DMath

Math library for Delphi, FreePascal and Lazarus

Jean Debord

December 22, 2012

Contents

Chapter 1

Installation and compilation

1.1 Introduction

Welcome to DMath, a mathematical package for Delphi, FreePascal (FPC) and Lazarus. This chapter explains how to install this library and how to compile a program which uses it.

1.2 Installation

Extract the archive `dmat[...].zip` (where [...] stands for version number) in a given directory.

Be sure to preserve the directory structure. For instance, if you use `pkunzip`, add the option `-d` (i. e. `pkunzip -d dmat[...].zip`).

1.3 Compilation of programs

Programs using DMath can be compiled with Delphi under Windows and with FreePascal (FPC) or Lazarus under Windows and Linux.

Note: with FPC/Lazarus, the programs should be compiled in the Delphi mode (option `-Mdelphi`) to ensure that the `Integer` type is 32-bit (modify the FPC configuration file if necessary).

The `uses` clause must include all the DMath units needed by the program. Unit `utypes` is usually present, together with the units containing the routines called by the program. See file `filelist.txt` in the `units` subdirectory for a list of available units and procedures.

For instance, a program which uses the Gamma function (defined in unit `ugamma`) will have the following line:

```
uses utypes, ugamma;
```

The path to the DMath units must be specified in the configuration file or on the command line. For instance:

```
dcc32 prog.pas -cc -U\dmath\units
```

or:

```
fpc prog.pas -Mdelphi -Fu\dmath\units
```

assuming that you have installed the library in `\dmath`

1.4 Demo programs

Some demo programs are supplied with DMath. They can be classified into 3 groups:

1. Console programs, in the `demo\console` subdirectory: these are command-line programs which don't display graphics. They can be compiled with FPC, or with Delphi with the option `-cc`

2. BGI programs, in the `demo\bgi` subdirectory: these are command-line programs which display BGI graphics. These programs are for FPC only.

3. GUI programs, in the `demo\gui` subdirectory: these are graphic programs which have been designed with Delphi but should be compatible with Lazarus.

1.5 Use of DMath as a shared library

1.5.1 Introduction

DMath can be used as a shared library, such as a Windows DLL. Compilation scripts are supplied in the `dll` subdirectory to generate the shared library file, the interface file and (in the case of FPC/Lazarus) the object file. The following table displays the different possibilities:

Compiler / OS	Script	Shared lib.	Interface	Object
Delphi / Windows	`dcompil.bat`	`dmath.dll`	`dmath.dcu`	
FPC / Windows	`fpcompil.bat`	`dmath.dll`	`dmath.ppu`	`dmath.o`
Lazarus / Windows	`lcompil.bat`	`dmath.dll`	`dmath.ppu`	`dmath.o`
FPC / Linux	`fpcompil.sh`	`libdmath.so`	`dmath.ppu`	`dmath.o`
Lazarus / Linux	`lcompil.sh`	`libdmath.so`	`dmath.ppu`	`dmath.o`

Notes:

1. The only difference between the FPC and Lazarus versions concern the graphic library. With FPC it uses the Borland Graphics Interface (BGI) by means of the built-in `graph` unit, while with Lazarus it uses the graphic components (e. g. a `TImage` component).

2. BGI graphics will work under Linux only if `SVGAlib` is installed and functional.

3. Compilation with Lazarus may require the path to the Lazarus `Graphics` unit. This path may be added to the FPC configuration file.

4. Some FPC versions may generate an additional file: `libimpdmath.a` which is an import library.

1.5.2 Compilation of the library

In order to compile the library:

1. Run the appropriate script from the command line in the `dll` subdirectory.

2. Copy the shared library file to the appropriate directory, e. g. `\Windows\System` or `/usr/lib`

3. Copy the interface file, the object file and the import library (if any) to the directory where the compiler stores its units (or to any directory which is in the unit search path)

1.5.3 Using the shared library in a program

In order to use the shared library in a program, add the line:

```
uses dmath;
```

at the beginning of the program. The compilation can be done by:

```
    dcc32 prog.pas -cc
```

or:

```
    fpc prog.pas -Mdelphi
```

For the demo programs, the default method uses the individual units. However, it is possible to use the shared library by defining the symbol `USE_DLL`.

Chapter 2

Numeric precision

This chapter explains how to set the mathematical precision for the computations involving real numbers.

2.1 Numeric precision

DMath allows you to use three floating point types `Single` (4-byte real, about 6 significant digits), `Double` (8-byte real, about 15 significant digits), or `Extended` (10-byte real, about 18 significant digits).

The choice of a given type is done by defining a compilation symbol: `SINGLEREAL`, `DOUBLEREAL` or `EXTENDEDREAL`.

The symbol may be defined on the command line, using the `-d` option (e. g. `dcc32 prog.pas -dEXTENDEDREAL ...`) or in the IDE.

If no symbol is defined, then type `Double` will be automatically selected. It is therefore the default type.

If you wish to compare the results given by a DMath program with those of a reference program written in another language (e. g. Fortran), be sure that the two programs have been compiled with the same numeric precision.

2.2 Type Float

A type `Float` is defined in unit `utypes`. It corresponds to `Single`, `Double` or `Extended`, according to the compilation options.

So, a program which uses real variables should begin with something like:

```
uses
  utypes;
var
  X : Float;
```

2.3 Type Complex

For complex numbers, a `Complex` type is defined as follows:

```
type Complex = record
  X, Y : Float;
end;
```

2.4 Machine-dependent constants

The following constants are defined in unit `utypes` :

Constant	Meaning
`MachEp`	The smallest real number such that (`1.0 + MachEp`) has a different representation (in the computer memory) than 1.0; it may be viewed as a measure of the numeric precision which can be reached within the given floating point type.
`MaxNum`	The highest real number which can be represented.
`MinNum`	The lowest positive real number which can be represented.
`MaxLog`	The highest real number X for which `Exp(X)` can be computed without overflow.
`MinLog`	The lowest (negative) real number X for which `Exp(X)` can be computed without underflow.
`MaxFac`	The highest integer for which the factorial can be computed.
`MaxGam`	The highest real number for which the Gamma function can be computed.
`MaxLgm`	The highest real number for which the logarithm of the Gamma function can be computed.

2.5 Demo program

The program `testmach.pas` located in the `demo\console\fmath` subdirectory checks that the machine-dependent constants are correctly handled by the computer.

This program lists the sizes of the integer and floating point types, together with the values of the machine-dependent constants, and computes the following quantities:

`Exp(MinLog)`	Should be approximately equal to `MinNum`
`Ln(MinNum)`	Should be approximately equal to `MinLog`
`Exp(MaxLog)`	Should be approximately equal to `MaxNum`
`Ln(MaxNum)`	Should be approximately equal to `MaxLog`

`Fact(MaxFac)`
`Gamma(MaxGam)`
`LnGamma(MaxLgm)`
	Should be computed without overflow.

The following results were obtained with FPC 2.6.0 in double precision:

```
Integer type        = Integer  (4 bytes)
Long Integer type   = LongInt  (4 bytes)
Floating point type = Double   (8 bytes)
Complex type        = Complex  (16 bytes)

MachEp          =  2.2204460492503130E-0016

MinNum          =  2.2250738585072020E-0308
Exp(MinLog)     =  2.2250738585072152E-0308

MinLog          = -7.0839641853226410E+0002
Ln(MinNum)      = -7.0839641853226411E+0002

MaxNum          =  1.7976931348623150E+0308
Exp(MaxLog)     =  1.7976931348623216E+0308

MaxLog          =  7.0978271289338400E+0002
Ln(MaxNum)      =  7.0978271289338400E+0002

MaxFac          =  170
Fact(MaxFac)    =  7.25741561530800E+306
```

```
MaxGam          =  1.7162437695630200E+0002
Gamma(MaxGam)   =  1.79769313486231E+308

MaxLgm          =  2.5563480000000000E+0305
LnGamma(MaxLgm) =  1.79513667145944E+308
```

Chapter 3

Elementary functions

This chapter describes the mathematical constants and elementary mathematical functions available in DMath.

3.1 Constants

The following mathematical constants are defined in unit `utypes` :

Constant	Value	Meaning
`Pi`	3.14159...	π
`Ln2`	0.69314...	$\ln 2$
`Ln10`	2.30258...	$\ln 10$
`LnPi`	1.14472...	$\ln \pi$
`InvLn2`	1.44269...	$1/\ln 2$
`InvLn10`	0.43429...	$1/\ln 10$
`TwoPi`	6.28318...	2π
`PiDiv2`	1.57079...	$\pi/2$
`SqrtPi`	1.77245...	$\sqrt{\pi}$
`Sqrt2Pi`	2.50662...	$\sqrt{2\pi}$
`InvSqrt2Pi`	0.39894...	$1/\sqrt{2\pi}$
`LnSqrt2Pi`	0.91893...	$\ln \sqrt{2\pi}$
`Ln2PiDiv2`	0.91893...	$(\ln 2\pi)/2$
`Sqrt2`	1.41421...	$\sqrt{2}$
`Sqrt2Div2`	0.70710...	$\sqrt{2}/2$
`Gold`	1.61803...	Golden Ratio $= (1 + \sqrt{5})/2$
`CGold`	0.38196...	`2 - Gold`

Note : The constants are stored with 20 to 21 significant digits. So, they will match the highest degree of precision available (i.e. type `Extended`).

3.2 Error handling

The function `MathErr()` (also defined in `utypes`) returns the error code from the last function evaluation. It must be checked immediately after a function call:

```
Y := f(X);  { f is one of the functions of the library }
if MathErr = FOk then ...
```

If an error occurs, a default value is attributed to the function. The possible error codes are the following:

Error code	Value	Meaning
`FOk`	0	No error
`FDomain`	-1	Argument domain error
`FSing`	-2	Function singularity
`FOverflow`	-3	Overflow range error
`FUnderflow`	-4	Underflow range error
`FTLoss`	-5	Total loss of precision
`FPLoss`	-6	Partial loss of precision

3.3 Min, max, sign and exchange

The following functions are defined in unit `uminmax` :

- Function `FMin(X, Y)` returns the minimum of two real numbers X, Y.
- Function `IMin(X, Y)` returns the minimum of two integer numbers X, Y.
- Function `FMax(X, Y)` returns the maximum of two real numbers X, Y.
- Function `IMax(X, Y)` returns the maximum of two integer numbers X, Y.
- Function `Sgn(X)` returns 1 if $X \geq 0$, -1 if $X < 0$.
- Function `Sgn0(X)` returns 1 if $X > 0$, 0 if $X = 0$, -1 if $X < 0$.
- Function `DSgn(A, B)` transfers the sign of B to A. It is therefore equivalent to: `Sgn(B) * Abs(A)`
- Function `FSwap(X, Y)` exchanges two real numbers X, Y.
- Function `ISwap(X, Y)` exchanges two integer numbers X, Y.

3.4 Rounding functions

The following functions are defined in unit `uround` :

- Function `RoundN(X, N)` will round X to N decimal places. N must be between 0 and 16.
- Function `Floor(X)` returns the lowest integer $\geq X$
- Function `Ceil(X)` returns the highest integer $\leq X$

3.5 Logarithms and exponentials

These functions are defined in unit `umath`.

The functions `Expo` and `Log` may be used instead of the standard functions `Exp` and `Ln`, when it is necessary to check the range of the argument. The new function performs the required tests and calls the standard function if the argument is within the acceptable limits (for instance, X > 0 for `Ln(X)`); otherwise, the function returns a default value and `MathErr()` will return the appropriate error code.

Calling these functions is more time-consuming than calling the standard `Exp` and `Ln`, because each function involves several tests and two procedure calls (one to the function itself and another to the standard `Exp` or `Ln`). Hence, if the program must compute lots of logarithms or exponentials, it may be more efficient to use the standard functions `Exp` and `Ln`. In this case, however, the error handling must be done by the main program.

The same remark applies to the other logarithmic and exponential functions defined in the library:

Function	Definition	Pascal code
`Exp2(X)`	2^X	`Exp(X * Ln2)`
`Exp10(X)`	10^X	`Exp(X * Ln10)`
`Log2(X)`	$\log_2 X$	`Ln(X) * InvLn2`
`Log10(X)`	$\log_{10} X$	`Ln(X) * InvLn10`
`LogA(X, A)`	$\log_A X$	`Ln(X) / Ln(A)`

Here, too, it may be more efficient to use the Pascal code *inline* rather than calling the DMath function, but the error control will be lost.

3.6 Power functions

The following functions are also defined in unit `umath` :

- Function `Power(X, Y)` returns X^Y. Y may be integer or real, but if Y is real then X cannot be negative.
- Function `IntPower(X, N)` returns X^N where N is integer.

Note: To ensure the continuity of the function X^X when $X \to 0$, the value 0^0 has been set to 1.

3.7 Trigonometric functions

In addition to the standard Pascal functions `Sin`, `Cos` and `ArcTan`, DMath provides the following functions in unit `utrigo` :

Function	Definition
`Tan(X)`	$\frac{\sin X}{\cos X} \qquad X \neq (2k+1)\frac{\pi}{2}$
`ArcSin(X)`	$\arctan \frac{X}{\sqrt{1-X^2}} \qquad (-1 \leq X \leq 1)$
`ArcCos(X)`	$\frac{\pi}{2} - \arcsin X \qquad (-1 \leq X \leq 1)$
`ArcTan2(Y, X)`	$\arctan \frac{Y}{X}$, result in $[-\pi, \pi]$
`Pythag(X, Y)`	$\sqrt{X^2 + Y^2}$
`FixAngle(Theta)`	Returns the angle `Theta` in the range $[-\pi, \pi]$

Note: If (X, Y) are the cartesian coordinates of a point in the plane, its polar coordinates are:

```
R := Pythag(X, Y);
Theta := ArcTan2(Y, X)
```

3.8 Hyperbolic functions

The following functions are available in unit `uhyper` :

Function	Definition	
`Sinh(X)`	$\frac{1}{2}(e^X - e^{-X})$	
`Cosh(X)`	$\frac{1}{2}(e^X + e^{-X})$	
`Tanh(X)`	$\frac{\sinh X}{\cosh X}$	
`ArcSinh(X)`	$\ln(X + \sqrt{X^2 + 1})$	
`ArcCosh(X)`	$\ln(X + \sqrt{X^2 - 1})$	$X \geq 1$
`ArcTanh(X)`	$\frac{1}{2} \ln \frac{X+1}{X-1}$	$-1 < X < 1$

In addition, the subroutine `SinhCosh(X, SinhX, CoshX)` computes the hyperbolic sine and cosine simultaneously, saving the computation of one exponential.

3.9 Demo programs

These program are located in the `demo\console\fmath` subdirectory.

Function accuracy

Program `testfunc.pas` checks the accuracy of the elementary functions. For each function, 20 random arguments are picked, then the function is computed, the reciprocal function is applied to the result, and the relative error between this last result and the original argument is computed. This error should be around 10^{-15} in double precision.

Computation speed

Program `speed.pas` measures the execution time of the built-in mathematical functions, as well as the additional functions provided in `DMath`. The results are printed on the screen and saved in a text file named `speed.out`.

Chapter 4

Special functions

This chapter describes the special functions available in DMath. Most of them have been adapted from C codes in the Cephes library by S. Moshier (`http://www.moshier.net`).

4.1 Factorial

Function `Fact(N)`, defined in unit `ufact`, returns the factorial of the non-negative integer N, also noted $N!$:

$$N! = 1 \times 2 \times \cdots \times N \qquad 0! = 1$$

The constant `MaxFac` defines the highest integer for which the factorial can be computed (See chapter 2, p. 10).

