same meaning as in `DerivPolynom` function, `Coef` is similarly 0-based. `CrtPoints` of TRealPoints type contains upon call abscissas in `X` and ordinates in `Y` of critical points found, and `PointTypes` contain information about the type (-1 for minimum, 1 for maximum or 0 for no extremum) of every found critical point. CrtPoints and PointTypes place meaningful infortmation beginning from index `ResLb`, by default 1. Example:

```
  NCrtP := CriticalPoints(Coefs,Deg,CrtP,PointTypes);
```

Example programs to these functions are `extremum.lpr` and `polyderiv.lpr` located at `demo/console/polynoms`.

13 Utility types and functions for work with arrays

13.1 uVectorHelper

Unit `uVectorHelper` in the package `lmMathUtil` defines type helpers for `TVector` and `TIntVector` to make more convenient simple operations with vectors, such as filling with a value or array at the initialization, swapping two elements, sorting or string representation.

Beginning from ver. 3.2, Free Pascal supports intrinsic procedures Delete() and Insert() over dynamic arrays with functionality similar to `Insert` and `Remove` implemented here. *However, unlike these intrinsics, our procedures do not change length of an array.* Insert() leads to the loss of last elements of an array; Remove() appends zeroes at the end of the array. This implementation avoids call to `SetLength` at each deletion/insertion, making the code faster. Use standard procedures if you need changed length.

```
TVectorHelper = type helper for TVector
  procedure Insert(value:Float; index:integer);
  procedure Remove(index:integer);
  procedure Swap(ind1,ind2:integer);
  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); overload;
  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 }

TIntVectorHelper = type helper for TIntVector
  procedure Insert(value:Integer; index:integer);
  procedure Remove(index:integer);
  procedure Fill(Lb, Ub : integer; Val:Integer);
  procedure FillWithArr(Lb : integer; Vals:array of Integer);
  procedure InsertFrom(Source:TIntVector; Lb, Ub: integer; ind:integer); overloa
  procedure InsertFrom(constref source: array of integer; ind: integer); overloa
  procedure Swap(ind1,ind2:integer);
  function ToString(Index:integer):string;
  function ToStrings(Dest:TStrings; First, Last:integer;
  Indices:boolean; Delimiter: char):integer;
end;
```

These types are similar and will be described here together.

`Procedure Insert` inserts a value into position `index`, all following elements of array are shifted to the right. Last element is lost.

`Procedure Remove` is opposite to `insert`. It removes an element in position index; following elements shifted to the left. Last element is set to 0.

`Procedure Fill` Fills `Self` from `Self[Lb]` to `Self[Ub]` with `Val` replacing old values. If `En` > `High(Self)` then `Self` is filled to its end.

`Procedure FillWithArr` fills `Self` beginning from `Self[Lb]` with the elements of array `Vals` replacing old values. If `length(Vals)` > `High(Self)-Lb` then only part of Vals is used. Main purpose of this procedure is convenient array initialization.

`Procedure InsertFrom` inserts elements of source in the range `Source[Lb]` to `Source[Ub]` into `Self` beginning from position `Self[Ind]`. All following elements are shifted to the right. Rightmost `Ub - Lb` elements of `Self` are lost.

If `Self` was not allocated prior to the call of `Fill`, `FillWithArr` or `InsertFrom`, its length is set to the necessary values, otherwise length of `Self` is not changed by any of these methods. If length of arrays in `FillWithArr` or `InsertFrom`, they are used only partially such that `Self` is filled up to existing `High(Self)`. All `Insert*` procedures overwrite last part of the array, therefore, if necessary, call `SetLength` or `DimArray` before call to these procedures. We do it this way for two reasons. First, all these procedures can be applied to the lines of a matrix and we do not want ragged matrices. Second, `SetLength` is very expensive and it is better to avoid its call for every insertion.

`procedure Swap` swaps elements in positions Ind1 and Ind2.

`Function ToString` returns a string representation of a value at a position `Index`.

`Procedure ToStrings` sends string representations of elements from `First` to

Last into Dest:TStrings. If Indices is true, then each line contains element index, delimiter as defined at the function call, and value of the corresponding element.

These methods use FloatStr and IntStr functions from uStrings unit for formatting.

13.2 uVecUtils

This unit defines several handy functions and procedures for work with one- and two-dimensional arrays.

Types

```
TMatCoords = record
  Row, Col :integer;
end;
```

Besides, this unit uses two functional types defined in TTypes unit, lmGenMath package:

```
TTestFunc = function(X:Float):boolean;
TIntTestFunc = function(X:Integer):boolean;
```

TTestFunc and TIntTestFunc are used as arguments for functions described below. TMatCoord represents a position of an element in a matrix.

tmCoords

```
function tmCoords(ARow,ACol:integer):TMatCoords;
```
Function tmCoords constructs TMatCoords from two integers.

Apply

Procedure Apply applies a function to every element of an array in a range from Lb to Ub. Applied function must have a type TFunc for arrays of Float and TIntFunc for arrays of Integer. Both are defined in uTypes unit. Apply passes an element of an array to this function and assigns the result to the element. It is defined for TVector, TMatrix, TIntVector and TIntMatrix:

```
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);
```

Besides that, exists a masked version of Apply which applies the function only to elements with indices listed in additional array mask.

```
procedure Apply(V:TVector; Lb, Ub: integer;
          Mask:TIntVector; MaskLb:integer; Func:TFunc);
procedure Apply(V:TIntVector; Lb, Ub: integer;
          Mask:TIntVector; MaskLb:integer; Func:TIntFunc);
```

These masked functions can be especially handy when used together with Selelements

(see 13.2.10).

Beginning from LMath 0.6, open array versions of `Apply` functions are available:

```
procedure Apply(var V:array of Float; Func:TFunc); overload;
procedure Apply(var V:array of Integer; Func:TIntFunc); overload;
```

In these versions, dynamic and static arrays or even slices of arrays can be passed. There are no open array versions of masked `Apply` functions, because open array indexing inside the function always begins from zero so that internal indexing does not always correspond to the external one and this discrepancy can lead to lots of errors.