4.2 Gamma function and related functions

4.2.1 Gamma function

The following functions are defined in unit `ugamma` :

- Function `Gamma(X)` returns the Gamma function, defined by:

$$\Gamma(X) = \int_0^\infty t^{X-1} e^{-t} dt$$

  This function is related to the factorial by:

$$N! = \Gamma(N+1)$$

The Gamma function is indefinite for $X = 0$ and for negative integer values of X. It is positive for $X > 0$. For $X < 0$ the Gamma function changes its sign whenever X crosses an integer value. More precisely, if X is an even negative integer, $\Gamma(X)$ is positive on the interval $]X, X+1[$, otherwise it is negative.

- Function `SgnGamma(X)` returns the sign of the Gamma function for a given value of X.
- Function `LnGamma(X)` returns the natural logarithm of the Gamma function.
- Function `Stirling(X)` approximates `Gamma(X)` with Stirling's formula, for $X \geq 30$.
- Function `StirLog(X)` approximates `LnGamma(X)` with Stirling's formula, for $X \geq 13$.

The constants `MaxGam` and `MaxLgm` define the highest values for which the Gamma function and its logarithm, respectively, can be computed (See chapter 2, p. 10).

4.2.2 Incomplete Gamma function

The following functions are defined in unit `uigamma` :

- Function `IGamma(A, X)` returns the incomplete Gamma function, defined by:

$$\frac{1}{\Gamma(A)}\int_0^X t^{A-1}e^{-t}dt \qquad A > 0, X > 0$$

- Function `JGamma(A, X)` returns the complement of the incomplete Gamma function, defined by:

$$\frac{1}{\Gamma(A)}\int_X^\infty t^{A-1}e^{-t}dt$$

  Although formally equivalent to `1.0 - IGamma(A, X)`, this function uses specific algorithms to minimize roundoff errors.

4.2.3 Inverse of incomplete Gamma function

Function `InvGamma(A, Y)`, defined in unit `uinvgam`, returns `X` such that `IGamma(A, X) = Y`

4.2.4 Polygamma functions

The polygamma function of order n, denoted $\psi_n(x)$, is the n-th derivative of the logarithm of the gamma function:

$$\psi_n(x) = \frac{d^n}{dx^n} \ln \Gamma(x)$$

The cases $n = 1$ and $n = 2$ are implemented in DMath as `DiGamma(X)` and `TriGamma(X)`. These functions are defined in unit `udigamma`.

4.3 Beta function and related functions

- Function `Beta(X, Y)`, defined in unit `ubeta`, returns the Beta function, defined by:

  $$\mathcal{B}(X,Y) = \int_0^1 t^{X-1}(1-t)^{Y-1} dt = \frac{\Gamma(X)\Gamma(Y)}{\Gamma(X+Y)}$$

  (Here $\mathcal{B}$ denotes the uppercase greek letter 'Beta' !)

- Function `IBeta(A, B, X)`, defined in unit `uibeta`, returns the incomplete Beta function, defined by:

  $$\frac{1}{\mathcal{B}(A,B)} \int_0^X t^{A-1}(1-t)^{B-1} dt \qquad A > 0, B > 0, 0 \leq X \leq 1$$

- Function `InvBeta(A, B, Y)`, defined in unit `uinvbeta`, returns `X` such that `IBeta(A, B, X) = Y`

4.4 Error function

The following functions are defined in unit `uigamma` :

- Function `Erf(X)` returns the error function, defined by:

  $$\mathrm{erf}(X) = \frac{2}{\sqrt{\pi}} \int_0^X \exp(-t^2) dt$$

- Function `Erfc(X)` returns the complement of the error function, defined by:

  $$\mathrm{erfc}(X) = \frac{2}{\sqrt{\pi}} \int_X^\infty \exp(-t^2) dt$$

4.5 Lambert's function

Lambert's W function is the reciprocal of the function xe^x. That is, if $y = W(x)$, then $x = ye^y$. Lambert's function is defined for $x \geq -1/e$, with $W(-1/e) = -1$. When $-1/e < x < 0$, the function has two values; the value $W(x) > -1$ defines the *upper branch*, the value $W(x) < -1$ defines the *lower branch*.

The function `LambertW(X, UBranch, Offset)`, defined in unit `ulambert`, computes Lambert's function.

- `X` is the argument of the function (must be $\geq -1/e$)
- `UBranch` is a boolean parameter which must be set to `True` for computing the upper branch of the function and to `False` for computing the lower branch.
- `Offset` is a boolean parameter indicating if `X` is an offset from $-1/e$. In this case, $W(X - 1/e)$ will be computed (with $X > 0$). Using offsets improves the accuracy of the computation if the argument is near $-1/e$.

The code for Lambert's function has been translated from a Fortran program written by Barry *et al* (`http://www.netlib.org/toms/743`).

4.6 Demo programs

These program are located in the `demo\console\fmath` subdirectory.

- Program `specfunc.pas`, checks the accuracy of the functions `Fact`, `Binomial`, `Gamma`, `IGamma`, `Erf`, `Erfc`, `Beta`, `IBeta`, `DiGamma` and `TriGamma`

  Most of the data come from *Numerical Recipes* (`http://www.nr.com`), but the reference values have been re-computed to 20 significant digits with the `Maple` software (`http://www.maplesoft.com`) and the Gamma values for negative arguments have been corrected.

  Each program computes the values of a given function for a set of predefined arguments and compares the results to the reference values. Then it displays the number of correct digits found. This number should be between 14 and 16 in double precision.

- Program `testw.pas` checks the accuracy of the Lambert function.

  The program computes Lambert's function for a set of pre-defined arguments and compares the results with reference values. It displays the number of exact digits found. This number should correspond with the numeric precision used (14-16 digits in double precision).

  This program has been translated from a Fortran program written by Barry *et al* (`http://www.netlib.org/toms/743`).

Chapter 5

Probability distributions

This chapter describes the functions available in DMath to compute probability distributions. Most of them are applications of the special functions studied in chapter 4.

5.1 Binomial distribution

Binomial distribution arises when a trial has two possible outcomes: ‘failure’ or ‘success’. If the trial is repeated N times, the random variable X is the number of successes.

- Function `Binomial(N,K)`, defined in unit `ubinom`, returns the binomial coefficient $\binom{N}{K}$, which is defined by:

$$\binom{N}{K} = \frac{N!}{K!(N-K)!} \qquad 0 \leq K \leq N$$

- Function `PBinom(N, P, K)`, also defined in `ubinom`, returns the probability of obtaining K successes among N repetitions, if the probability of success is P.

$$\text{Prob}(X = K) = \binom{N}{K} P^K Q^{N-K} \qquad \text{with } Q = 1 - P$$

- Function `FBinom(N, P, K)`, defined in unit `uibtdist`, returns the probability of obtaining at most K successes among N repetitions, i. e. $\text{Prob}(X \leq K)$. This is called the *cumulative probability function* and is defined by:

$$\text{Prob}(X \leq K) = \sum_{k=0}^{K} \binom{N}{k} P^k Q^{N-k} = 1 - I_{\mathcal{B}}(K+1, N-K, P)$$

where $I_{\mathcal{B}}$ denotes the incomplete Beta function.

The mean of the binomial distribution is $\mu = NP$, its variance is $\sigma^2 = NPQ$. The standard deviation is therefore $\sigma = \sqrt{NPQ}$.

5.2 Poisson distribution

The Poisson distribution can be considered as the limit of the binomial distribution when $N \to \infty$ and $P \to 0$ while the mean $\mu = NP$ remains small (say $N \geq 30$, $P \leq 0.1$, $NP \leq 10$)

- Function `PPoisson(Mu, K)`, defined in unit `upoidist`, returns the probability of observing the value K if the mean is μ. It is defined by:

$$\text{Prob}(X = K) = e^{-\mu} \, \frac{\mu^K}{K!}$$

- Function `FPoisson(Mu, K)`, defined in unit `uigmdist`, gives the cumulative probability function, defined by:

$$\text{Prob}(X \leq K) = \sum_{k=0}^{K} e^{-\mu} \, \frac{\mu^k}{k!} = 1 - I_\Gamma(K+1, \mu)$$

  where I_Γ denotes the incomplete Gamma function.

5.3 Standard normal distribution

The normal distribution (a. k. a. Gauss distribution or Laplace-Gauss distribution) corresponds to the classical bell-shaped curve. It may also be considered as a limit of the binomial distribution when N is sufficiently 'large' while P and Q are sufficiently different from 0 or 1. (say $N \geq 30$, $NP \geq 5$, $NQ \geq 5$).

The normal distribution with mean μ and standard deviation σ is denoted $\mathcal{N}(\mu, \sigma)$ with $\mu = NP$ and $\sigma = \sqrt{NPQ}$. The special case $\mathcal{N}(0, 1)$ is called the standard normal distribution.

- Function `DNorm(X)`, defined in unit `unormal`, returns the probability density of the standard normal distribution, defined by:

$$f(X) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{X^2}{2}\right)$$

  The graph of this function is the bell-shaped curve.

- Function `FNorm(X)`, defined in unit `uigmdist`, returns the cumulative probability function:

$$\Phi(X) = \text{Prob}(U \leq X) = \int_{-\infty}^{X} f(x)dx = \frac{1}{2}\left[1 + \text{erf}\left(X\frac{\sqrt{2}}{2}\right)\right]$$

  where U denotes the standard normal variable and erf the error function.

- Function `PNorm(X)`, also defined in `uigmdist`, returns the probability that the standard normal variable exceeds X in absolute value, i. e. $\text{Prob}(|U| > X)$.

- Function `InvNorm(P)`, defined in unit `uinvnorm`, returns the value X such that $\Phi(X) = P$.

5.4 Student's distribution

Student's distribution is widely used in Statistics, for instance to estimate the mean of a population from a sample taken from this population. The distribution depends on an integer parameter ν called the *number of degrees of freedom* (in the mean estimation problem, $\nu = n - 1$ where n is the number of individuals in the sample). When ν is large (say > 30) the Student distribution is approximately equal to the standard normal distribution.

- Function `DStudent(Nu, X)`, defined in unit `ugamdist`, returns the probability density of the Student distribution with `Nu` degrees of freedom, defined by:

$$f_\nu(X) = \frac{1}{\nu^{1/2}\,\mathcal{B}\left(\frac{\nu}{2}, \frac{1}{2}\right)} \cdot \left(1 + \frac{X^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

  where $\mathcal{B}$ denotes the Beta function.

- Function `FStudent(Nu, X)`, defined in unit `uibtdist`, returns the cumulative probability function:

$$\Phi_\nu(X) = \text{Prob}(t \leq X) = \int_{-\infty}^{X} f_\nu(x)dx = \begin{cases} I/2 & \text{if} X \leq 0 \\ 1 - I/2 & \text{if} X > 0 \end{cases}$$

  where t denotes the Student variable and $I = I_{\mathcal{B}}\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu+X^2}\right)$

- Function `PStudent(Nu, X)`, also defined in `uibtdist`, returns the probability that the Student variable t exceeds X in absolute value, i. e. $\text{Prob}(|t| > X)$.

- Function `InvStudent(Nu, P)`, defined in unit `uinvbeta`, returns the value X such that $\Phi_\nu(X) = P$.

5.5 Khi-2 distribution

The χ^2 distribution is a special case of the Gamma distribution (see below). It depends on an integer parameter ν which is the number of degrees of freedom.

- Function `DKhi2(Nu, X)`, defined in unit `ugamdist`, returns the probability density of the χ^2 distribution with `Nu` degrees of freedom, defined by:

$$f_\nu(X) = \frac{1}{2^{\frac{\nu}{2}}\,\Gamma\left(\frac{\nu}{2}\right)} \cdot X^{\frac{\nu}{2}-1} \cdot \exp\left(-\frac{X}{2}\right) \qquad (X > 0)$$

- Function `FKhi2(Nu, X)`, defined in unit `uigmdist`, returns the cumulative probability function:

$$\Phi_\nu(X) = \text{Prob}(\chi^2 \leq X) = \int_0^X f_\nu(x)dx = I_\Gamma\left(\frac{\nu}{2}, \frac{X}{2}\right)$$

  where I_Γ denotes the incomplete Gamma function.

- Function `PKhi2(Nu, X)`, also defined in `uigmdist`, returns the probability that the χ^2 variable exceeds X, i. e. $\text{Prob}(\chi^2 > X)$.

- Function `InvKhi2(Nu, P)`, defined in unit `uinvgam`, returns the value X such that $\Phi_\nu(X) = P$.

5.6 Snedecor's distribution

The Snedecor (or Fisher-Snedecor) distribution is used mainly to compare two variances. It depends on two integer parameters ν_1 and ν_2 which are the degrees of freedom associated with the variances.

- Function `DSnedecor(Nu1, Nu2, X)`, defined in unit `ugamdist`, returns the probability density of the Snedecor distribution with `Nu1` and `Nu2` degrees of freedom, defined by:

$$f_{\nu_1,\nu_2}(X) = \frac{1}{\mathcal{B}\left(\frac{\nu_1}{2}, \frac{\nu_2}{2}\right)} \cdot \left(\frac{\nu_1}{\nu_2}\right)^{\frac{\nu_1}{2}} \cdot X^{\frac{\nu_1}{2}-1} \cdot \left(1 + \frac{\nu_1}{\nu_2}X\right)^{-\frac{\nu_1+\nu_2}{2}} \qquad (X > 0)$$

- Function `FSnedecor(Nu1, Nu2, X)`, defined in unit `uibtdist`, returns the cumulative probability function:

  $$\Phi_{\nu_1,\nu_2}(X) = \mathrm{Prob}(F \leq X) = \int_0^X f_{\nu_1,\nu_2}(x)dx = 1 - I_{\mathcal{B}}\left(\frac{\nu_2}{2}, \frac{\nu_1}{2}, \frac{\nu_2}{\nu_2 + \nu_1 X}\right)$$

  where F denotes the Snedecor variable.

- Function `PSnedecor(Nu1, Nu2, X)`, also defined in `uibtdist`, returns the probability that the Snedecor variable F exceeds X, i. e. $\mathrm{Prob}(F > X)$.

- Function `InvSnedecor(Nu1, Nu2, P)`, defined in unit `uinvbeta`, returns the value X such that $\Phi_{\nu_1,\nu_2}(X) = P$.

5.7 Exponential distribution

The exponential distribution is used in many applications (radioactivity, chemical kinetics...). It depends on a positive real parameter A.

The following functions are defined in unit `uexpdist` :

- Function `DExpo(A, X)` returns the probability density of the exponential distribution with parameter `A`, defined by:

  $$f_A(X) = A \exp(-AX) \qquad (X > 0)$$

- Function `FExpo(A, X)` returns the cumulative probability function:

  $$\Phi_A(X) = \int_0^X f_A(x)dx = 1 - \exp(-AX)$$

5.8 Beta distribution

The Beta distribution is often used to describe the distribution of a random variable defined on the unit interval [0, 1]. It depends on two positive real parameters A and B.

- Function `DBeta(A, B, X)`, defined in unit `ugamdist`, returns the probability density of the Beta distribution with parameters `A` and `B`, defined by:

  $$f_{A,B}(X) = \frac{1}{\mathcal{B}(A,B)} \cdot X^{A-1} \cdot (1-X)^{B-1} \qquad (0 \leq X \leq 1)$$

- Function `FBeta(A, B, X)`, defined in unit `uibtdist`, returns the cumulative probability function:

$$\Phi_{A,B}(X) = \int_0^X f_{A,B}(x)dx = I_B(A, B, X)$$

5.9 Gamma distribution

The Gamma distribution is often used to describe the distribution of a random variable defined on the positive real axis. It depends on two positive real parameters A and B.

- Function `DGamma(A, B, X)`, defined in unit `ugamdist`, returns the probability density of the Gamma distribution with parameters `A` and `B`, defined by:

$$f_{A,B}(X) = \frac{B^A}{\Gamma(A)} \cdot X^{A-1} \cdot \exp(-BX) \qquad (X > 0)$$

- Function `FGamma(A, B, X)`, defined in unit `uigmdist`, returns the cumulative probability function:

$$\Phi_{A,B}(X) = \int_0^X f_{A,B}(x)dx = I_\Gamma(A, BX)$$

The χ^2 distribution is a special case of the Gamma distribution, with $A = \nu/2$ and $B = 1/2$.

5.10 Demo program

Program `binom.pas`, located in the `demo\console\proba` subdirectory, compares the cumulative probabilities of the binomial distribution, estimated by function `FBinom`, with the values obtained by summing up the individual probabilities.

Chapter 6

Expression evaluation

DMath provides a math parser for evaluating expressions at run time. The expression is a character string which may contain numbers, operators, parentheses, variables and functions as described below.

6.1 Numbers

Integers (32-bit, signed) and reals (`Float` type) must be entered in decimal. The exponential notation (e. g. `6.62E-34`) is not accepted. You must write `6.62 * 10^(- 34)`.

6.2 Operators

+	addition, unary plus
-	subtraction, unary minus
*	multiplication
/	division
\	integer division
%	modulus operator
^	exponentiation
>	shift bit right
<	shift bit left
&	bitwise AND
|	bitwise OR
$	bitwise XOR
!	bitwise NOT
@	bitwise IMP
=	bitwise EQV

Operator precedence

!	bitwise NOT (highest precedence, evaluated first)
&	bitwise AND
|	bitwise OR
$	bitwise XOR
=	bitwise EQV
@	bitwise IMP
^	exponentiation
+ -	unary plus, unary minus
* /	multiplication, division
\	integer division
%	modulus operator
<>	shift bit left, shift bit right
+ -	addition, subtraction (lowest precedence, evaluated last)

6.3 Parentheses

Parentheses can be used to override operator precedence, but within parentheses operator precedence is used.

6.4 Variables

There are 26 variables from A to Z

6.5 Functions

Function arguments must be enclosed within parentheses. These parentheses must be present even if there is no argument passed to the function.

There are 58 built-in functions available. Most of them have been described in the previous chapters. There are, however, some differences which are documented below.

- Standard functions :

  ```
  Abs(X), Sgn(X), Int(X), Sqrt(X)
  Exp(X), Ln(X), Log10(X), Log2(X)
  Fact(N), Binomial(N, K), Rnd()
  ```

  `Rnd()` returns a 32-bit real random number in [0, 1) from the 'Mersenne Twister' generator (see chapter 15).

- Trigonometric functions :

  ```
  Deg(X), Rad(X)
  Sin(X), Cos(X), Tan(X)
  ArcSin(X), ArcCos(X), ArcTan(X), ArcTan2(Y, X)
  ```

  `Deg(X)` and `Rad(X)` convert their argument to degrees and radians, respectively.

- Hyperbolic functions :

  ```
  Sinh(X), Cosh(X), Tanh(X)
  ArcSinh(X), ArcCosh(X), ArcTanh(X)
  ```

- Special functions :

  ```
  Gamma(X), IGamma(A, X)
  Beta(X, Y), IBeta(A, B, X)
  Erf(X), LambertW(X)
  ```

  `LambertW(X)` returns only the upper branch of the Lambert function.

- Probability functions :

  ```
  PBinom(N, P, K), FBinom(N, P, K)
  PPoisson(Mu, K), FPoisson(Mu, K)
  DExpo(A, X), FExpo(A, X)
  DGamma(A, B, X), FGamma(A, B, X)
  DBeta(A, B, X), FBeta(A, B, X)
  DNorm(X), FNorm(X), PNorm(X), InvNorm(P)
  DStudent(N, X), FStudent(N, X)
  PStudent(N, X), InvStudent(N, P)
  DKhi2(N, X), FKhi2(N, X), PKhi2(N, X), InvKhi2(N, P)
  DSnedecor(N1, N2, X), FSnedecor(N1, N2, X)
  PSnedecor(N1, N2, X), InvSnedecor(N1, N2, P)
  ```

6.6 Exported functions

These functions are defined in unit `ueval`.

6.6.1 InitEval

This function initializes the built-in functions and returns their number. It also re-initializes the random number generator. It is mandatory to call this function before using the other functions.

```
var
  N : Integer;
begin
  N := InitEval;
  Writeln(N, ' functions initialized');
  ...
```

6.6.2 Eval

This function evaluates an expression passed as a character string and returns its value.

```
var
  X : Float;
  S : String;
begin
  InitEval;
  Write('Enter an expression : ');
  Readln(S);
  X := Eval(S);
  Writeln(' = ', X);
  ...
```

6.6.3 SetVariable

This procedure assigns a value to a variable. The name of the variable is case-insensitive.

```
SetVariable('x', 1);
```

6.6.4 SetFunction

This procedure adds a new function to the parser. Up to 42 functions can be added, with a maximum of 26 arguments for each function. The function declaration must conform to the internal type:

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

where `ArgC` is the argument count, `TArgC` denotes the interval `1..26` and `ArgV` contains the argument values, from `ArgV[1]` to `ArgV[ArgC]`.

```
function Average(N : TArgC; X : TVector) : Float;
var
  I : Integer;
  S : Float;
begin
  S := 0.0;
  for I := 1 to N do
    S := S + X[I];
  Average := S / N;
end;

begin
  SetFunction('Average', Average);
  Writeln(Eval('Average(1, 2, 3)'));
  ....
```

6.7 Demo programs

6.7.1 Console programs

These programs are located in `demo\console\fmath`

- Program `eval1.pas` evaluates a mathematical expression entered at run time.
- Program `eval2.pas` demonstrates the math parser and shows how to add variables and functions from the calling program.

6.7.2 GUI programs

- Program `calc.dpr` (in `demo\gui\calc`) implements a scientific calculator with 4 variables.
- Program `probcalc.dpr` (in `demo\gui\probcalc`) implements a probability calculator.

Chapter 7

Graphic functions

7.1 Introduction

DMath provides some graphic routines for plotting curves. They are available in three versions :

- a BGI version, based on Borland's Graph unit. This version is provided in unit `uplot`. It can be used with FreePascal (FPC).

  The BGI drivers and fonts are not distributed with DMath. They must be obtained from one of the Borland compilers which are freely available on the Internet, for instance Turbo Pascal 7 : `http://pascal.developpez.com/compilateurs/tp7/`

- a Delphi / Lazarus version, provided in unit `uwinplot`.

- a LaTeX version, which uses the `pstricks` extension to allow the creation of PostScript files. This version is provided in unit `utexplot`.

In addition, routines for converting between the RGB and HSV color spaces are provided in unit `uhsvrgb`. They can be used with all versions.

Note : a shared library compiled from the scripts in the `dll` subdirectory will contain either the BGI or Delphi version, along with the LaTeX version.

7.2 BGI graphics

7.2.1 Initializing graphics

- Function `InitGraphics(Pilot, Mode, BGIPath)` will place the computer in the graphic mode defined by the parameters `Pilot` (which corresponds to the graphic driver) and `Mode`. `BGIPath` is a string defining the directory in which the BGI drivers and fonts are stored.

The function returns `True` if the initialization is successful.

Example : `InitGraphics(9, 2, 'C:\TP\BGI')` will select the VGA mode, 640 × 480 resolution with 16 colors (see the documentation of the Graph unit for a list of available modes).

- Procedure `SetWindow(X1, X2, Y1, Y2, GraphBorder)` defines the size of the graphic window.

  `X1, X2, Y1, Y2` are the window coordinates in %

  `GraphBorder` is a boolean parameter which must be set to `True` for plotting a border around the window.

  This function must be called *after* `InitGraphics`

  Example : `SetWindow(15, 85, 15, 85, True)`

7.2.2 Coordinate axes

DMath allows to plot curves in either linear or logarithmic coordinates. The type of scale may be selected by using the predefined symbols `LinScale` or `LogScale`

For each axis, the scale is specified by:

- `SetOxScale(Scale, OxMin, OxMax, OxStep)`
- `SetOyScale(Scale, OyMin, OyMax, OyStep)`

where `Scale` may be either `LinScale` or `LogScale`, and the other parameters define the bounds and step on the axis.

Default values are: linear scale from 0 to 1 with a step of 0.2. These values will be used automatically if the calls to `SetOxScale` or `SetOyScale` are omitted.

An automatic scale, suitable for plotting all the values in an array `X[Lb..Ub]`, may be determined by:

`AutoScale(X, Lb, Ub, Scale, XMin, XMax, XStep)`

where `Scale` defines the type of scale. The results are returned in `XMin, XMax, XStep`.

Note that, for a logarithmic scale, the bounds must be powers of 10 and the step is always 10.

Once the scales have been specified, the axes may be plotted with the statements `PlotOxAxis` and `PlotOyAxis`.

In addition, a grid may be plotted with `PlotGrid(Grid)` where `Grid` may be: `HorizGrid` (horizontal lines only), `VertiGrid` (vertical lines only) or `BothGrid` (horizontal and vertical lines).

The current scale parameters can be retrieved for each axis with the procedures:

- `GetOxScale(Scale, OxMin, OxMax, OxStep)`
- `GetOyScale(Scale, OyMin, OyMax, OyStep)`

7.2.3 Titles and fonts

The default titles are 'X' and 'Y' for the axes, and none for the main graph. This may be changed with statements:

- `SetOxTitle(Title)`
- `SetOyTitle(Title)`
- `SetGraphTitle(Title)`

where `Title` is the relevant string.

The current titles are returned by the corresponding functions: `GetOxTitle, GetOyTitle, GetGraphTitle`.

The fonts used to print the titles and axis labels are the BGI fonts (`*.chr` files). The default font is the small font. It can be changed with the following procedures:

`SetTitleFont(FontIndex, Width, Height)`	for main graph
`SetOxFont(FontIndex, Width, Height)`	for Ox axis
`SetOyFont(FontIndex, Width, Height)`	for Oy axis
`SetLgdFont(FontIndex, Width, Height)`	for curve legends

`FontIndex` is the index of the font, as specified by the Graph unit (e. g. 2 for the small font). `Width` and `Height` define the font size. In the case of the axes, these settings will affect both the titles and labels.

Plotting the axes will automatically print the titles and labels. For the main graph title, the statement `WriteGraphTitle` should be used.

7.2.4 Clipping

Procedure `SetClipping(Clip)`, where `Clip` is a boolean parameter, is used to decide if the subsequent graphics will be clipped to the boundaries of the current viewport (defined by `SetWindows`).

7.2.5 Curves

Curve properties

Each curve is comprised of:

- a set of points, defined by their coordinates (x_i, y_i)
- a line connecting the points

It is possible to plot the points only, the line only, or both.

The points have the following properties:

- `Symbol` : index of the symbol to be plotted, according to the following convention:

  0 : point (1 pixel)
  1 : solid circle
  2 : open circle
  3 : solid square
  4 : open square
  5 : solid triangle
  6 : open triangle
  7 : plus (+)
  8 : multiply (×)
  9 : star (∗)

- `Size` : size in pixels (will have no effect if `Symbol` = 0)
- `Color` : index of color, according to the current palette

The lines have the following properties:

- `Style` : line style, according to the Graph unit

0	none
1	solid
2	dotted
3	centered
4	dashed

- `Width` : line width, according to the Graph unit

1	normal
3	thick

- `Color` : index of color

The curves have two additional properties :

- a legend (30 characters max.)
- a step, which defines how many points will be plotted (plot 1 point every step points)

By default, 9 curves may be plotted, with the following default parameters:

- symbol = index of curve
- point size = 2
- line style = 1 (solid)
- line width = 1 (normal)
- legend = 'Y1', 'Y2', ...
- step = 1
- color set for a 16-color palette (VGA mode)

Curve index	Color index	Color
1	12	LightRed
2	14	Yellow
3	10	LightGreen
4	9	LightBlue
5	11	LightCyan
6	13	LightMagenta
7	4	Red
8	2	Green
9	1	Blue

The maximal number of curves may be changed with `SetMaxCurv(NCurv)` where `NCurv` is a number between 1 and 255. The current number of curves is returned by function `GetMaxCurv`.

The settings for the curve `CurvIndex` may be changed with the following procedures:

- `SetPointParam(CurvIndex, Symbol, Size, Color)`
- `SetLineParam(CurvIndex, Style, Width, Color)`
- `SetCurvLegend(CurvIndex, Legend)`
- `SetCurvStep(CurvIndex, Step)`

The current settings are retrieved by the two procedures:

- `GetPointParam(CurvIndex, Symbol, Size, Color)`
- `GetLineParam(CurvIndex, Style, Width, Color)`

and the two functions:

- `GetCurvLegend(CurvIndex)`
- `GetCurvStep(CurvIndex)`

Plotting curves

- Procedure `PlotCurve(X, Y, Lb, Ub, CurvIndex)` plots a curve defined by the point coordinates `X[Lb..Ub]`, `Y[Lb..Ub]`, according to the parameters of curve #`CurvIndex`. The coordinates are expressed in user units (not in pixels).
- `PlotCurveWithErrorBars(X, Y, S, Ns, Lb, Ub, CurvIndex)` plots the curve and adds a vertical error bar to each point. The individual errors (usually expressed as standard deviations) are stored in vector `S[Lb..Ub]`. `Ns` is an integer such that the total height of the bar is `2 * Ns * S[I]` (e. g. set `Ns` to 3 for plotting 6 standard deviations).
- Procedure `PlotPoint(Xp, Yp, CurvIndex)` plots a single point. Here the coordinates `Xp, Yp` must be in *pixels*. This procedure is mainly used internally by the two previous ones.

7.2.6 Plotting a function

Procedure `PlotFunc(Func, X1, X2, CurvIndex)` plots the graph of function `Func` from `X1` to `X2` according to the parameters of curve #`CurvIndex`.

`Func` is of type `TFunc` and is declared as:

```
function Func(X : Float) : Float;
```

7.2.7 Legends

- Procedure `WriteLegend(NCurv, ShowPoints, ShowLines)` draws a box at the right side of the screen, with the legends of the plotted curves. `NCurv` is the number of curves. `ShowPoints` and `ShowLines` are boolean parameters selecting which symbols are displayed to identify the curves.
- Procedure `WriteLegendSelect(NSelect, Select, ShowPoints, ShowLines)` writes the legends of the selected curves. `NSelect` is the number of selected curves. Their indices are stored in the integer vector `Select`. For instance, if you wish to plot the legends of curves #2, 5 and 10, set `NSelect = 3, Select[1] = 2, Select[2] = 5, Select[3] = 10`.

7.2.8 Contour plots

Procedure `ConRec(Nx, Ny, Nc, X, Y, Z, F)` generates a contour plot of a two-dimensional function $f(x, y)$. The algorithm is adapted from Paul Bourke (`http://paulbourke.net/papers/conrec/`)

`Nx` and `Ny` are the number of steps on Ox and Oy. `Nc` is the number of contour levels. The point coordinates (in pixels) are stored in vectors `X[0..Nx]` and `Y[0..Ny]`. The contour levels (in increasing order) are stored in vector `Z[0..(Nc - 1)]`. The function values are stored in matrix `F[0..Nx, 0..Ny]`, such that `F[I,J]` is the function value at point `(X[I], Y[J])`.

7.2.9 Coordinate conversion

Functions `Xpixel(X)` and `Ypixel(Y)` convert the user coordinates `X` and `Y` to the corresponding screen coordinates (pixels). Functions `Xuser(X)` and `Yuser(Y)` do the inverse.

7.2.10 Leaving graphics

Procedure `LeaveGraphics` quits the graphic mode and returns to the text mode.

7.3 Delphi graphics

In this section we will present the Delphi graphics with respect to their BGI counterparts. The main difference is the presence, in some functions, of an additional parameter `Canvas` which denotes the component on which the graphic is drawn. When present, this parameter is always the first one.

7.3.1 Initializing graphics

- Function `InitGraphics(Width, Height)` enters graphic mode. The parameters `Width` and `Height` refer to the canvas 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)
    ```

  The function returns `True` if the initialization is successful.

- Procedure `SetWindow` behaves as its BGI equivalent, except for the additional parameter `Canvas`.