ApplyRecursive

```
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;
```

CompVec

Function `CompVec` (compare vectors) checks if each element of vector X is within a fraction Tol of the corresponding element of the reference vector Xref. In this case, the function returns True, otherwise it returns False.

```
function CompVec(X, Xref : TVector; Lb, Ub  : Integer;
   Tol : Float) : Boolean;
```

Any

Function `Any` applies test function (TTestFunc or TIntTestFunc) to every element of a target array and returns `True` if the test function returns `True` for at least one element.

```
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;
```

Beginning from version 0.6, open array versions of this function are available:

```
function Any(constref Vector:array of Float; Test:TTestFunc):boolean;
function Any(constref Vector:array of integer; Test:TIntTestFunc):boolean;
```

FirstElement

Function `FirstElement` returns a position of a first element which satisfies a condition. The function is implemented for `TVector`, `TIntVector`, `TMatrix` and `TIntMatrix`. In the latter two cases position is returned as `TMatCoords` record:

```
TMatCoords = record
```

```
  Row, Col :integer;
end;
```

If nothing is found, the function returns $Ub + 1$.

Search condition may be defined in several ways. First, with the help of test function, same way as implemented in `Any`.

```
function FirstElement(Vector:TVector; Lb, Ub : integer;
   Test:TTestFunc):integer;
function FirstElement(M:TMatrix; LRow, URow, LCol, UCol : integer;
   Test:TTestFunc):TMatCoords;
```

In a second implementation, a user defines the reference value and the type of comparison. Type of comparison is represented by a parameter of type `TCompOperator` which is defined in `uTypes` unit as follows:
`TCompOperator = (LT,LE,EQ,GE,GT,NE);`
In this form, `FirstElement` returns a position of a first element which satisfies following condition:

Op	Condition		
LT	x[I]	<	Ref
LE	x[I]	<=	Ref
EQ	x[I]	=	Ref
GE	x[I]	>=	Ref
GT	x[I]	<	Ref
NE	x[I]	<>	Ref

```
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;
```

This is the most efficient way to call `FirstElement` because it does not involve calls of test functions for every element, and calls are expensive. However, if more complex algorithms are needed than simple comparisons, it is possible to build and use a user-defined comparator function:

```
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:TIntVector; Lb, Ub : integer; Ref:integer;
   Comparator:TIntComparator):integer;
function FirstElement(M:TIntMatrix; LRow, URow, LCol, UCol : integer;
   Ref:integer; Comparator:TIntComparator):TMatCoords;
```

TComparator here is a function which takes two float values as arguments and returns boolean. Similar, TIntComparator takes two integers and returns boolean. Both TComparator and TIntComparator types are defined in Types unit as follows:

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

Element of an array is passed to a comparator as a first argument, reference value as a second.

MaxLoc, MinLoc

Functions MaxLoc and MinLoc return position of a maximal or minimal element of an array or a slice of an array.

```
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;
```

Seq, ISeq

Functions Seq and ISeq generate an arithmetic progression and fill with it a slice of an array between Lb and Ub. If prior to the call Vector is empty, it is allocated with the length Ub + 1 and fills a slice $Lb..Ub$ with the generated sequence:

$$First, First + Increment, \ldots, First + Increment \cdot (Ub - Lb)$$

```
function Seq(Lb, Ub : integer; first, increment:Float;
   Vector:TVector = nil):TVector;
function ISeq(Lb, Ub : integer; first, increment:integer;
   Vector:TIntVector = nil):TIntVector;
```

SelElements and ExtractElements

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).

```
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:TVector; Lb, Ub, ResLb : integer;
    Test:TTestFunc):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;
function SelElements(Vector:TIntVector; Lb, Ub, ResLb : integer;
    Test:TIntTestFunc):TIntVector;
```

Function ExtractElements allows to extract selected elements into a separate `TVector`:

```
function ExtractElements(Vector:TVector;
      Mask:TIntVector; Lb:integer):TVector;
```

Lb is applied for both sourse `Vector` and function result. Demo programs for `uVectorHelper` and `uVecUtils` are contained in `demo/console/MathUtil` folder.

13.3 uVecFunc

Small unit uVecFunc defines procedures which apply `Sqrt` and `Abs` functions to every element of a vector or matrix:

```
procedure VecAbs(V : TVector; Lb, Ub : integer);
procedure VecAbs(V : TIntVector; Lb, Ub : integer);
procedure MatAbs(M : TMatrix; Lb1, Ub1, Lb2, Ub2 : integer);
procedure MatAbs(M : TIntMatrix; Lb1, Ub1, Lb2, Ub2 : integer);

procedure VecSqr(V : TVector; Lb, Ub : integer);
procedure VecSqr(V : TIntVector; Lb, Ub : integer);
procedure MatSqr(M : TMatrix; Lb1, Ub1, Lb2, Ub2 : integer);
procedure MatSqr(M : TIntMatrix; Lb1, Ub1, Lb2, Ub2 : integer);

procedure VecSqrt(V : TVector; Lb, Ub : integer);
procedure MatSqrt(M : TMatrix; Lb1, Ub1, Lb2, Ub2 : integer);
```

13.4 uSorting

Another important operation on arrays is sorting. Unit `uSorting` implements three different algorithms: Quick sort, Insert sort and Heap sort. Of them, quick sort is fastest in general case, but becomes very inefficient if an input array is already highly ordered, which is not uncommon. Besides, it is implemented recursively, which makes it relatively unsafe. Heap sort is slightly slower in general, but performs much better in worst case. In most cases, it is a preferrable choice. Insertion sort is not very efficient for long arrays, but works well in arrays up to 50 elements, and is quite efficient for initially highly ordered arrays. All three algorithms are implemented for `TVector`, `TRealPointVector` for X and `TRealpointVector` for Y.

```
procedure QuickSort(Vector : TVector; Lb,Ub:integer; desc:boolean);
procedure QuickSortX(Points : TRealPointVector;
   Lb,Ub:integer; desc:boolean);
procedure QuickSortY(Points : TRealPointVector;
   Lb,Ub:integer; desc:boolean);

procedure InsertSort(Vector : TVector; Lb,Ub:integer; desc:boolean);
procedure InsertSortX(Points : TRealPointVector;
   Lb,Ub:integer; desc:boolean);
procedure InsertSortY(Points : TRealPointVector;
   Lb,Ub:integer; desc:boolean);

procedure Heapsort(Vector:TVector; Lb, Ub : integer; desc:boolean);
procedure HeapSortX(Points:TRealPointVector;
   Lb, Ub : integer; desc:boolean);
procedure HeapSortY(Points:TRealPointVector;
   Lb, Ub : integer; desc:boolean);
```

If `Desc` is true, sorting in descending order is performed.