  Example : `SetWindow(Image1.Canvas, 15, 85, 15, 85, True)`

7.3.2 Coordinate axes

- Procedures `AutoScale, SetOxScale, SetOyScale, GetOxScale, GetOyScale` are identical to their BGI counterparts.
- Procedures `PlotOxAxis, PlotOyAxis, PlotGrid` have the additional parameter `Canvas`.

7.3.3 Titles and fonts

- The fonts are specified by the property of the canvas. Hence, there is no need for specific functions in `DMath` to change them.
- Procedures `SetGraphTitle, SetOxTitle, SetOyTitle` and their corresponding functions `GetGraphTitle, GetOxTitle, GetOyTitle` are identical to their BGI counterparts.
- Procedure `WriteGraphTitle` has the additional parameter `Canvas`.

7.3.4 Curves

- The curves have the same properties than in the BGI version, except that `Style` is of type `TPenStyle` and `Color` is of type `TColor`.

  The default colors are as follows:

Curve index	Color
1	clRed
2	clGreen
3	clBlue
4	clFuchsia
5	clAqua
6	clLime
7	clNavy
8	clOlive
9	clPurple

- Procedures `SetMaxCurv, SetCurvLegend, SetCurvStep` and their corresponding functions `GetMaxCurv, GetCurvLegend, GetCurvStep` are identical to their BGI counterparts.
- Procedures `SetPointParam, SetLineParam, GetPointParam, GetLineParam` are identical to their BGI counterparts, except for the types of parameters `Style` and `Color`.
- Procedures `PlotCurve` and `PlotCurveWithErrorBars` have the additional parameter `Canvas`.
- Procedure `PlotPoint` has the additional parameter `Canvas`. Moreover, the point coordinates must be specified in user units instead of pixels.

7.3.5 Other functions

- Procedure `PlotFunc(Canvas, Func, Xmin, Xmax, Npt, CurvIndex)` has a specific parameter `Npt` which denotes the number of points to be plotted.
- Procedures `WriteLegend, WriteLegendSelect` and `ConRec` have the additional parameter `Canvas`.
- Functions `Xpixel, Ypixel, Xuser, Yuser` are identical to their BGI counterparts.

7.4 LaTeX graphics

Most graphic statements have a LaTeX counterpart with a name beginning with 'TeX', e.g. `TeX_InitGraphics`.

However, colors and fonts are not handled at this time. So, the resulting plot will be black and white and all labels will be printed with the default font.

7.4.1 Initializing graphics

- Function `TeX_InitGraphics(FileName, PgWidth, PgHeight, Header)` initializes the LaTeX file.
  - `FileName` is the name of the LaTeX file (e. g. `'figure.tex'`)
  - `PgWidth, PgHeight` are the dimensions of the graphic area in cm

– `Header` is a boolean parameter which allows to create a new file and write the following preamble:

  ```
  \documentclass[12pt,a4paper]{article}
  \usepackage{pst-plot}
  \pagestyle{empty}
  \begin{document}
  ```

  If `Header` is `False`, no preamble will be written.

Symmetrically, the statement `TeX_LeaveGraphics` has a boolean parameter `Footer` which allows to print the `\end{document}` section and close the file.

So, you can place several plots in a single document by calling the two procedures with the relevant boolean parameters. For instance, the following sequence will place two plots on the same page:

```
{ Create new file and write preamble }
if TeX_InitGraphics('figure.tex', 15, 10, True) then
  begin
    TeX_SetWindow(10, 90, 10, 90, True);
    ....................................

    (* Close file but don't write the '\end{document}' section *)
    TeX_LeaveGraphics(False);
  end;

{ Append to existing file and don't write preamble }
if TeX_InitGraphics('figure.tex', 15, 10, False) then
  begin
    TeX_SetWindow(10, 90, 10, 90, True);
    ....................................

    (* Close file and write the '\end{document}' section *)
    TeX_LeaveGraphics(True);
  end;
```

• Procedure `TeX_SetWindow` behaves as its BGI equivalent.

7.4.2 Axes and titles

The following procedures behave as their BGI equivalents:

- `TeX_SetOxScale`
- `TeX_SetOyScale`
- `TeX_SetGraphTitle`
- `TeX_SetOxTitle`
- `TeX_SetOyTitle`
- `TeX_PlotOxAxis`
- `TeX_PlotOyAxis`
- `TeX_PlotGrid`
- `TeX_WriteGraphTitle`

7.4.3 Curves

- The curves have the same properties than in the BGI version, except that:
  - there is no `Color` parameter;
  - there is an additional boolean property `Smooth` which indicates if the curve must be smoothed.

  These properties are set with the following procedures:

  - `TeX_SetPointParam(CurvIndex, Symbol, Size)`
  - `TeX_SetLineParam(CurvIndex, Style, Width, Smooth)`
- The following procedures behave as their BGI equivalents:
  - `TeX_SetMaxCurv`, `TeX_SetCurvLegend`, `TeX_SetCurvStep`
  - `TeX_PlotCurve`, `TeX_PlotCurveWithErrorBars`
  - `TeX_WriteLegend`, `TeX_WriteLegendSelect`
  - `TeX_ConRec`
- Procedure `TeX_PlotFunc(Func, X1, X2, Npt, CurvIndex)` has an additional parameter `Npt` which denotes the number of points to be plotted.

7.4.4 Other functions

Functions `Xcm(X)` and `Ycm(Y)` convert the user coordinates `X` and `Y` to their equivalent in cm.

7.5 RGB / HSV conversion

The following procedures allow to convert a color between the RGB (*Red, Green, Blue*) and HSV (*Hue, Saturation, Value*) color spaces:

- `RGBtoHSV(R, G, B, H, S, V)`
- `HSVtoRGB(H, S, V, R, G, B)`

where:

- `R, G, B` are in [0, 255] (`Byte` type)
- `H` is in [0, 360] (`Float` type)
- `S` and `S` are in [0, 1] (`Float` type)

7.6 Demo programs

7.6.1 BGI programs

These programs are located in the `demo\bgi\fmath` subdirectory.

- Program `plot.pas` plots a function in linear or logarithmic coordinates. The square root function is taken as example, since its graph becomes a straight line in log-log coordinates.
- Program `contour.pas` draws a contour plot of a function of two variables, using the `ConRec` algorithm developed by Paul Bourke.

  The example function is:

  $$f(x, y) = \sin \sqrt{x^2 + y^2} + \frac{1}{2\sqrt{(x + c)^2 + y^2}}$$

  where c is a constant which may be varied to modifiy the aspect of the graph.

7.6.2 GUI programs

These programs are located in the `demo\gui` subdirectory.

- Program `fplot.dpr` (in `demo\gui\fplot`) allows to plot up to 9 functions defined by the user. These functions are interpreted at run-time by the expression parser described in the previous chapter. The functions must therefore be written according to the syntax of the parser.

  Clicking the button "Graph Options" will bring in a special dialog for setting all the options for the whole graphic, axes, curves, titles etc. These options can be saved in a configuration file having the *.GCF extension.

  The image can be saved in a BMP file by clicking the "Save" button.

- Program `hsv_rgb.dpr` (in `demo\gui\hsv_rgb`) performs the conversion between the RGB and HSV color spaces.

7.6.3 LaTeX program

Program `texdemo.pas`, placed in the `demo\console\fmath` directory, creates a LaTeX file `figure.tex` which plots a Fourier series:

$$F_1 = 0.75(1 + \cos\phi)$$
$$F_2 = 2.5[1 + \cos(2\phi - \pi)]$$
$$F_3 = F_1 + F_2$$

This function is used in chemistry to describe the torsional energy of an amide bond.

The resulting file must be processed with `latex`. The graphics can be converted to PostScript by the `dvips` utility:

```
latex figure
dvips figure
```

Chapter 8

String functions

Some string functions have been added to DMath, mainly to help printing results.

8.1 Trim functions

- function `LTrim(S)` removes the leading blanks in string `S`
- function `RTrim(S)` removes the trailing blanks in string `S`
- function `Trim(S)` removes the leading and trailing blanks in string `S`

8.2 Fill functions

- function `RFill(S, L)` returns string `S` completed with trailing blanks for a total length `L`
- function `LFill(S, L)` returns string `S` completed with leading blanks for a total length `L`
- function `CFill(S, L)` returns string `S` completed with leading blanks so as to center the string on a total length `L`
- function `StrChar(N, C)` returns a string made of character `C` repeated `N` times

8.3 Character replacement

Subroutine `Replace(S, C1, C2)` replaces in string `S` all the occurences of character `C1` by character `C2`

8.4 Parsing

- function `Extract(S, Index, Delim)` extracts a field from string `S`. `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.

- procedure `Parse(S, Delim, Field, N)` parses string `S` 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.

8.5 Formatting functions

These functions allow to convert numbers to strings.

- procedure `SetFormat(NumLength, MaxDec, FloatPoint, NSZero)` defines the numeric format, according to the following parameters:

`NumLength`	:	Length of numeric field (default 10)
`MaxDec`	:	Max. number of decimal places (default 4)
`FloatPoint`	:	Select floating point notation (default `False`)
`NSZero`	:	Write non significant zero's (default `True`)

- function `FloatStr(X)` converts the real number `X` to a string according to the numeric format specified by `SetFormat`

- function `IntStr(N)` converts the integer `N` to a string.

- function `CompStr(Z)` converts the complex number `Z` to a string.

8.6 Delphi specific functions

These functions control the appearance of strings which represent floating point numbers, according to the decimal separator (point or comma) defined in the Windows settings.

- function `StrDec(S)` modifies string `S` so that it contains the correct decimal separator (e. g. `1.2` will be converted to `1,2` if the comma is selected as the decimal separator).
- function `IsNumeric(S, X)` replaces in string `S` the decimal separator by a point and returns TRUE if the string represents a number, which is returned in `X`.
- function `ReadNumFromEdit(Edit)` reads a floating point number from an Edit control.
- procedure `WriteNumToFile(F, X)` writes the floating point number `X` in the text file `F`, forcing the use of a decimal point.

Chapter 9

Matrices and linear equations

This chapter describes the procedures and functions available in DMath to perform vector and matrix operations, and to solve systems of linear equations.

9.1 Using vectors and matrices

DMath defines the following dynamic array types:

Vector type	Matrix type	Base variable
`TVector`	`TMatrix`	Floating point number (type `Float`)
`TIntVector`	`TIntMatrix`	Integer
`TCompVector`	`TCompMatrix`	Complex number (type `Complex`)
`TBoolVector`	`TBoolMatrix`	Boolean
`TStrVector`	`TStrMatrix`	String

To use these arrays in your programs, you must:

1. Declare variables of the appropriate type, then allocate each array *before* using it:

```
var
  V : TVector;
  A : TMatrix;
begin
  DimVector(V, N);     { creates vector V[0..N] }
  DimMatrix(A, N, M);  { creates matrix A[0..N, 0..M] }
                       { N, M are integer variables }
  ...
end.
```

Each vector or matrix type has a matching `Dim...` statement, e. g. `DimIntVector`, `DimCompMatrix` ...

If the allocation does not succeed, the array is given the value `nil`. So, it is possible to test the result:

```
DimVector(V, 10000);
if V = nil then
  Write('Not enough memory!');
```

Note that this allocation step is mandatory. Unlike standard Pascal arrays, it is not sufficient to declare the variables!

2. Use these arrays as in standard Pascal, noting that:

   (a) All arrays begin at index 0, so that the 0-indexed element is always present, even if you don't use it.

   (b) A matrix is declared as an array of vectors, so that `A[I]` denotes the I-th row of matrix `A` and may be used as any vector.

   (c) Vector and matrix parameters must be passed to functions or procedures with the `var` attribute when these parameters are dimensioned inside the procedure. Otherwise, this attribute is not necessary.

9.2 Maximal array sizes and initialization

The maximal array size is $2^{31} - 1 = 2147483647$ for Delphi and $2^{15} - 1 = 32767$ for FPC/Lazarus.

In principle, the compiler should initialize the numeric arrays to zero, the boolean arrays to `False` and the string arrays to the null string. If this is not the case, it is possible to force this initialization with the statement `SetAutoInit(True)`, and to revert it with `SetAutoInit(False)`.

9.3 Programming conventions

The following conventions have been adopted:

- Parameters `Lb` and `Ub` denote the lower and upper bounds of the indices, for a vector `V[Lb..Ub]` or a square matrix `A[Lb..Ub, Lb..Ub]`.

- Parameters `Lb1, Ub1` and `Lb2, Ub2` denote the lower and upper bounds of the indices, for a rectangular matrix `A[Lb1..Ub1, Lb2..Ub2]`.
- With the exception of the memory allocation routines (`DimVector, DimMatrix ...`), the procedures do not allocate the vectors or matrices present in their parameter lists. These allocations must therefore be performed by the main program, before calling the procedures.

9.4 Error codes

The following error codes are defined:

Error code	Value	Meaning
`MatOk`	0	No error
`MatNonConv`	-1	Non-convergence of an iterative procedure
`MatSing`	-2	Quasi-singular matrix
`MatErrDim`	-3	Non-compatible dimensions
`MatNotPD`	-4	Matrix not positive definite

9.5 Gauss-Jordan elimination

9.5.1 General case

If $\mathbf{B}(n \times n)$ and $\mathbf{C}(n \times m)$ are two real matrices, the Gauss-Jordan elimination can compute the inverse matrix $\mathbf{B}^{-1}$, the solution $\mathbf{X}$ to the system of linear equations $\mathbf{BX} = \mathbf{C}$, and the determinant of $\mathbf{B}$.

This procedure is implemented in unit `ugausjor` as:

```
GaussJordan(A, Lb, Ub1, Ub2, Det)
```

where:

- On input, `A[Lb..Ub1, Lb..Ub2]` is the global matrix $[\mathbf{B}|\mathbf{C}]$, which means that:
  - the first n columns of $\mathbf{A}$ contain the matrix $\mathbf{B}$
  - the other columns of $\mathbf{A}$ contain the matrix $\mathbf{C}$
- On output, $\mathbf{A}$ is transformed into the global matrix $[\mathbf{B}^{-1}|\mathbf{X}]$, which means that:

  - the first n columns of $\mathbf{A}$ contain the inverse matrix $\mathbf{B^{-1}}$
  - the other columns of $\mathbf{A}$ contain the solution matrix $\mathbf{X}$

- `Det` is the determinant of $\mathbf{B}$

Notes:

- $\mathbf{C}$ may be a vector, in this case $m = 1$ and $\mathbf{X}$ is also a vector.
- The original matrix $\mathbf{A}$ is overwritten by the procedure. If necessary, the calling program must save a copy of it.

After a call to `GaussJordan`, the function `MathErr` will return the error code:

- `MatOk` if no error
- `MatErrDim` if `Ub1` > `Ub2`
- `MatSing` if $\mathbf{B}$ is quasi-singular

9.5.2 Special case

Procedure `LinEq(A, B, Lb, Ub, Det)`, defined in unit `ulineq`, solves the system $\mathbf{AX} = \mathbf{B}$, where $\mathbf{A}$ is a square matrix and $\mathbf{B}$ a vector, by the Gauss-Jordan elimination method. In this case, the inverse matrix is returned in `A` and the solution vector $\mathbf{X}$ is returned in `B`.

9.6 LU decomposition

The LU decomposition algorithm factors the square matrix $\mathbf{A}$ as a product $\mathbf{LU}$, where $\mathbf{L}$ is a lower triangular matrix (with unit diagonal terms) and $\mathbf{U}$ is an upper triangular matrix.

The linear system $\mathbf{AX} = \mathbf{B}$ is then solved by:

$$\mathbf{LY} = \mathbf{B} \tag{9.1}$$

$$\mathbf{UX} = \mathbf{Y} \tag{9.2}$$

System 9.1 is solved for vector $\mathbf{Y}$, then system 9.2 is solved for vector $\mathbf{X}$. The solutions are simplified by the triangular nature of the matrices.

DMath provides the following procedures in unit `ulu` :

- procedure `LU_Decomp(A, Lb, Ub)` performs the LU decomposition of matrix `A[Lb..Ub, Lb..Ub]`.

  The matrices **L** and **U** are stored in **A**, which is therefore destroyed.

  After a call to `LU_Decomp`, the function `MathErr` will return one of the following error codes:

  – `MatOk` if no error

  – `MatSing` if **A** is quasi-singular

- procedure `LU_Solve(A, B, Lb, Ub, X)` solves the system $\mathbf{AX} = \mathbf{B}$, where **X** and **B** are real vectors, once the matrix **A** has been transformed by `LU_Decomp`.

9.7 QR decomposition

This method factors a matrix **A** as a product of an orthogonal matrix **Q** by an upper triangular matrix **R**:

$$\mathbf{A} = \mathbf{QR}$$

The linear system $\mathbf{AX} = \mathbf{B}$ then becomes:

$$\mathbf{QRX} = \mathbf{B}$$

Denoting the transpose of **Q** by $\mathbf{Q}^\top$ and left-multiplying by this transpose, one obtains:

$$\mathbf{Q}^\top\mathbf{QRX} = \mathbf{Q}^\top\mathbf{B}$$

or:

$$\mathbf{RX} = \mathbf{Q}^\top\mathbf{B}$$

since the transpose of an orthogonal matrix is equal to its inverse.

The last system is solved by making advantage of the triangular nature of matrix **R**.

Note : The QR decomposition may be applied to a rectangular matrix $n \times m$ (with $n > m$). In this case, **Q** has dimensions $n \times m$ and **R** has dimensions $m \times m$. For a linear system $\mathbf{AX} = \mathbf{B}$, the solution minimizes the norm of the vector **AX** - **B**. It is called the *least squares* solution.

DMath provides the following procedures in unit `uqr` :

- procedure `QR_Decomp(A, Lb, Ub1, Ub2, R)` performs the QR decomposition on the input matrix `A[Lb..Ub1, Lb..Ub2]`.

  The matrix $\mathbf{Q}$ is stored in $\mathbf{A}$, which is therefore destroyed.

  After a call to `QR_Decomp`, the function `MathErr` will return one of the following error codes:

  - `MatOk` if no error
  - `MatErrDim` if `Ub2` $>$ `Ub1`
  - `MatSing` if $\mathbf{A}$ is quasi-singular

- procedure `QR_Solve(Q, R, B, Lb, Ub1, Ub2, X)` solves the system $\mathbf{QRX} = \mathbf{B}$.

9.8 Singular value decomposition

Singular value decomposition (SVD) factors a matrix $\mathbf{A}$ as a product:

$$\mathbf{A} = \mathbf{USV}^\top$$

where $\mathbf{U}$ et $\mathbf{V}$ are orthogonal matrices. $\mathbf{S}$ is a diagonal matrix. Its diagonal terms S_{ii} are all ≥ 0 and are called the *singular values* of $\mathbf{A}$. The *rank* of $\mathbf{A}$ is equal to the number of non-null singular values.

- If $\mathbf{A}$ is a regular matrix, all S_{ii} are > 0. The inverse matrix is given by:

  $$\mathbf{A}^{-1} = (\mathbf{USV}^\top)^{-1} = (\mathbf{V}^\top)^{-1}\mathbf{S}^{-1}\mathbf{U}^{-1} = \mathbf{V} \times \operatorname{diag}(1/S_{ii}) \times \mathbf{U}^\top$$

  since the inverse of an orthogonal matrix is equal to its transpose.

  So the solution of the system $\mathbf{AX} = \mathbf{B}$ is given by $\mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$

- If $\mathbf{A}$ is a singular matrix, some S_{ii} are null. However, the previous expressions remain valid provided that, for each null singular value, the term $1/S_{ii}$ is replaced by zero.

  It may be shown that the solution so calculated corresponds:

  - in the case of an under-determined system, to the vector $\mathbf{X}$ having the least norm.

– in the case of an impossible system, to the least-squares solution.

Note : Just like the QR decomposition, the SVD may be applied to a rectangular matrix $n \times m$ (with $n > m$). In this case, $\mathbf{U}$ has dimensions $n \times m$, $\mathbf{S}$ and $\mathbf{V}$ have dimensions $m \times m$. For a linear system $\mathbf{AX} = \mathbf{B}$, the SVD method gives the least squares solution.

DMath provides the following procedures in unit `usvd` :

- procedure `SV_Decomp(A, Lb, Ub1, Ub2, S, V)` performs the singular value decomposition on the input matrix `A[Lb..Ub1, Lb..Ub2]`.

  The matrix $\mathbf{U}$ (such that $\mathbf{A} = \mathbf{USV}^\top$) is stored in $\mathbf{A}$, which is therefore destroyed.

  After a call to `SV_Decomp`, the function `MathErr` will return one of the following error codes:

  – `MatOk` if no error

  – `MatErrDim` if `Ub2` > `Ub1`

  – `MatNonConv` if the iterative process does not converge

- procedure `SV_SetZero(S, Lb, Ub, Tol)` sets to zero the singular values S_i which are lower than a fraction `Tol` of the highest singular value. This procedure may be used when solving a system with a near-singular matrix.

- procedure `SV_Solve(U, S, V, B, Lb, Ub1, Ub2, X)` solves the system $\mathbf{USV}^\top\mathbf{X} = \mathbf{B}$.

- procedure `SV_Approx(U, S, V, Lb, Ub1, Ub2, A)` approximates a matrix $\mathbf{A}$ by the product $\mathbf{USV}^\top$, after the lowest singular values have been set to zero by `SV_SetZero`.

9.9 Cholesky decomposition

The symmetric matrix $\mathbf{A}$ is said to be *positive definite* if, for any nonzero vector $\mathbf{x}$, the product $\mathbf{x}^\top\mathbf{Ax}$ is > 0.

For such matrices, it is possible to find a lower triangular matrix $\mathbf{L}$ such that:

$$\mathbf{A} = \mathbf{LL}^\top$$

$\mathbf{L}$ can be viewed as a kind of 'square root' of $\mathbf{A}$.

Subroutine `Cholesky(A, L, Lb, Ub)`, defined in unit `ucholesk`, performs the Cholesky decomposition on `A[Lb..Ub, Lb..Ub]`. After a call to the subroutine, function `MathErr` returns the error code:

- `MatOk` if there is no error.
- `MatErrDim` if the matrix dimensions do not match.
- `MatNotPD` if `A` is not positive definite.

9.10 Eigenvalues and eigenvectors

9.10.1 Definitions

A square matrix $\mathbf{A}$ is said to have an eigenvalue λ, associated to an eigenvector $\mathbf{V}$, if and only if:

$$\mathbf{A} \cdot \mathbf{V} = \lambda \cdot \mathbf{V}$$

A symmetric matrix of size n has n distinct real eigenvalues and n orthogonal eigenvectors.

A non-symmetric matrix of size n has also n eigenvalues but some of them may be complex, and some may be equal (they are said to be degenerate).

9.10.2 Symmetric matrices

- Procedure `EigenSym(A, Lb, Ub, Lambda, V)`, defined in unit `ueigsym`, computes the eigenvalues and eigenvectors of the real symmetric positive semi-definite matrix `A[Lb..Ub, Lb..Ub]` by singular value decomposition.

  The eigenvectors are returned in matrix $\mathbf{V}$; the eigenvalues are returned in vector `Lambda`.

  The eigenvectors are stored along the columns of $\mathbf{V}$. They are normalized, with their first component always positive.

  The error codes are those of the `SV_Decomp` procedure.

- Procedure `Jacobi(A, Lb, Ub, MaxIter, Tol, Lambda, V)`, defined in unit `ujacobi`, computes the eigenvalues and eigenvectors of the real symmetric matrix `A[Lb..Ub, Lb..Ub]`, using the iterative method of Jacobi. The eigenvalues and eigenvectors are ordered and normalized as with the previous procedure.

`MaxIter` is the maximum number of iterations, `Tol` is the required precision on the eigenvalues.

After a call to `Jacobi`, function `MathErr` returns one of two error codes:

- `MatOk` if all goes well.
- `MatNonConv` if the iterative process does not converge.

These procedures destroy the original matrix **A**.

9.10.3 General square matrices

- procedure `EigenVals(A, Lb, Ub, Lambda)`, defined in unit `ueigval`, computes the eigenvalues of the real square matrix `A[Lb..Ub, Lb..Ub]`.

  Eigenvalues are stored in the complex vector `Lambda`. The real and imaginary parts of the ith eigenvalue are stored in `Lambda[i].X` and `Lambda[i].Y`, respectively. The eigenvalues are unordered, except that complex conjugate pairs appear consecutively with the value having the positive imaginary part first.

  Function `MathErr` returns the following error codes:

  - 0 if no error
  - (-i) if an error occurred during the determination of the ith eigenvalue. The eigenvalues should be correct for the indices > i.

  This procedure destroys the original matrix **A**.

- procedure `EigenVect(A, Lb, Ub, Lambda, V)`, defined in unit `ueigvec`, computes the eigenvalues and eigenvectors of the real square matrix `A[Lb..Ub, Lb..Ub]`.

  Eigenvalues are stored in the *complex* vector `Lambda`, just like with `EigenVals`.

  Eigenvectors are stored along the columns of the *real* matrix **V**.

  If the ith eigenvalue is real, the ith column of **V** contains its eigenvector. If the ith eigenvalue is complex with positive imaginary part, the ith and (i+1)th columns of **V** contain the real and imaginary parts of its eigenvector. The eigenvectors are unnormalized.

Function `MathErr` returns the same error codes than `EigenVals`. If the error code is not null, none of the eigenvectors has been found.

This procedure destroys the original matrix $\mathbf{A}$.

9.11 Demo programs

9.11.1 Console programs

These programs are located in the `demo\console\matrices` subdirectory.

Determinant and inverse of a square matrix

Program `detinv.pas` computes the determinant and inverse of a square matrix. The inverse matrix is re-inverted and the result (which should be equal to the original matrix) is printed. The determinant of the inverse matrix is also evaluated and the product of the two determinants (which should be -1) is displayed.

The example matrix is:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 0 & -1 \\ -1 & 4 & 3 & -0.5 \\ 2 & 2 & 1 & -3 \\ 0 & 0 & 3 & -4 \end{bmatrix}$$

The inverse is:

$$\mathbf{A}^{-1} = \begin{bmatrix} -\frac{41}{21} & \frac{4}{21} & \frac{11}{7} & -\frac{5}{7} \\ \frac{16}{21} & \frac{1}{21} & -\frac{5}{14} & \frac{1}{14} \\ -\frac{40}{21} & \frac{8}{21} & \frac{8}{7} & -\frac{3}{7} \\ -\frac{10}{7} & \frac{2}{7} & \frac{6}{7} & -\frac{4}{7} \end{bmatrix}$$

or, in approximate form:

$$\mathbf{A}^{-1} \approx \begin{bmatrix} -1.9523 & 0.1905 & 1.5714 & -0.7143 \\ 0.7619 & 0.0476 & -0.3571 & 0.0714 \\ -1.9048 & 0.3810 & 1.1429 & -0.4286 \\ -1.4286 & 0.2857 & 0.8571 & -0.5714 \end{bmatrix}$$

The determinant is -21.

Hilbert matrices

Program `hilbert.pas` tests the Gauss-Jordan method by solving a series of Hilbert systems of increasing order. Such systems have matrices of the form:

$$\mathbf{A} = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{N} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{N+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \cdots & \frac{1}{N+2} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \cdots & \frac{1}{N+3} \\ \vdots & & & & & \vdots \\ \frac{1}{N} & \frac{1}{N+1} & \frac{1}{N+2} & \frac{1}{N+3} & \cdots & \frac{1}{2N-1} \end{bmatrix}$$

Each element of the constant vector (stored in the $(N+1)^{th}$ column of matrix **A**) is equal to the sum of the terms in the corresponding line of the matrix :

$$A_{i,N+1} = \sum_{j=1}^{N} A_{ij}$$

The solution of such a system is $[1, 1, 1, \cdots 1]$

The determinant of the Hilbert matrix tends towards zero when the order increases. The program stops when the determinant becomes too low with respect to the numerical precision of the floating point numbers. This occurs at order 13 in double precision.

Gauss-Jordan method: single constant vector

Program `lineq1.pas` solves the linear system $\mathbf{AX} = \mathbf{B}$. After a call to `LinEq`, **A** contains the inverse matrix and **B** contains the solution vector.

The example system matrix is:

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 5 & -8 \\ 7 & 6 & 2 & 2 \\ -1 & -3 & -10 & 4 \\ 2 & 2 & 2 & 1 \end{bmatrix}$$

The constant vector is:

$$\mathbf{B} = \begin{bmatrix} 0 \\ 17 \\ -10 \\ 7 \end{bmatrix}$$

The solution vector is:

$$\mathbf{X} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The determinant is -135

Gauss-Jordan method: multiple constant vectors

Program `lineqm.pas` solves a series of linear systems with the same system matrix and several constant vectors. The system matrix is stored in the first n columns of matrix $\mathbf{A}$; the constant vectors are stored in the following columns. After a call to `GaussJordan`, the first n columns of $\mathbf{A}$ contain the inverse matrix and the following columns contain the solution vectors.

The example system matrix from the previous program is used. The matrix of constant vectors is:

$$\begin{bmatrix} 0 & -15 & 14 & -13 & 5 \\ 17 & 50 & 1 & 84 & 30 \\ -10 & -5 & -12 & -51 & -15 \\ 7 & 17 & 1 & 37 & 10 \end{bmatrix}$$

The solution matrix is:

$$\begin{bmatrix} 1 & 2 & 1 & 4 & 0 \\ 1 & 5 & -1 & 5 & 5 \\ 1 & 0 & 1 & 6 & 0 \\ 1 & 3 & -1 & 7 & 0 \end{bmatrix}$$

LU, QR and SV decompositions

The demo programs `test_lu.pas`, `test_qr.pas` and `test_svd.pas` solve the linear system used by `lineq1.pas` (paragraph 9.11.1) with the LU, QR, and singular value decompositions, respectively.

Cholesky decomposition

Program `cholesk.pas` performs the Cholesky decomposition of a positive definite symmetric matrix. The matrix is decomposed then the program computes the product $\mathbf{LL}^\top$ which must give the original matrix.

The example matrix is:

$$\mathbf{A} = \begin{bmatrix} 60 & 30 & 20 \\ 30 & 20 & 15 \\ 20 & 15 & 12 \end{bmatrix}$$

Its Cholesky factor is:

$$\mathbf{L} = \begin{bmatrix} 2\sqrt{15} & 0 & 0 \\ \sqrt{15} & \sqrt{5} & 0 \\ \frac{2}{3}\sqrt{15} & \sqrt{5} & \frac{1}{3}\sqrt{3} \end{bmatrix}$$

or, in approximate form:

$$\mathbf{L} \approx \begin{bmatrix} 7.745967 & 0 & 0 \\ 3.872983 & 2.236068 & 0 \\ 2.581989 & 2.236068 & 0.577350 \end{bmatrix}$$

Eigenvalues of a symmetric matrix

Program `eigensym.pas` computes the eigenvalues and eigenvectors of Hilbert matrices (see program `hilbert.pas`) by the SVD or Jacobi methods. Such matrices are very ill-conditioned, which can be seen from the high ratio between the highest and lowest eigenvalues (the *condition number*).

Eigenvalues of a general square matrix

Program `eigenval.pas` computes the eigenvalues of a general square matrix.

The example matrix from the `detinv.pas` program is used. It has two real and two complex (conjugate) eigenvalues:

```
-1.075319 +     1.709050 * i
-1.075319 -     1.709050 * i
-1.000000
 5.150639
```

Eigenvalues and eigenvectors of a general square matrix

Program `eigenvec.pas` computes both the eigenvalues and eigenvectors of a general square matrix. The same example matrix is used.