13.5 uVecFileUtils

This unit defines routines which allow to save TVector or TMatrix into a delimited text file or read it from such file.

```
procedure SaveVecToText(FileName:string; V:TVector; Lb, Ub:integer);
```

saves slice of a vector `V[Lb..Ub]` as a column into a file. If $Ub > High(V)$, vector till the end is saved.

```
function LoadVecFromText(FileName:string;
    Lb:integer; out HighLoaded : integer) : TVector;
```

allocates and loads `TVector` from a file. Length of allocated vector is `Lb` plus number of lines in the file. At a reading, if a line cannot be parsed as a valid real number, it is silently skipped. Number of actually read values is returned in `HighLoaded`.

```
procedure SaveMatToText(FileName : string;
  M : TMatrix; delimiter : char;
  FirstCol, LastCol, FirstRow, LastRow : integer);
```

saves a rectangular slice of a matrix M[FirstCol..LastCol, FirstRow..LastRow] into a file, values within a row delimited by `Delimiter`.

```
function LoadMatFromText(FileName: string;
   delimiter: char; Lb:integer; MD:Float): TMatrix;
```

allocates and loads a matrix from text file. If $Lb \neq 0$, rows and columns with indices $< Lb$ remain empty (filled with 0). If a cell in the file cannot be parsed (for example, is empty line), corrsponding 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. Importantly, maximal number of delimiters per line of a file is used to determine length of a row in the matrix. Hence, a space can be used as a delimiter only if a delimiting space is always singular. 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 pro 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.

13.6 uVecMatPrn

This unit defines procedures `PrintVector` and `PrintMatrix` for console output.

```
procedure PrintVector(V:TIntVector);
procedure PrintVector(V:TVector);
procedure PrintMatrix(A:TMatrix);
```

If output begins from V[0] or V[1], is defined by variable `vprnLB`, default value is 1. Numbers are formatted according to typed constants

```
vprnFmtStr : string = '%8.3f';
vprnIntFmtStr : string = '%4d';
```

14 Linear Algebra enhancements

14.1 uMatrix

Unit `uMatrix` in the `lmLinearAlgebra` package defines several routines for basic linear algebra operations which were strangely missing in the original DMath library. Following functions perform element-wise operation with a `TVector` and `Float`: a `Float` is added to or subtracted from a vector `V`:

```
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;
```

`R` is added to (`VecFloatAdd`) or subtracted from (`VecFloatSubtr`) every element of `V` beginning from `V[Lb]` and to `V[Ub]`.

```
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;
```
Operation is performed, result is placed in `Ziel` array beginning from `ResLb`. Note that default value for `ResLb` is 1. Reason for it is that many routines in DMath and LMath were written originally in Fortran, and this inheritance makes us to use quasi 1-based arrays.

Similar procedures performing element-wise operations with a float and matrix:
```
function MatFloatAdd(M:TMatrix; R:Float; Lb, Ub1, Ub2 : integer;
         Ziel : TMatrix = nil) : TMatrix;
function MatFloatSubstr(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;
```

Next set of functions performs element-wise operations over two vectors.
Add or subtract:
```
function VecAdd(V1,V2:TVector; Ziel : TVector = nil): TVector;
function VecSubstr(V1,V2:TVector; Ziel : TVector = nil): TVector;
```

Multiply and divide:
```
function VecElemMul(V1,V2:TVector; Ziel : TVector = nil): TVector;
function VecDiv(V1,V2:TVector; Ziel : TVector = nil): TVector;
```

Dot product, outer product and cross product of two vectors:
```
function VecDotProd(V1,V2:TVector; Lb, Ub : integer) : float;
function VecOuterProd(V1, V2:TVector; Lb, Ub1, Ub2 : integer;
          Ziel : TMatrix = nil):TMatrix;
function VecCrossProd(V1, V2:TVector; Lb: integer;
          Ziel :TVector = nil):TVector;
```

Euclidean length of a vector (dot product with itself):
```
function VecEucLength(V:TVector; LB, Ub :  integer) :  float;
```
Matrix and vector product:

```
function MatVecMul(M:TMatrix; V:TVector; LB: integer;
         Ziel: TVector = nil): TVector;
```

Multiplication of matrices:
```
function MatMul(A, B : TMatrix; LB : integer;
         Ziel : TMatrix = nil) : TMatrix;
```

Transposition of a matrix. First one places a result in `Ziel` which must have compatible dimensions or be `nil`, second one transposes a square matrix in place.
```
function MatTranspose(M:TMatrix; LB: integer;
         Ziel: TMatrix = nil): TMatrix;
procedure MatTransposeInPlace(M:TMatrix; Lb, Ub : integer);
```

Operators `+,-,*,/` are defined for `TVector` and `Float` and for `TMatrix` and `Float` (in this order), as well as `+,-` for `TVector` and `TVector`, which are element-wise operations. They are done for all elements of a vector beginning from index 0,

result is placed in a new vector or matrix of the same dimensions. For vector and vector operations both should have the same length.

15 Statistics

15.1 uMeanSD_MD: statistics with missing values

Unit `uMeanSD_MD` implements simple descriptive statistics for datasets with missing values. Missing values are represented by default as NAN (Not A Number). Alternatively, a user can define a custom missing value code using
`procedure SetMD(aMD:float);`

Test if a value is undefined (NAN or missing code) with
`function Undefined(F:Float):boolean;`

`function ValidN(X:TVector; Lb, Ub:Integer):integer;`
returns number of valid (defined) elements of an array `X` and
`function FirstDefined(X:TVector; Lb,Ub:Integer):integer;`
returns an index of a first defined element of an array.