The eigenvectors are stored columnwise in a matrix **V**. In order to retrieve the eigenvectors associated with complex eigenvalues, the program takes into account the following properties:

- Complex conjugate pairs of eigenvalues are stored consecutively in vector `Lambda`, with the value having the positive imaginary part first.
- If the ith eigenvalue is complex with positive imaginary part, the ith and (i+1)th columns of matrix **V** contain the real and imaginary parts of its eigenvector.
- Eigenvectors associated with complex conjugate eigenvalues are themselves complex conjugate.

Hence the algorithm:

```
if Lambda[I].Y = 0.0 then
  { Eigenvector is in column I of V }
else if Lambda[I].Y > 0.0 then
  { Real and imag. parts of eigenvector are in columns I and (I+1)
    For component K: real part  = V[K, I]
                     imag. part = V[K, I+1] }
else
  { Real and imag. parts of eigenvector are in columns (I-1) and I
    For component K: real part  = V[K, I-1],
                     imag. part = - V[K, I] }
```

The results obtained with the example matrix are the following:

```
Eigenvalue:

    -1.075319 +      1.709050 * i

Eigenvector:

    -0.220224 +      0.394848 * i
     0.078289 -      0.303345 * i
     0.029348 +      0.787594 * i
     0.374358 +      0.589119 * i

Eigenvalue:

    -1.075319 -      1.709050 * i

Eigenvector:
    -0.220224 -      0.394848 * i
     0.078289 +      0.303345 * i
     0.029348 -      0.787594 * i
     0.374358 -      0.589119 * i

Eigenvalue:

    -1.000000

Eigenvector:

     2.605054
    -1.042021
     3.126065
     3.126065

Eigenvalue:

     5.150638

Eigenvector:

     0.345194
     0.788801
     0.441744
     0.144823
```

9.11.2 GUI programs

Linear system solver

Program `linsolve.dpr`, located in `demo\gui\linsolve`, solves a system of linear equations $\mathbf{AX} = \mathbf{B}$, where $\mathbf{A}$ is the system matrix and $\mathbf{B}$ is the constant vector, by any of the four methods: Gauss-Jordan (default), LU, QR or SVD. The product $\mathbf{AX}$ is then computed for comparison with the constant vector $\mathbf{B}$.

The following buttons are available:

- **Load system** : reads data from a text file (example: `matrix3.dat` in `demo\console\matrices`)
- **Save system** : writes data in a text file

- **Numeric format** : sets the printing format for the results according to the function `SetFormat` (see chapter 8).
- **Solve system**
- **Save solution** : stores the solution vector in a text file, using the settings from 'Numeric format'.

Chapter 10

Function minimization

This chapter describes the procedures and functions available in DMath to minimize functions of one or several variables. Only deterministic optimizers are considered here. Stochastic optimization will be studied in another chapter.

10.1 Functions of one variable

Let `Func` be a function of a real variable `X`. In DMath such a function is declared as:

```
function Func(X : Float) : Float;
```

There is a special type `TFunc` for this kind of functions.

The problem is to find the real `Xmin` for which `Func(X)` is minimal.

Procedure `GoldSearch(Func, A, B, MaxIter, Tol, Xmin, Ymin)`, defined in unit `ugoldsrc`, performs the minimization by the 'golden search' method. This means that, at each iteration, the number `Xmin` is 'bracketed' by a triplet (`A, B, C`) such that:

- $A < B < C$
- A, B, C are within the golden mean ϕ, i.e.

$$\frac{B-A}{C-B} = \frac{C-A}{B-A} = \phi = \frac{1+\sqrt{5}}{2} \approx 1.618$$

- `Func(B) < Func(A)` and `Func(B) < Func(C)`.

The user must provide two numbers `A` and `B` which define the 'unit vector' on the X axis. The number `C` is found by the program itself. It is not necessary that the interval `[A, B]` contains the minimum.

The user must also provide:

- the maximum number of iterations `MaxIter`
- the tolerance `Tol` with which the minimum must be located. This value should not be higher than the square root of the machine precision (`MachEp`$^{1/2} \approx 1.5 \times 10^{-8}$ in double precision)

The procedure returns the coordinates `(Xmin, Ymin)` of the minimum.

After a call to `GoldSearch`, function `MathErr()` will return one of two error codes:

- `OptOk` if no error occurred
- `OptNonConv` (non-convergence) if the number of iterations exceeds the maximum value `MaxIter`

The determination of the bracketing triplet `A, B, C` is performed within `GoldSearch` by a call to a procedure `MinBrack`, defined in unit `uminbrak`. This procedure may be called independently. Its syntax is:

```
MinBrack(Func, A, B, C, Fa, Fb, Fc)
```

The user must provide the first two numbers `A` and `B`. The number `C` is found by the procedure. The corresponding values of the function are returned in `Fa, Fb, Fc`.

10.2 Functions of several variables

Let `Func` be a function of a real vector $\mathbf{x}$ such that $\mathbf{x} = [x_1, x_2, \cdots]$. In DMath such a function is declared as:

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

There is a special type `TFuncNVar` for this kind of functions.

The problem is to find the vector `X` for which `Func(X)` is minimal.

10.2.1 Minimization along a line

If $\mathbf{x}^0$ is a starting point and $\delta\mathbf{x}$ is a constant vector, minimizing f from $\mathbf{x}^0$ along the direction specified by $\delta\mathbf{x}$ is equivalent to finding the number r such that $g(r) = f(\mathbf{x}^0 + r \cdot \delta\mathbf{x})$ is minimal.

The following procedure, defined in unit `ulinmin` :

```
LinMin(Func, X, DeltaX, Lb, Ub, R, MaxIter, Tol, F_min)
```

will minimize function `Func` from point `X[Lb..Ub]` in the direction specified by vector `DeltaX[Lb..Ub]`. `R` is the initial step in that direction, expressed as a fraction of the norm of `DeltaX`. If `R` is set to 0 or a negative value, the procedure will use the default value `R` = 1. The user must also provide the maximum number of iterations `MaxIter` and the tolerance `Tol`, as for procedure `GoldSearch`.

On output, `LinMin` returns:

- the coordinates of the minimum in `X`
- the step corresponding to the minimum in `R`
- the function value at the minimum in `F_min`

After a call to `LinMin`, function `MathErr()` will return one of the error codes `OptOk` or `OptNonConv`, as with `GoldSearch`.

10.2.2 Newton-Raphson method

The Newton-Raphson method starts with an approximation $\mathbf{x}^0$ for the coordinates of the minimum and generates a new approximation $\mathbf{x}$ by using the second-order Taylor series expansion of function f around $\mathbf{x}^0$:

$$f(\mathbf{x}) = f(\mathbf{x}^0) + (\mathbf{x} - \mathbf{x}^0)^\top \cdot \mathbf{g}(\mathbf{x}^0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^0)^\top \cdot \mathbf{H}(\mathbf{x}^0) \cdot (\mathbf{x} - \mathbf{x}^0) \quad (10.1)$$

$\mathbf{g}$ denotes the gradient vector (vector of first partial derivatives) and $\mathbf{H}$ denotes the hessian matrix (matrix of second partial derivatives). For instance, for a fonction of two variables $f(x_1, x_2)$:

$$\mathbf{g}(\mathbf{x}^0) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1^0, x_2^0) \\ \\ \frac{\partial f}{\partial x_2}(x_1^0, x_2^0) \end{bmatrix} \qquad \mathbf{H}(\mathbf{x}^0) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x_1^0, x_2^0) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(x_1^0, x_2^0) \\ \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(x_1^0, x_2^0) & \frac{\partial^2 f}{\partial x_2^2}(x_1^0, x_2^0) \end{bmatrix}$$

By differentiating eq. (1) we obtain the gradient of f at point $\mathbf{x}$:

$$\mathbf{g}(\mathbf{x}) = \mathbf{g}(\mathbf{x}^0) + \mathbf{H}(\mathbf{x}^0) \cdot (\mathbf{x} - \mathbf{x}^0) \tag{10.2}$$

If $\mathbf{x}$ is sufficiently close to the minimum, $\mathbf{g}(\mathbf{x}) \approx 0$ so:

$$\mathbf{x} = \mathbf{x^0} - \mathbf{H^{-1}}(\mathbf{x}^0) \cdot \mathbf{g}(\mathbf{x}^0)$$

In practice, it is better to determine the step k which minimizes the function in the direction specified by $-\mathbf{H^{-1}}(\mathbf{x}^0) \cdot \mathbf{g}(\mathbf{x}^0)$:

$$\mathbf{x} = \mathbf{x^0} - k \cdot \mathbf{H^{-1}}(\mathbf{x}^0) \cdot \mathbf{g}(\mathbf{x}^0)$$

The determination of k is performed by line minimization.

The following procedure, defined in unit `unewton` :

`Newton(Func, HessGrad, X, Lb, Ub, MaxIter, Tol, F_min, G, H_inv, Det)`

minimizes function `Func` by the Newton-Raphson method.

The user must provide a procedure `HessGrad` to compute the gradient `G` and the hessian `H` of the function at point `X`. This procedure is declared as:

```
procedure HessGrad(X, G : TVector; H : TMatrix);
```

which corresponds to type `THessGrad`.

`MaxIter` and `Tol` have their usual meaning.

On output, `Newton` returns:

- the coordinates of the minimum in `X`
- the function value at the minimum in `F_min`
- the gradient at the minimum in `G` (should be near 0)
- the inverse hessian matrix at the minimum in `H_inv`
- the determinant of the hessian matrix at the minimum in `Det`

After a call to `Newton`, function `MathErr()` will return one of three error codes:

- `OptOk` if no error occurred
- `OptNonConv` (non-convergence) if the number of iterations exceeds the maximum value `MaxIter`
- `OptSing` if the hessian matrix is quasi-singular

Approximate gradient and hessian

Although it is recommended to compute the gradient and hessian from analytical derivatives, approximate values may be found using finite difference approximations:

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \approx \frac{f(x_i + h_i) - f(x_i - h_i)}{2h_i}$$

$$\frac{\partial^2 f}{\partial x_i^2}(\mathbf{x}) \approx \frac{f(x_i + h_i) + f(x_i - h_i) - 2f(x_i)}{h_i^2}$$

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) \approx \frac{f(x_i + h_i, x_j + h_j) - f(x_i + h_i, x_j) - f(x_i, x_j + h_j) + f(x_i, x_j)}{h_i h_j}$$

The increment h_i is such that $h_i = \eta \mid x_i \mid$ where η is a constant which should not be less than the cube root of the machine epsilon ($\texttt{MachEp}^{1/3} \approx 6.06 \times 10^{-6}$ in double precision).

This method is illustrated in the demo programs `testnewt.pas` and `testmarq.pas` (see paragraph 10.3).

10.2.3 Marquardt method

This method is a variant of the Newton-Raphson method, in which each diagonal term of the hessian matrix is multiplied by a scalar equal to $(1 + \lambda)$, where λ is the Marquardt parameter. This parameter is initialized at some small value (e.g. 10^{-2}) at the beginning of the iterations, then it is decreased by a factor 10 if the iteration leads to a decrease of the function, otherwise it is increased by a factor 10. If the method converges, λ is set to zero and an additional iteration (equivalent to a Newton-Raphson step) is performed.

This procedure is implemented in unit `umarq` as:

```
Marquardt(Func, HessGrad, X, Lb, Ub, MaxIter, Tol, F_min, G, H_inv, Det)
```

It is used like `Newton`, except that an additional error code, `OptBigLambda`, may be returned by `MathErr` if the Marquard parameter increases beyond a predefined value (10^3 in this implementation).

10.2.4 BFGS method

The BFGS (Broyden-Fletcher-Goldfarb-Shanno) method is another variant of the Newton method in which the hessian matrix does not need to be computed explicitly. It is said a *quasi-Newton* method.

The BFGS algorithm uses the following formula to construct the inverse hessian matrix iteratively:

$$\mathbf{H}_{i+1}^{-1} = \mathbf{H}_i^{-1} + \frac{\delta\mathbf{x} \cdot \delta\mathbf{x}^\top}{\delta\mathbf{x}^\top \cdot \delta\mathbf{g}} - \frac{(\mathbf{H}_i^{-1} \cdot \delta\mathbf{g}) \cdot (\mathbf{H}_i^{-1} \cdot \delta\mathbf{g})^\top}{\delta\mathbf{g}^\top \cdot \mathbf{H}_i^{-1} \cdot \delta\mathbf{g}} + (\delta\mathbf{g}^\top \cdot \mathbf{H}_i^{-1} \cdot \delta\mathbf{g}) \cdot \mathbf{u} \cdot \mathbf{u}^\top$$

with:

$$\delta\mathbf{x} = \mathbf{x}_{i+1} - \mathbf{x}_i \qquad \delta\mathbf{g} = \mathbf{g}(\mathbf{x}_{i+1}) - \mathbf{g}(\mathbf{x}_i) \qquad \mathbf{u} = \frac{\delta\mathbf{x}}{\delta\mathbf{x}^\top \cdot \delta\mathbf{g}} - \frac{\mathbf{H}_i^{-1} \cdot \delta\mathbf{g}}{\delta\mathbf{g}^\top \cdot \mathbf{H}_i^{-1} \cdot \delta\mathbf{g}}$$

The algorithm is usually started with the identity matrix ($\mathbf{H}_0^{-1} = \mathbf{I}$).

This procedure is implemented in unit `ubfgs` as:

```
BFGS(Func, Gradient, X, Lb, Ub, MaxIter, Tol, F_min, G, H_inv)
```

The user must provide a procedure `Gradient` to compute the gradient `G` of the function at point `X`. This procedure is declared as:

```
    procedure Gradient(X, G : TVector);
```

which corresponds to type `TGradient`.

The other parameters have the same meaning than in `Newton`.

Approximate gradient

It is possible to estimate the gradient of function `Func` by finite difference approximations, as described for the Newton method. Here the relative increment η should not be less than the square root of the machine epsilon (about 1.5×10^{-8} in double precision).

See demo program `testbfgs.pas` for an example.

As usual, it is recommended to use analytical derivatives whenever possible.

10.2.5 Simplex method

Unlike previous methods, the simplex method of Nelder and Mead does not use derivatives to locate the minimum. Instead it constructs a geometrical figure (the 'simplex') having $(n+1)$ vertices, if n is the number of variables. For instance, in the two-dimensional space ($n = 2$), the simplex would be a triangle. Depending on the function values at the vertices, the simplex is reduced or expanded until it comes close to the minimum.

This method is implemented in unit `usimplex` as:

```
Simplex(Func, X, Lb, Ub, MaxIter, Tol, F_min)
```

where the parameters have their usual meaning.

10.2.6 Log files

It is possible to create 'log files' which save the progress of the iterations. If the algorithm terminates abnormally, checking these files may help finding the error. For each method (Newton, Marquard, BFGS, Simplex) there is a `Save...` procedure which creates the log file. Each procedure accepts the name of the file as its parameter (e.g. `SaveBFGS('bfgs.txt')`). The file is automatically closed when the optimization procedure ends.

See the demo programs for examples using such files.

10.3 Demo programs

These programs are located in the `demo\console\optim` subdirectory.

Function of one variable

Program `minfunc.pas` performs the golden search minimization on the function:

$$f(x) = e^{-2x} - e^{-x}$$

The minimum is at $(\ln 2, -1/4)$.

The minimum found by `GoldSearch` is compared with the true minimum.

Minimization along a line

Program `minline.pas` applies line minimization to the function of 3 variables (taken from the *Numerical Recipes* example book):

$$f(x_1, x_2, x_3) = (x_1 - 1)^2 + (x_2 - 1)^2 + (x_3 - 1)^2$$

The minimum is $f(1,1,1) = 0$, i. e. for a step $r = 1$ from $\mathbf{x} = [0,0,0]$ in the direction $\delta\mathbf{x} = [1,1,1]$.