Following statistical functions are implemented:
`function Max(X : TVector; Lb, Ub :  Integer) :  Float;`
Returns maximum of sample X.

`function Mean(X : TVector; Lb, Ub :  Integer) :  Float;`
Returns mean of sample X.

`function StDev(X : TVector; Lb, Ub :  Integer) :  Float;`
Returns Standard deviation estimated from sample X.

`function StDevP(X : TVector; Lb, Ub :  Integer) :  Float;`
Returns Standard deviation of population.

15.2 DGaussian in uNormal unit

`function DGaussian(X, Mean, Sigma: float) : float;`
added. It returns probability density function of a normal distribution with give mathematical expectation (`mean`) and standard deviation (`sigma`).

16 Optimization

Two important optimization algorithms were added to the library in LMath ver 0.5. First is COBYLA (Constrained optimization by linear approximation) developed initially by Michael J. D. Powell. For LMath it was adapted from Fortran 77 by Viatcheslav V. Nesterov. Second is simplex method for linear programming, adapted from *Numeric Recipes in Fortran 77*.

16.1 Linear Programming

Second important addition to `lmOptimum` package is the `uLinSimplex` unit which contains procedures for solving linear programming problems. Task of linear programming is a maximization of a linear function of several variables, subject to several equality and inequality constrains.

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

is the user-level linear programming solver. Input: `N` is number of variables to be optimized, `M1, M2, M3` are number of constrains in $\leq, \geq$ and $=$ form, respectively. Matrix `A[M+2,N+1]` contains a tableau with the coefficients of the objective function and constrains. Output: iCase is the outcome of calculation, 0 means that finite solution was found, 1: objective function is unbounded, -1: no solution exists; FuncVal is value of the function upon optimization, and SolVector contains optimal values of argument variables. Theory and details of the programming are described in the Reference guide, section uLinSimplex.

Example program is located at:
demo/gui/LinProg/

16.2 COBYLA

Sometimes it is important to minimize or maximize an objective function subject to one or more equality and inequality constrains. To be able to solve this type of problems with LMath, I implemented COBYLA algorithm. As a bonus, it 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 non-negative.

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

N is number of variables to optimize, residing in `X[N]` array.
M Number of inequality constrains.
X An array of variables to be optimized. Guess values before call, optimized after. As with many optimization algorithms, it is advantageous to have realistic guess values.
F The objective function value upon minimization.
MaxCV Maximal constraint violation, ideally should be zero.
RhoBeg A magnitude of initial change of the optimized variables. This value must be set by a user to a reasonable value. COBYLA uses an iterative process with a linear approximation of the objective function at each iteration. Since linear approximation works well only in relatively small intervals, `RhoBeg` should not be too big, especially if an objective function has a complex form. However, too little `RhoBeg` leads to inefficient calculations. Value of `RhoBeg` depends also of a level of uncertainty of initial values. There are no strict rules for finding an optimal `RhoBeg` and it may be matter of experimentation to find one.
RhoEnd A desired precision of objective function and constrain satisfaction. Like `RhoBeg`, this value must be reasonably set by a user.
MaxFun At input is a limit on the number of calls of `CALCFC` user-supplied function, to avoid endless looping, at the end number of actual calls.

CalcFC The objective function.

User-defined function must be of `TCobylaObjectProc` type which is defined in `uTypes` as follows:

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

N Number of arguments to be adjusted;
M Number of constraints;
X Vector of length N+1, current vector of variables.
Con Vector of length [M+3], vector of constraint values.

The subroutine should return the objective function in F, constrain values in CON[1], CON[2], ...,CON[M]. 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 non-negative. Importantly, constraints can be violated during the calculation!

Example programs of using `COBYLA` are
demo/console/optim/TestCobyla/testcobyla.lpr
and testfunc.pas. Sample output of the program is in the file outputcobyla.txt.

17 Regression and interpolation

17.1 Constrained non-linear regression

Unit uConstrNlFit uses COBYLA algorithm (See 16.2) to solve problems of non-linear regression with constrains. As in all non-linear regression procedures of DMath/LMath, a user must supply an objective function to be fitted of type `TRegFunc`, defined in `uTypes` as `function(X : Float; B : TVector) : Float;`. `X` is an independent variable, `B` is vector of regression parameters. Besides, for constrained regression, a procedure for calculation of constrain expressions must be supplied, of

```
type
  TConstrainsProc = procedure(MaxCon: integer; B, Con : TVector);
```

`MaxCon` is the number of constrains, `B` is vector of parameters. `B` and `Con` are allocated by calling procedure. Results of constrain calculation must be placed into `Con` beginning from `Con[1]`. Con[0] is not used because of Fortran inheritance. The fitting procedure finds such regression parameters B that constrain expressions are non-negative. However, during the calculation of fit, constrains may be violated and one should not rely on their non-negativity. `RegFunc : TRegFunc` and `ConstProc : TConstrainsProc` are passed to

```
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);
```

Other parameters are: `X` and `Y`, data for the regression;
`Lb` and `Ub`, bounds of X and Y arrays;
`MaxFun`, on input: maximal number of calls to objective function, to avoid infinite looping, which may happen with some unfortunate sets of parameters; on output:

actual number of calls to objective function.
`Tol` is tolerance of fit (RhoEnd in COBYLA algorithm);
`B` is vector of parameters. `B` must be allocated by a calling procedure and contain n input guess values. On output `B` contains fitted values.
`LastPar` is a number of parameters in B. First parameter is placed in b[1], last in B[LastParam].
`LastCon` is the number of constrains. It must be equal to the number of constrain expressions in `ConstProc`. `ConstrNlFit` allocates and passes to `ConstrNlFit Con` vector with the length LastCon+1.
`MaxCV` on input is a maximal acceptable constrain violation; on output it is an actual violation upon the fit.

Unit uConstrNLFit defines also variable `RhoBeg`, which is passed to COBYLA algorithm, used by `ConstrNLFit`. See details in the section COBYLA (16.2). Default value is 1. Adjust it to the scale of your variables before call of `ConstrNLFit`.

Afterwards, revise your model with the procedures

```
function GetCFResiduals: TVector;
function GetCFFittedData: TVector;
```

First of them returns theoretical values corresponding to observed points supplied in `X` and `Y` vectors passed to `ConstrNLFit`. Second one returns corresponding residuals: Y(calculated) - Y(observed).

Example program for uConstrNlFit is:

`LMath/demo/gui/ConstrFit/constr_fit.lpr`

17.2 Spline

Unit uSpline defines procedures for interpolation of data with natural cubic spline, finding derivative and extremums of resulting spline function. There are two stages.

First, use the

```
procedure InitSpline(Xv, Yv:TVector; var Ydv:TVector;
    Lb,Ub:integer);
```

to prepare spline data in `Vdv` vector. X and Y are vectors of the data to be interpolated; Lb and Ub are, as usually, lower and upper indices, they must be equal for X and Y. `Ydv` after the call to `InitSpline` has length `Ub+1`. After `InitSpline` use

```
function SplInt(X:Float; Xv, Yv, Ydv: TVector;
    Lb,Ub:integer):Float;
```

to get value of the spline function at a point `X`. All other parameters are same as were used for a call to `InitSpline`. Next two functions are intended for investigation of a resulting spline. Both must be called after `InitSpline`.

```
function SplDeriv(X:Float; Xv, Yv, Ydv: TVector;
   Lb, Ub:integer):float;
```

has same parameters as `Splint` and returns a derivative of the spline function in a given point X.

```
procedure FindSplineExtremums(Xv,Yv,Ydv:TVector; Lb,Ub:integer;
    out Minima, Maxima:TRealPointVector;
       out NMin, NMax: integer; ResLB:integer = 1);
```

finds minima and maxima of the spline function. `Xv, Yv, Ydv, Lb` and `Ub` are same as before. Minima and Maxima arrays are output variables which contain coordinates of every Minimum and Maximum, argument in X and value in Y. Both these arrays are allocated by the `FindSplineExtremums`. Indexing of `Minima` and `Maxima` begins from `ResLb`, default value is 1. `NMin` and `NMax` contain number of minima and maxima found, respectively.

Example using `uSpline` is located at:
/demo/gui/Spline/.

18 Special Regression Models

Package **lmSpecregress** contains several special regression models, predominantly specific for chemistry and biology and few statistical distributions.

18.1 Distributions

Unit **uDistribs** defines several distributions and instruments to model experimental data with these distributions. Defined are binomial, exponential, hypoexponential and hyperexponential distributions.

Binomial distribution

`Function dBinom(k,n:integer;q:Float):Float;`
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.

Exponential distribution

`Function ExponentialDistribution(beta, X:float):float;`
evaluates exponential probability density

$$pdf(x;\beta) = \begin{cases} \frac{1}{\beta} e^{-\frac{x}{\beta}} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{1}$$

with time constant β = beta for given X. If $X < 0$ returns 0.

Hyperexponential distribution

Hyperexponential distribution arises when several concurrent processes with different kinetics are going. In electrophysiology it may be, for example, when an ion channel has two independent closed states with different time constants and can enter each of these states with a probabilities P1, P2, etc. Variation coefficient CV of such distribution is always greater than 1, hence the name "hyperexponential".

```
Function HyperExponentialDistribution(N:integer;
    var Params:TVector; X:float):float;
```
evaluates hyperexponential distribution

$$pdf(\beta_i, p_i, x) = \begin{cases} \sum_{i=1}^{n} \frac{p_i}{\beta_i} e^{-\frac{x}{\beta_i}} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{2}$$

N defines number of phases; Params: zero-based TVector[2*N] containing pairs of parameters for each phase: probability and time constant β (see Equation 2). Sum of all probabilities must be 1.

```
procedure Fit2HyperExponents(var Xs, Ys:TVector;
   Ub:integer; var P1, beta1, beta2:float);
```

Fits a hyperexponential distribution with 2 phases. Input parameters: `Xs` and `Ys` are 1-based vectors with a density table of a hyperexponentially distributed random variable; `Ub` is their upper bound (number of (X;Y) pairs). Output: `P1` is a probability of the first phase; `beta1` and `beta2` are time constants for the first and second phases respectively. Probability of the second phase is 1 − P1.

Hypoexponential distribution

Hypoexponential distribution arises when there are several sequential processes, each with own rate constant. Hypoexponential distribution describes a distribution of time needed to reach end state in a process which passes sequentially several stages each with its own time constant β. Equation for probability density of the hypoexponential distribution with two phases has the form:

$$pdf(\beta_1, \beta_2, x) = \begin{cases} \frac{e^{-x/\beta_1} - e^{-x/\beta_2}}{1/\beta_1 - 1/\beta_2} & x \geq 0 \\ 0 & x < 0 \end{cases} \tag{3}$$

For hypoexponential distribution,

$$CV < 1$$

hence the name of the distribution.

`function HypoExponentialDistribution2(beta1, beta2, X:float): float;` returns value of the hypoexponential distribution with the parameters `beta1` and `beta2` for value X. LMath offers two procedures to estimate parameters of a hypoexponential distribution with 2 phases.

```
procedure EstimateHypoExponentialDistribution(
      M,CV:float; out beta1, beta2:float);
```

finds parameters of the distribution from a known sample mean (M) and coefficient of the variation (CV).

```
procedure Fit2HypoExponents(var Xs, Ys:TVector; Ub:integer;
   out beta1, beta2:float);
```

fits an observed probability density with the hyperexponential distribution with the parameters `beta1` and `beta2`. Input parameters: `Xs` and `Ys` contain density table of the distribution.