The program tries a series of directions:

$$\delta\mathbf{x} = \left[\sqrt{2}\cos\left(i\frac{\pi}{20}\right), \sqrt{2}\sin\left(i\frac{\pi}{20}\right), 1\right] \qquad i = 1..10$$

For each pass, the location of the minimum, and the value of the function at the minimum, are printed. The true minimum is found at $i = 5$.

Newton-Raphson method

Program `testnewt.pas` uses the Newton-Raphson method to minimize Rosenbrock's function (H. Rosenbrock, *Comput. J.*, 1960, **3**, 175):

$$f(x,y) = 100(y - x^2)^2 + (1 - x)^2$$

for which the gradient and hessian are:

$$\mathbf{g}(x,y) = \begin{bmatrix} -400(y - x^2)x - 2 + 2x \\ 200y - 200x^2 \end{bmatrix} \qquad \mathbf{H}(x,y) = \begin{bmatrix} 1200x^2 - 400y + 2 & -400x \\ -400x & 200 \end{bmatrix}$$

and the determinant of the hessian is:

$$\det \mathbf{H}(x,y) = 80000(x^2 - y) + 400$$

The minimum is $f(1,1) = 0$, where:

$$\mathbf{g}(1,1) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{H}^{-1}(1,1) = \begin{bmatrix} \frac{1}{2} & 1 \\ 1 & \frac{401}{200} \end{bmatrix}$$

$$\det \mathbf{H}(1,1) = 400$$

In the demo program, the gradient and hessian are computed analytically. You can compare with the numerical computations by including file `numhess.inc` in the program.

Other programs

Programs `testmarq.pas`, `testbfgs.pas` and `testsimp.pas` minimize Rosenbrock's function with the Marquardt, BFGS and Simplex methods, respectively.

Chapter 11

Nonlinear equations

This chapter describes the procedures available in DMath to solve nonlinear equations in one or several variables. Only general methods are considered here. Polynomial equations will be studied in the next chapter.

11.1 Equations in one variable

The goal is to solve the nonlinear equation $f(x) = 0$, or, in other terms, find a root of function f.

11.1.1 Bisection method

Procedure `Bisect(Func, X, Y, MaxIter, Tol, F)`, defined in unit `ubisect`, finds a root of function `Func` by the bisection method. At each iteration, the root is bounded by two numbers (`X, Y`) such that the function has opposite signs. Then, a new approximation to the root is generated by taking the mean of these numbers.

The function `Func` must be declared as:

```
function Func(X : Float) : Float;
```

The user must provide initial values for `X` and `Y`. It is not necessary that the interval `[X, Y]` contains the root.

The user must also provide:

- the maximum number of iterations `MaxIter`
- the tolerance `Tol` with which the root must be located.

The procedure returns the refined values of `X` and `Y` and the function value `Func(X)` in `F`.

After a call to `Bisect`, function `MathErr()` will return one of two error codes:

- `OptOk` if no error occurred
- `OptNonConv` (non-convergence) if the number of iterations exceeds the maximum value `MaxIter`

If the starting interval `[X, Y]` does not contain the root, `Bisect` will expand it by calling a procedure `RootBrack`, also defined in `ubisect`. This procedure may be called independently. Its syntax is:

```
RootBrack(Func, X, Y, FX, FY)
```

The user must provide initial values for the two numbers `X` and `Y`, which will be refined by the procedure. The corresponding function values are returned in `FX` and `FY`.

11.1.2 Secant method

The secant method also starts with two approximations x and y and generates a new approximation z from the formula:

$$z = \frac{xf(y) - yf(x)}{f(y) - f(x)}$$

z is the intersection of the Ox axis with the line connecting the points $(x, f(x))$ and $(y, f(y))$, i. e. the secant.

This method is implemented in unit `usecant` as:

```
Secant(Func, X, Y, MaxIter, Tol, F)
```

The parameters and error codes are the same than in `Bisect`. Here too, it is not necessary that the interval `[X, Y]` contains the root.

11.1.3 Newton-Raphson method

The Newton-Raphson method starts with an approximate root x^0 and generates a new approximation x by using the first-order Taylor series expansion of function f around x^0:

$$f(x) \approx f(x^0) + f'(x^0) \cdot (x - x^0)$$

If x is sufficiently close to the root, $f(x) \approx 0$ so:

$$x = x^0 - \frac{f(x^0)}{f'(x^0)}$$

This method is implemented in unit `unewteq` as:

```
NewtEq(Func, Deriv, X, MaxIter, Tol, F)
```

where `Func` and `Deriv` are the procedures which compute the function and its derivative, respectively (they have the same syntax). The user must provide the initial approximation `X`.

After a call to `NewtEq`, function `MathErr()` will return one of three error codes:

- `OptOk` if no error occurred
- `OptNonConv` (non-convergence) if the number of iterations exceeds the maximum value `MaxIter`
- `OptSing` if the derivative becomes zero

11.2 Equations in several variables

The goal is to solve a system of n nonlinear equations in n unknowns $x_1, x_2, \cdots x_n$:

$$\begin{array}{c} f_1(x_1, x_2, \cdots x_n) = 0 \\ f_2(x_1, x_2, \cdots x_n) = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots \\ f_n(x_1, x_2, \cdots x_n) = 0 \end{array}$$

or, in matrix notation:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

where $\mathbf{f}$ is a function vector.

11.2.1 Newton-Raphson method

The Newton-Raphson method starts with an approximate root $\mathbf{x}^0$ and generates a new approximation $\mathbf{x}$ by using the first-order Taylor series expansion of function $\mathbf{f}$ around $\mathbf{x}^0$:

$$\mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}^0) + \mathbf{D}(\mathbf{x}^0) \cdot (\mathbf{x} - \mathbf{x}^0)$$

D denotes the jacobian matrix (matrix of first partial derivatives). For instance, for a system of 2 equations in two variables:

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

the jacobian matrix is:

$$\mathbf{D}(\mathbf{x}^0) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1}(x_1^0, x_2^0) & \frac{\partial f_1}{\partial x_2}(x_1^0, x_2^0) \\ \\ \frac{\partial f_2}{\partial x_1}(x_1^0, x_2^0) & \frac{\partial f_2}{\partial x_2}(x_1^0, x_2^0) \end{bmatrix}$$

If $\mathbf{x}$ is sufficiently close to the root, $\mathbf{f}(\mathbf{x}) \approx 0$ so:

$$\mathbf{x} = \mathbf{x^0} - \mathbf{D^{-1}}(\mathbf{x}^0) \cdot \mathbf{f}(\mathbf{x}^0)$$

In practice, it is better to determine a step k in the direction specified by $\mathbf{D^{-1}}(\mathbf{x}^0) \cdot \mathbf{f}(\mathbf{x}^0)$:

$$\mathbf{x} = \mathbf{x^0} - k \cdot \mathbf{D^{-1}}(\mathbf{x}^0) \cdot \mathbf{f}(\mathbf{x}^0)$$

The determination of k is performed by line minimization applied to the sum of squared functions:

$$S(\mathbf{x}) = \sum_{i=1}^{n} f_i(\mathbf{x})^2$$

This method is implemented in unit `unewteqs` as:

```
NewtEqs(Equations, Jacobian, X, F, Lb, Ub, MaxIter, Tol)
```

where `Equations` and `Jacobian` are the procedures which compute the function vector and the jacobian matrix, respectively. Their syntaxes are:

```
procedure Equations(X, F : TVector);
procedure Jacobian(X : TVector; D : TMatrix);
```

They correspond to types `TEquations` and `TJacobian`, respectively.

The user must provide the initial approximations to the roots in vector `X[Lb..Ub]`. After refinement by the procedure, the corresponding function values are returned in vector `F`.

The possible error codes returned by `MathErr` are:

- `OptOk` if no error occurred
- `OptNonConv` (non-convergence) if the number of iterations exceeds the maximum value `MaxIter`
- `OptSing` if the jacobian matrix is quasi-singular

Approximate jacobian

Approximate values of the jacobian matrix may be computed using finite difference approximations:

$$\frac{\partial f_i}{\partial x_j}(\mathbf{x}) \approx \frac{f_i(x_j + h_j) - f_i(x_j - h_j)}{2h_j}$$

The increment h_j is such that $h_j = \eta \mid x_j \mid$ where η is a constant which should not be less than the square root of the machine epsilon ($\texttt{MachEp}^{1/2}$).

The demo program `testnr.pas` gives an example of using such a procedure.

As usual, it is recommended to use analytical expressions for the derivatives whenever possible.

11.2.2 Broyden's method

This method is similar to the BFGS method of function minimization. It can also be viewed as a multidimensional version of the secant method.

Broyden's algorithm uses the following formula to construct the inverse jacobian matrix iteratively:

$$\mathbf{D}_{i+1}^{-1} = \mathbf{D}_i^{-1} + \frac{\left[(\delta\mathbf{x} - \mathbf{D}_i^{-1} \cdot \delta\mathbf{f}) \cdot \delta\mathbf{x}^\top\right] \cdot \mathbf{D}_i^{-1}}{\delta\mathbf{x}^\top \cdot \mathbf{D}_i^{-1} \cdot \delta\mathbf{f}}$$

with:

$$\delta\mathbf{x} = \mathbf{x}_{i+1} - \mathbf{x}_i \qquad \delta\mathbf{f} = \mathbf{f}(\mathbf{x}_{i+1}) - \mathbf{f}(\mathbf{x}_i)$$

The algorithm is usually started with the identity matrix ($\mathbf{D}_0^{-1} = \mathbf{I}$).

This method is implemented in unit `ubroyden` as:

```
Broyden(Equations, X, F, Lb, Ub, MaxIter, Tol)
```

where the parameters have the same significance than in `NewtEqs`.

The possible error codes returned by `MathErr` are `OptOk` and `OptNonConv`.

11.3 Demo programs

These programs are located in the `demo\console\equation` subdirectory.

Equations in one variable

The demo programs `testbis.pas`, `testsec.pas` and `testnr1.pas` demonstrate the bisection, secant and Newton-Raphson methods, respectively, on the equation:

$$f(x) = x \ln x - 1 = 0$$

for which the derivative is:

$$f'(x) = \ln x + 1$$

The true solution is $x = 1.763222834...$

Equations in several variables

The demo programs `testnr.pas` and `testbrdn.pas` demonstrate the Newton-Raphson and Broyden methods, respectively, on the following system (taken from the *Numerical Recipes* example book) :

$$f(x, y) = x^2 + y^2 - 2 = 0$$

$$g(x, y) = \exp(x - 1) + y^3 - 2 = 0$$

for which the jacobian is:

$$\mathbf{D}(x, y) = \begin{bmatrix} 2x & 2y \\ \exp(x-1) & 3y^2 \end{bmatrix}$$

The true solution is $(x, y) = (1, 1)$.

Chapter 12

Polynomials

This chapter describes the procedures and functions related to polynomials and rational fractions.

12.1 Polynomials

Function `Poly(X, Coef, Deg)`, defined in unit `upolynom`, evaluates the polynomial:

$$P(X) = \mathrm{Coef}[0] + \mathrm{Coef}[1] \cdot X + \mathrm{Coef}[2] \cdot X^2 + \cdots + \mathrm{Coef}[\mathrm{Deg}] \cdot X^{\mathrm{Deg}}$$

12.2 Rational fractions

Function `RFrac(X, Coef, Deg1, Deg2)`, also defined in `upolynom`, evaluates the rational fraction:

$$F(X) = \frac{\mathrm{Coef}[0] + \mathrm{Coef}[1] \cdot X + \cdots + \mathrm{Coef}[\mathrm{Deg1}] \cdot X^{\mathrm{Deg1}}}{1 + \mathrm{Coef}[\mathrm{Deg1}+1] + \cdots + \mathrm{Coef}[\mathrm{Deg1}+\mathrm{Deg2}] \cdot \mathrm{X}^{\mathrm{Deg2}}}$$

12.3 Roots of polynomials

Analytical methods can be used to compute the roots of polynomials up to degree 4. For higher degrees, iterative methods must be used.

12.3.1 Analytical methods

- Function `RootPol1(A, B, X)`, defined in unit `urtpol1`, solves the linear equation $A + BX = 0$. The function returns 1 if no error occurs

$(B \neq 0)$, -1 if X is undetermined $(A = B = 0)$, -2 if there is no solution $(A \neq 0, B = 0)$.

- Functions `RootPol`n`(Coef, Z)`, with $n = 2, 3, 4$, solve the equation:

  $$\text{Coef}[0] + \text{Coef}[1] \cdot X + \text{Coef}[2] \cdot X^2 + \cdots + \text{Coef}[\text{N}] \cdot X^{\text{N}} = 0$$

  These functions are defined in their respective units `urtpol`n.

  The roots are stored in the complex vector `Z`. The real part of the i^{th} root is in `Z[i].X`, the imaginary part in `Z[i].Y`.

  If no error occurs, the function returns the number of real roots, otherwise it returns (-1) or (-2) just like `RootPol1`.

12.3.2 Iterative method

Function `RootPol(Coef, Deg, Z)`, defined in unit `urootpol`, solves the polynomial equation:

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = 0$$

by the method of the *companion matrix*.

The companion matrix **A** is defined by:

$$\mathbf{A} = \begin{bmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \cdots & -\frac{a_1}{a_n} & -\frac{a_0}{a_n} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

It may be shown that the eigenvalues of this matrix are equal to the roots of the polynomial (Eigenvalues are treated in § 9.10).

The coefficients of the polynomial are passed in vector `Coef`, such that `Coef[0]` $= a_0$, `Coef[2]` $= a_1$ etc. The degree of the polynomial is passed in `Deg`. The roots are returned in the complex vector `Z` as described before.

If no error occurred, the function returns the number of real roots.

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

12.4 Ancillary functions

Two procedures have been added in unit `upolutil` to facilitate the handling of polynomials roots:

- Function `SetRealRoots(Deg, Z, Tol)` allows to set the imaginary part of a root to zero if it is less than a fraction `Tol` of the real part. The function returns the total number of real roots.

  Due to roundoff errors, some real roots may be computed with a very small imaginary part, e.g. $1 + 10^{-8}i$. The function `SetRealRoots` tries to correct this problem.

- Procedure `SortRoots(Deg, Z)` sort the roots such that:

  1. The `N` real roots are stored in elements `1..N` of vector `Z`, in increasing order.
  2. The complex roots are stored in elements `(N + 1)..Deg` of vector `Z` and are unordered.

12.5 Demo programs

12.5.1 Console programs

These programs are located in the `demo\console\polynom` subdirectory.

Evaluation of a polynomial

Program `evalpoly.pas` evaluates a polynomial for a series of user-specified values. Entering 0 stops the program.

Evaluation of a rational fraction

Program `evalfrac.pas` performs the same task as the previous program, but with a rational fraction.

Roots of a polynomial

Program `polyroot.pas` computes the roots of a polynomial with real coefficients. Analytical methods are used up to degree 4, otherwise the method of the companion matrix is used.

The example polynomial is:

$$x^6 - 21x^5 + 175x^4 - 735x^3 + 1624x^2 - 1764x + 720$$

for which the roots are 1, 2 ... 6

12.5.2 GUI program

Program `polysolve.dpr`, located in `demo\gui\polysolve`, solves a polynomial with real coefficients, up to degree 20. The roots are ordered according to the `SortRoots` procedure. The complex part can be neglected if it is lower than a specified fraction of the real part, as explained for the `SetRealRoots` function. Once the roots have been computed, the real and imaginary parts of the polynomial are displayed for each root.

The following buttons are available:

- **Load** and **Save** : reads or writes the coefficients from/to a text file (example: `coef.dat`)
- **Numeric format** : sets the printing format for the results according to the function `SetFormat` (see chapter 8).
- **Solve polynomial**
- **Save roots** : stores the roots in a text file, using the settings from 'Numeric format'. Each root is stored in a single line (real part followed by imaginary part). The polynomial values are not stored.