The easiest way to investigate primary data with the help of `Fit2HyperExponents` and `Fit2HypoExponents` is to classify entries of the population into binned histogram using `Distrib` function and extracting `Xs` `Ys` vectors from `TStatClassVector` returned by `Distrib` with `distExtractX` and `DistExtractD`. All these procedures are located in `uDistrib` unit, see Reference guide, 7.4. Programming example is in

`LMath\demo\gui\distribs\distribs.lpr`

Sum of gaussians

Sometimes, notably in chromatography or in electrophysiology, one meets a situation when a random value is distributed as a sum of several gaussians with different probabilities:

$$\begin{cases} pdf(x) = p_0\,\mathcal{N}(x,\mu_0,\sigma_0) + \sum\limits_{i=1}^{n-1} p_i\,\mathcal{N}(x,\mu_i,\sigma) \\ \sum\limits_{i=0}^{n-1} p_i = 1 \end{cases} \tag{4}$$

where n is number of gaussians in the distribution, $p_0..p_{n-1}$ are probabilities of gaussians, $\mathcal{N}$ is gaussian distribution:

$$\mathcal{N}(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}, \tag{5}$$

$\mu_0..\mu_{n-1}$ are mathematical expectations and σ_0 and σ are dispersions of $\mathcal{N}_0$ and of $\mathcal{N}_1..\mathcal{N}_{n-1}$ correspondingly.

In some cases, standard deviations (σ) of all gaussians are equal ($\sigma_0 = \sigma$) in Equation 4, in other special cases, it may be different in one case: $\sigma_0 \neq \sigma$. It happens, for example, in patch-clamp for "all closed" level. Finally, sometimes intervals between peaks of gaussians may be different, and in some times, again in patch-clamp, they must be equal:

$$\mu_{i+1} - \mu_i = \delta_\mu = const \tag{6}$$

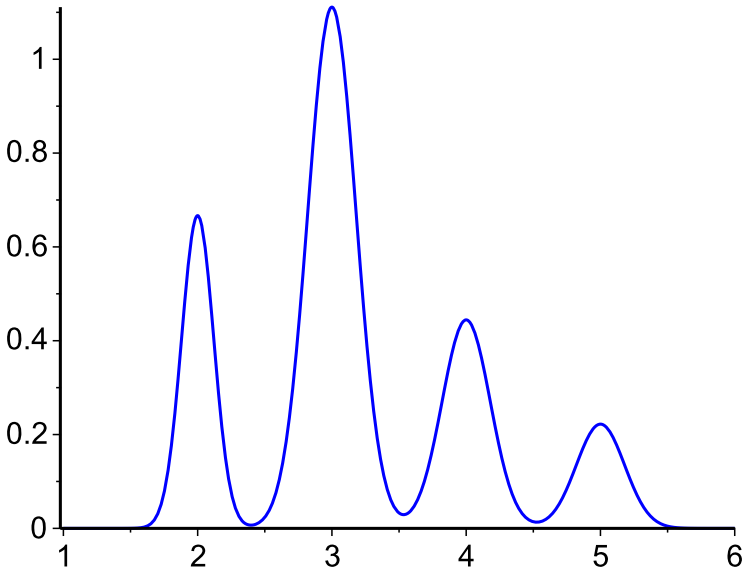

Figure 1 – Composition (sum) of gaussian distributions

These models are defined and fitted in the units `uGauss`, for variable interval between gaussians, and `uGaussF`, for a constant one. It must be noted, however, that these models do not enforce a limitation that sum of probabilities for all gaussians is 1, therefore equivalent of probabilities is called merely scaling factors (ScF). Figure 1 shows such distribution with $\mu_0 = 2$, $\mu_1 = 3$, $\mu_2 = 4$ and $\mu_3 = 5; \sigma_0 = 0.12; \sigma = 0.18; \{ScF_0..ScF_3\} = \{0.2, 0.5, 0.2, 0.1\}$.

This model was written initially for analysis of single-channel data in patch-clamp, where first gaussian typically corresponds to a situation "all channels closed", second to one open channel etc., that is why gaussians are counted from "0".

Unit uGauss, variable interval between peaks This unit implements following procedures and functions.

```
function ScaledGaussian(mu, sigma, ScF, X:float):float;
```

Returns value of probability density function of a gaussian with a mathematical expectation `mu` and standard deviation `sigma`, scaled by factor `ScF`, in a point `X`. For example, if a model includes a sum of three gaussians with probabilities 30%, 50% and 20%, ScF should be 0.3, 0.5 and 0.2, correspondingly, such that integral of the whole sum is 1. depending on a quality of actual data and presence of points which are not explained by a multigaussian model, actual sum of scaling factors may be slightly different from 1.

```
function SumGaussians(X:Float; Params:TVector):float;
```
evaluates a sum of N gaussians with all $\sigma_0..\sigma_N$ equal.

X independent variable;
Params[1] σ;
Params[2]..Params[NumberOfGaussians+1] ScF_i (scaling factors);
Params[NumberOfGaussians+2..2*NumberOfGaussians+2] μ_i.

`function SumGaussiansS0(X:Float; Params:TVector):float;`
is similar to `SumGaussians`, but returns sum of gaussians where σ_0 is different from others. Consequently, `Params` are:

Params[1] σ_0
Params[2] σ for gaussians 1..n, n is number of gaussians

```
procedure SetGaussFit(ANumberOfGaussians:integer;
   AUseSigma0, AFitMeans: boolean);
```
Set model parameters. This procedure must be called before `SumGaussFit`.
Parameters:
ANumberOfGaussians Number of gaussians which form the distribution, $n+1$ where n is index of the last gaussian.
AUseSigma0 Should σ_0 in Equation 4 be different from others?
AFitMeans Should μ_i be fitted or they are fixed and only σ_i and scale factors ScS_i are fitted.

```
procedure SumGaussFit(var AMathExpect: TVector;
   var ASigma, ASigma0:Float;
   var ScFs : TVector; const AXV, AYV:TVector; Observ:integer);
```

This procedure does actual fit of the model. Parameters:

AMathExpect 0-based TVector; as input, guess values for means; as output, fitted means;
ASigma, ASigma0 on input guessed and on output fitted Sigma for all gaussians and, if needed, separately σ_0;
ScFs 0-based TVector, as input guess values and as output fitted scaling factors;
AXV, AYV 1-based TVector containing experimental data for X and for Y (observed probability distribution density);
Observ number of observations (High bound of AXV and AYV).

Unit uGaussF. Fixed interval between peaks

`function SumGaussiansF(X:Float; Params:TVector):float;`

and

`function SumGaussiansFS0(X:Float; Params:TVector):float;`

are similar to `SumGaussians` and `SumGaussiansS0` from `uGauss` unit, they return value of probability density function for given `X`, while `Params` contain parameters of the fitted distribution.

For `SumGaussiansF`:

Params[1] σ;
Params[2]..Params[NumberOfGaussians+1] ScF (scaling factors)
Params[NumberOfGaussians+2] μ_0
Params[NumberOfGaussians+3] $\delta = \mu_i - \mu_{i-1}$ distance for $i \in [1..n]$

For `SumGaussiansFS0`:

Params[1] σ_0;
Params[2] σ;

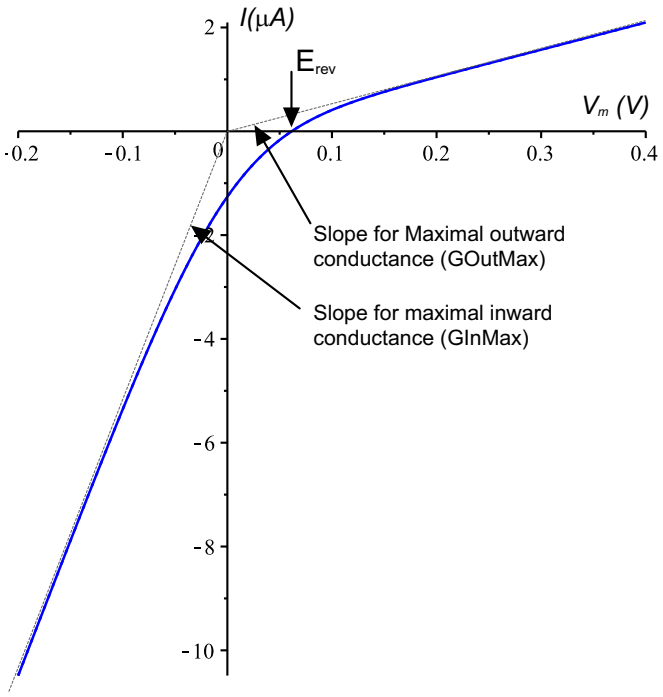

Figure 2 – Goldman-Hodgkin-Katz curve for monovalent cation, $C_E/C_I = 10$. Note none-zero current at zero voltage and crossing of "X" axis at $E_{rev} = 0,061\,V$. Note also that the curve asymptotically approaches linear Ohmic I-V relation when voltage becomes large, but Ohmic conductances differ for plus $+\infty$ or $-\infty$.

Params[3]..Params[NumberOfGaussians+2] ScF_i (Scaling factors);
Params[NumberOfGaussians+3] μ_0;
Params[NumberOfGaussians+4] δ.

```
procedure SetGaussFitF(ANumberOfGaussians:integer;
  AUseSigma0:boolean);
```

and

```
procedure DeltaFitGaussians(var ASigma, ASigma0, ADelta, AMu0: Float;
   var ScFs: TVector; const AXV, AYV: TVector; Observ: integer);
```

are similar to SetGaussFit and SumGaussFit.
codeAMu0 and `ADelta` on input contain guess values for μ_0 and for $\delta = \mu_i - \mu_{i-1}$, on output, fitted values. Other parameters are same as for `SetGaussFit` and `SumGaussFit`.

unit uGoldman. Membrane transport and Goldman-Hodgkin-Katz equation

If two solutions are separated by a membrane selectively permeable for a certain ion, and concentrations of this ion in these two solutions differ, then dependence of an electric current through a square unit of the membrane from voltage follows the Goldman-Hodgkin-Katz equation for current density:

$$I_s = P_s \frac{z^2 F^2 V_m}{RT} \cdot \frac{C_i - C_e\, \mathrm{e}^{-zFV_m/RT}}{1 - \mathrm{e}^{-zFV_m/RT}} \tag{7}$$

Here (P_s) is a specific membrane permeability,

$$\frac{mol/(m^2 \cdot s)}{mol/m^3} = m/s,$$

z is an ion valence, V_m is voltage, C_i and C_e are concentrations of the ion in "E" and "I" solutions (for example, extra- and intracellular, respectively), and T is temperature in Kelvins. Other values are Faraday constant (F) and gas constant (R). If a square area of a membrane is not known, equation for current may be used:

$$I = P\frac{z^2F^2V_m}{RT} \cdot \frac{C_i - C_e\,\mathrm{e}^{-zFV_m/RT}}{1 - \mathrm{e}^{-zFV_m/RT}} \tag{8}$$

where P (instead of P_s) is permeability constant, m^3/s and defines current through the whole membrane regardless of its surface area and dependent variable is current,Amp, rather then current density, Amp/m^2. Otherwise these equations are identical and `uGoldman` unit does not distinguish them.

Graphic presentation of this dependence for a monovalent cation with $C_{in} = 10\,C_{out}$ is shown in the Figure 2. Note, that at zero voltage current is non-zero and crosses "x" axis at a reversal potential E_{rev} which can be found with Nernst equation:

$$E_{rev} = \frac{RT}{zF}\ln\left(\frac{C_e}{C_i}\right) \tag{9}$$

Depending of experimental conditions, C_{in} may be controlled by an experimenter or unknown. Correspondingly, `uGoldman` unit can fit Goldman-Hodgkin-Katz equation for fixed C_{in} when reversal potential E_{rev} is fixed, or, in case of unknown C_{in}, E_{rev} is free and C_{in} is found by fitting procedure.

```
function GHK(P, z, Cin, Cout, Vm, TC: float):float;
```

evaluates Goldman-Hodgkin-Katz equation for given voltage V_m in mV. P is permeability constant m^3/s or specific permeability constant; m/s, z is ion valence; C_{in} and C_{out} are intracellular and extracellular concentrations of the ion, M/m^3 or mM/l; V_m is transmembrane voltage, mV; T_C is temperature, ℃. Returned value is Current, Amp, or current density, Amp/m^2.

```
procedure FitGHK(CinFixed: boolean; az, aCout, aTC: float;
  var Cin, P: float; Voltages, Currents: TVector; Lb, Ub: integer);
```

fits Goldman-Hodgkin-Katz equation from voltage and current data.
`CinFixed`: if C_{in} is known and fixed (true) or must be fitted (false); `az`: ion valence. It is float because some ions can exist and permeate a membrane in two different states, for example $H_2PO_4^-$ and HPO_4^{2-};`aCout` is the extracellular (outside) concentration of the ion; `aTC` is temperature, ℃. `Cin` is the intracellular concentration. If it must be fitted, guess value on input, found value on output, otherwise just known value as input; `P`, output only, is permeability, initial guess value is calculated within the fitting procedure with the function `PFromSlope`, see below; `Voltages` and `Currents` are experimental data of IV plot; `Lb` and `Ub` are their low and upper bounds.

Calculation of permeability P from the slope of IV plot; approximate value for non-linear IV characteristics and exact for linear:

`function PfromSlope(dI,dV,z,C,TC:float):float;`

Parameters: `dI` and `dV` are δ_I and δ_V which give the slope; `z`, `C` and `TC` are valence, concentration (mM/l) and temperature (℃). Several functions may be used to explore Goldman-Hodgkin-Katz dependence after the fit.

`function GOutMax(P,TC,Cin,Cout,z:float):float;`
`function GInMax(P,TC,Cin,Cout,z:float):float;`

These two functions find maximal conductance for outward current:

$$GOutMax = \lim_{V_m \to +\infty} G_s$$

and for inward current:

$$GInMax = \lim_{V_m \to -\infty} G_s,$$

see also Figure 2.

Parameters are, as usually, `P`: permeability; TC: temperature; `Cin and Cout`: concentrations; z: valence;

`function ERev(CIn, COut, z, TC:float):float;`

finds a reverse potential by Nernst equation, mV, see Equation 9.

`function Intracellular(Cout, z, TC, ERev:float):float;`

conversly, finds intracellular concentration from Cout, valence, temperature (℃), ERev (mV). Finally,

`function GSlope(Cin,Cout,z,TC,Vm,P:float):float;`

finds a slope conductance ($^{dI}/_{dV}$) at any Vm, from known ion concentrations, valence, temperature, voltage and permeability.

19 Evaluation of expressions

Mechanisms of expression evaluation defined in `uEval` unit were extended in LMath compared to DMath. In DMath, only single chars could be names of variables and only 28 variables could be defined. LMath allows any alphanumeric strings beginning from letters and unlimited number of variables. Besides that, variable `ParsingError` was made public enabling a calling procedure to detect and handle an error. Exponentiation can be invoked both with ^ or ** operators. New or modified procedures and functions:

`procedure SetVariable(VarName : String; Value : Float);`
Defines new variable and initializes it with `Value`.

`procedure DoneEval;`
Removes all functions and variables and frees memory.

Project eval: command line calculator can serve as an example program for `uEval` unit.

20 General utility functions

20.1 uSorting

Unit `uSorting` implements sorting of `TVector` and `TRealPointVector` with three algorithms: insertion sort, heap sort and quick sort. In a general case, Insertion sort is relatively ineffective and its use may be advisable in two cases: if an array is pretty small (ca 5 elements) or, and this is why we included it in the library, if the array is almost sorted from the beginning. Other two algorithms, heap and quick sort, are almost equally efficient, quick sort being slightly faster in general case, but being extremely slow if the array is almost ordered prior to the sorting, while heap sort has very similar time for best and worst cases, which makes it my personal favourite.

```
procedure QuickSort(Vector : TVector; Lb,Ub:integer; desc:boolean);
procedure QuickSortX(Points : TRealPointVector; Lb,Ub:integer;
  desc:boolean);
procedure QuickSortY(Points : TRealPointVector; Lb,Ub:integer;
  desc:boolean);

procedure InsertSort(Vector : TVector; Lb,Ub:integer; desc:boolean);
procedure InsertSortX(Points : TRealPointVector; Lb,Ub:integer;
  desc:boolean);
procedure InsertSortY(Points : TRealPointVector; Lb,Ub:integer;
  desc:boolean);

procedure Heapsort(Vector:TVector; Lb, Ub : integer; desc:boolean);
procedure HeapSortX(Points:TRealPointVector; Lb, Ub : integer;
  desc:boolean);
procedure HeapSortY(Points:TRealPointVector; Lb, Ub : integer;
  desc:boolean);
```

In all these procedures first argument (`Vector` or `Points`) is an array to be sorted; `Lb` and `Ub` are lower and upper bounds of subarray which is actually sorted; `desc` is a flag of descending sort. *SortX procedures sort `TRealPointVector` for X; *SortY procedures sort it for X.

Demo Program:

`\LMath\demo\gui\TestSort\`

20.2 uUnitsFormat

This unit implements

```
function FormatUnits(Val:float; UnitsStr:string; long:boolean=false):string;
```

which 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". If `long=True`, full form of prefix is used. Following prefixes are defined. Short form:
'a','f','p','n','μ','m','K','M','G','T','P','E';

long form:
'atto','femto','pico','nano','micro','milli','Kilo','Mega','Giga','Tera','Peta','Exa'
for
$10^{-18}, 10^{-15}, 10^{-12}, 10^{-9}, 10^{-6}, 10^{-3}, 10^{3}, 10^{6}, 10^{9}, 10^{12}, 10^{15}, 10^{18}$,
correspondingly.
Demo:

`\LMath\demo\console\MathUtil\testunitsformat`

21 LMComponents and signal processing

`LMcomponents` is an object-oriented extension of `LMath` library.

`TPoints` class from `lmPointsVec` unit is a class wrapper around `TRealPointVector` with several methods of data allocation and manipulation.

Unit `lmFilters` provides several algorithms of data filtering, namely, gaussian, moving average, and median filters which are implemented as non-visual components.

Besides, `TCoordSys` component from `lmCoordSys` unit serves for graphical representation of maths. It provides several procedures for conversion between screen and user space coordinates, primitives for line, circle and rectangle drawing in user coordinates, as well as of function graphic or spline drawings. For drawing of user's data, `OnDrawData` event is defined.

Finally, `lmNumericEdits` and `lmNumericInputDialogs` units contain `TFloatEdit` component and `Intervalquery`, `FloatInputDialog` and `IntegerInputDialog` functions.

See Documentation to lmComponents for details.