Chapter 13

Numerical integration and differential equations

This chapter describes the procedures available in DMath to integrate a function of one variable, and to solve systems of differential equations.

13.1 Integration

13.1.1 Trapezoidal rule

The trapezoidal rule approximates the integral I of a tabulated function by the formula:

$$I \approx \frac{1}{2} \sum_{i=0}^{N-1} (x_{i+1} - x_i)(y_{i+1} + y_i)$$

where (x_i, y_i) are the coordinates of the i^{th} point.

This procedure is implemented in unit `utrapint` as function `TrapInt(X, Y, N)`. Note that the lower bound of the arrays must be 0.

13.1.2 Gauss-Legendre integration

This method approximates the integral of a function f in an interval $[a, b]$ by:

$$\int_a^b f(x)dx \approx \frac{b-a}{2} \sum_{i=1}^{N} w_i f(y_i)$$

$$y_i = \frac{b-a}{2} x_i + \frac{b+a}{2}$$

The abscissae x_i and weights w_i are predefined values for a given number of points N.

This method is implemented in unit `ugausleg` as function `GausLeg(Func, A, B)` for $N = 16$. Function `Func` must be declared as:

```
function Func(X : Float) : Float;
```

For the special case $A = 0$ there is a variant `GausLeg0(Func, B)`.

13.2 Convolution

The convolution product of two functions f and g is defined by:

$$(f * g)(t) = \int_0^t f(u)g(t-u)du$$

This product is often used to describe the ouput of a linear system when $f(t)$ is the input signal (function of time) and $g(t)$ is the impulse response of the system.

- Function `Convol(Func1, Func2, T)`, defined in unit `ugausleg`, approximates the convolution product of the two functions `Func1` and `Func2` at time `T` by the Gauss-Legendre method. The functions must be declared as above.

- Procedure `ConvTrap(Func1, Func2, T, Y, N)`, defined in unit `utrapint`, approximates the convolution product of the two functions `Func1` and `Func2` over a range of equally spaced times `T[0..N]` by the trapezoidal rule. The results are returned in `Y[0..N]`.

13.3 Differential equations

The Runge-Kutta-Fehlberg (RKF) method allows to compute numerical solutions to systems of first-order differential equations of the form:

$$\begin{aligned} y_1'(t) &= f_1[t, y_1(t), y_2(t), \cdots] \\ y_2'(t) &= f_2[t, y_1(t), y_2(t), \cdots] \\ &\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \end{aligned}$$

where the f_i are known functions and the y_i are to be determined.

The RKF procedure is an extension of the classical Runge-Kutta method. For instance, in the case of a single differential equation

$$y'(t) = f[t, y(t)]$$

this method generates a sequence $\{t_n, y_n\}$ which approximates the function $y(t)$.

The order of the method corresponds to the number of points used in the interval $[t_n, t_{n+1}]$. For instance, the sequence generated by the 4-th order Runge-Kutta method is defined by:

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

with:

$$\begin{aligned}
k_1 &= h \cdot f(t_n, y_n) \\
k_2 &= h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\
k_3 &= h \cdot f\left(t_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \\
k_4 &= h \cdot f(t_n + h, y_n + k_3)
\end{aligned}$$

with $h = t_{n+1} - t_n$

In the RKF method, the step size h is automatically varied so as to maintain a given level of precision on the estimated y values.

The procedure `RKF45`, defined in unit `urkf`, is a translation of a Fortran program by H. A. Watts and L. F. Shampine (`http://www.csit.fsu.edu/~burkardt/f_src/rkf45/rkf45.f90`). It is intermediate between the 4-th and 5-th order Runge-Kutta methods, hence the name RKF45.

In order to use this procedure you must:

1. Define the following variables (the names are optional):

```
var
  Neqn           : Integer;  { Number of equations }
  Y, Yp          : TVector;  { Functions and derivatives }
  Tstart, Tstop  : Float;    { Integration interval }
  Nstep          : Integer;  { Number of steps }
  StepSize       : Float;    { Step size }
  AbsErr, RelErr : Float;    { Abs. and relative errors }
  Flag           : Integer;  { Error flag }
  T, Tout        : Float;    { Integration times }
  I              : Integer;  { Loop variable }
```

2. Define a procedure for computing the system of differential equations:

   ```
   procedure DiffEq(T : Float; Y, Yp : TVector);
   begin
     Yp[1] := Y[2];
     Yp[2] := - Y[1];
   end;
   ```

   (There is a special type `TDiffEqs` for such procedures)

3. Initialize variables, compute the step size and call RKF45 for each integration step (the initial values are given as examples, except for `Flag` which must be initialized to 1):

   ```
   begin
     Neqn := 2;

     DimVector(Y, Neqn);
     DimVector(Yp, Neqn);

     Y[1] := 1;  { Initial conditions }
     Y[2] := 0;

     Tstart := 0;
     Tstop  := 2 * Pi;
     Nstep  := 12;

     StepSize := (Tstop - Tstart) / Nstep;

     AbsErr := 1.0E-6;
     RelErr := 1.0E-6;

     Flag := 1;

     T := Tstart;
   ```

```
    for I := 1 to Nstep do
      begin
        Tout := T + StepSize;
        RKF45(DiffEq, Neqn, Y, Yp, T, Tout, RelErr, AbsErr, Flag);
        T := Tout;
      end;
  end.
```

Upon return from RKF45:

- `Y`, `Yp` contain the values of the functions and their first derivatives at `Tout`
- `Flag` contains an error code:
    - 2 : no error
    - 3 : too small `RelErr` value
    - 4 : too much function evaluations needed
    - 5 : too small `AbsErr` value
    - 6 : the requested accuracy could not be achieved
    - 7 : the method was unable to solve the problem
    - 8 : invalid input parameters

If an error occurs, it should be possible in most cases to restart the computation, using the values returned by the subroutine in `RelErr` and `AbsErr`.

Note : RKF45 may be used to compute a definite integral:

$$\int_a^b f(t)dt = F(b) - F(a)$$

since this is equivalent to integrate the differential equation:

$$F'(t) = f(t)$$

between a and b, with the initial condition specified by $f(a)$.

13.4 Demo programs

These programs are located in the `demo\console\integral` subdirectory.

- Program `trap.pas` applies the trapezoidal rule to a tabulated function.

  The example function $f(x) = e^{-x}$ is tabulated for $x = 0$ to 1 by steps of 0.1. The integral is:

  $$\int_0^1 e^{-x} dx = 1 - e^{-1} \approx 0.6321$$

- Program `gauss.pas` demonstrates the Gauss-Legendre integration method.

  The example function is $f(x) = xe^{-x}$. The integral is:

  $$\int_0^x f(t)dt = 1 - (x+1)e^{-x}$$

- Program `conv.pas` computes the convolution of two functions by the Gauss-Legendre method.

  The example functions are $f(x) = xe^{-x}$ and $g(x) = e^{-2x}$. The convolution product is:

  $$(f * g)(x) = \int_0^x f(u)g(x-u)du = e^{-2x} \int_0^x ue^u du = (x-1)e^{-x} - e^{-2x}$$

- Program `convtrap.pas` computes the same convolution product by the trapezoidal rule.

- Program `test_rkf.pas` solves 3 systems of differential equations by the RKF method:

  1. A single nonlinear equation:

     $$y'(t) = 0.25 \cdot y(t) \cdot [1 - 0.05 \cdot y(t)]$$

     with the initial condition $y(0) = 1$.

     The analytic solution is:

     $$y(t) = \frac{20}{1 + 19\exp(-0.25t)}$$

2. A system of two linear equations:

$$y_1'(t) = y_2(t)$$
$$y_2'(t) = -y_1(t)$$

with the initial conditions $y_1(0) = 1, y_2(0) = 0$.

The analytic solution is:

$$y_1(t) = \cos t \qquad y_2(t) = -\sin t$$

3. A system of 5 equations with one nonlinear:

$$y_1'(t) = y_2(t)$$
$$y_2'(t) = y_3(t)$$
$$y_3'(t) = y_4(t)$$
$$y_4'(t) = y_5(t)$$
$$y_5'(t) = 45 \cdot y_3(t) \cdot y_4(t) \cdot y_5(t) - \frac{40[y_4(t)]^3}{9[y_3(t)]^2}$$

with initial conditions $y_i(0) = 1 \quad \forall i$

The program prints the numeric solution, and, if possible, the analytic one.

Chapter 14

Fourier transform

This chapter describes the procedures available in DMath to perform Fourier transforms. These procedures are defined in unit `ufft`, which is based on a previous work by Don Cross (`http://groovit.disjunkt.com/analog/time-domain/fft.html`).

14.1 Introduction

Fourier transform is a mathematical method which allows to determine the frequency spectrum of a given signal (for instance a sound). The mathematical definition is the following :

$$y(f) = \int_{-\infty}^{\infty} x(t)\exp(2\pi i f t)dt = \int_{-\infty}^{\infty} x(t)(\cos 2\pi f t + i \sin 2\pi f t) \tag{14.1}$$

where $x(t)$ is the input signal (function of time), f the frequency, and i the complex number such that $i^2 = -1$. y is the Fourier transform of x.

The input signal may have real or complex values. However, the Fourier transform is always a complex number. For each frequency f, the modulus of $y(f)$ represents the energy associated with this frequency. A plot of this modulus as a function of f gives the frequency spectrum of the input signal.

If the input signal is sampled as a sequence of n values $x_0, x_1, ..., x_{n-1}$, taken at constant time intervals, the Fourier transform is a sequence of complex number $y_0, y_1, ..., y_{n-1}$, such that:

$$y_p = \sum_{k=0}^{n-1} x_k \left[\cos\left(2\pi\frac{kp}{n}\right) + i \sin\left(2\pi\frac{kp}{n}\right)\right] \tag{14.2}$$

This formula allows, in principle, to compute the transform y_p at any point. In practice, a faster algorithm called the Fast Fourier Transform (FFT) is used.

14.2 Programming

14.2.1 Array dimensioning

The FFT algorithm requires that the number of points n is a power of 2. Moreover, the arrays must be dimensioned from 0 to $(n-1)$. For instance:

```
const
  NumSamples = 512;             { Buffer size must be power of 2 }
  MaxIndex   = NumSamples - 1;  { Max. array index }

var
  InArray, OutArray : TCompVector;

begin
  DimVector(InArray, MaxIndex);   { FFT input }
  DimVector(OutArray, MaxIndex);  { FFT output }
  ...
```

The maximal value of n depends on the maximal array size allowed by the compiler (see § 9.2). For Delphi (32 bits, double precision), this value is $2^{26} = 67108864$ (64 mega) points.

14.2.2 FFT procedures

- Procedure `FFT(NumSamples, InArray, OutArray)` calculates the Fast Fourier Transform of the array of complex numbers `InArray` to produce the output complex numbers in `OutArray`.
- Procedure `IFFT(NumSamples, InArray, OutArray)` calculates the Inverse Fast Fourier Transform of the array of complex numbers represented by `InArray` to produce the output complex numbers in `OutArray`.

  In other words, this procedure reconstitutes the input signal from its FFT.
- Procedure `FFT_Integer(NumSamples, RealIn, ImagIn, OutArray)` computes the Fast Fourier Transform on integer data. Here the real and

imaginary parts of the data are stored in two integer arrays `RealIn` and `ImagIn`, while the results are stored in the complex array `OutArray`.

- Procedure `FFT_Integer_Cleanup` clears the memory after a call to `FFT_Integer`.
- Function `CalcFrequency(NumSamples, FrequencyIndex, InArray)` calculates the complex frequency sample at a given index directly, by means of eq. 14.2. 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 Fourier Transform of a number of data which is not an integer power of 2. For example, you could calculate the transform of 100 points instead of rounding up to 128 and padding the extra 28 array slots with zeroes.

14.3 Demo programs

14.3.1 Console program

Program `freq.pas`, located in `demo\console\fourier`, compares the FFT with the direct computation of the frequencies, on a set of random data. Results are stored in the output file `freq.txt`

14.3.2 BGI programs

These programs are located in `demo\bgi\fourier`

- Program `testfft.pas` generates a time signal consisting of a large 200 Hz sine wave added to a small 2000 Hz cosine wave. Next, it performs the FFT and graphs the resulting complex frequency samples. Results are stored in the output file `testfft.txt`

  The sampling frequency `SamplingRate` is 22050 Hz, the number of points `NumSamples` is 512 (= 2^9). These two numbers determine the time and frequency units:

  ```
  DT := 1 / SamplingRate;            { Time unit }
  DF := SamplingRate / NumSamples;   { Frequency unit }
  ```

  so that the entry `InArray[I]` in the input array of procedure `FFT` corresponds to the signal value at time `I * DT`, and the entry `OutArray[I]` in the output array corresponds to the Fourier transform at frequency `I * DF`.

The highest frequency which may be detected is equal to `SamplingRate/2` and is called *Nyquist's frequency.* Hence, only the first half of array `OutArray` needs to be plotted (the second half contains symmetric values).

The program generates the input signal, plots it, then performs the FFT and plots the real and imaginary parts as a function of frequency. The plot shows two peaks, corresponding to the 5-th and 46-th entries in `OutArray` (as seen from the file `fftout.txt`). The corresponding frequencies are:

$$5 \times \frac{22050}{512} \approx 215 \text{ Hz}$$
$$46 \times \frac{22050}{512} \approx 1981 \text{ Hz}$$

The high peak corresponds to the main signal and the small peak to the parasite.

- Program `filter.pas` uses the same signal as the previous program and filters the parasite by setting to zero all the FFT values corresponding to the frequencies higher than 1000 Hz, according to the following code:

  ```
  FreqIndex := Trunc(1000.0 / DF);
  SymIndex := NumSamples - FreqIndex;

  for I := FreqIndex to SymIndex do
    begin
      OutArray[I].X := 0.0;
      OutArray[I].Y := 0.0;
    end;
  ```

  Note that the two halves of the output array, on either side of Nyquist's frequency, must be treated.

  The program then calls procedure `IFFT` to compute the inverse Fourier transform of the modified data and plots the result, showing that the parasite has been removed, at the expense of a slight distorsion of the main signal.

Chapter 15

Random numbers

This chapter describes the procedures and functions available to generate random numbers and perform stochastic simulation and optimization.

15.1 Random numbers

15.1.1 Introduction

DMath provides three random number generators (RNG) :

- the 'Multiply With Carry' (MWC) generator of George Marsaglia.
- the 'Mersenne Twister' (MT) generator of Takuji Nishimura and Makoto Matsumoto (`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`).
- the 'Universal Virtual Array Generator' (UVAG) contributed by Alex Hay.

The first method produces a sequence $\{I_n\}$ of integer numbers by means of the following recurrence relationships:

$$I_{n+1} = (aI_n + c_n) \bmod b$$

$$c_{n+1} = (aI_n + c_n) \operatorname{div} b$$

where a is the *multiplier*, b the *base* (such that $b = 2^k$), c_n the *carry*. One may start with $c_0 = 0$.

If a is properly chosen, the period of the generator is $a \times 2^{k-1} - 1$. In our implementation, a 32-bit integer is generated by concatenating two 16-bit integers, with $a_1 = 18000$ and $a_2 = 30903$. The period of the 32-bit generator is therefore:

$$(a_1 \times 2^{15} - 1) \times (a_2 \times 2^{15} - 1) \approx 6 \times 10^{17}$$

The second method is more complex and slightly slower but it may be safer for intensive simulations since it has a much longer period (about 10^{6000}) and produces uncorrelated numbers in 623 dimensions.

We have verified that the three generators pass Marsaglia's DIEHARD battery of tests (`http://stat.fsu.edu/pub/diehard/`).

15.1.2 Type definition

Type `RNG_Type` (defined in unit `utypes`) is used to identify the generator, with 3 possible values: `RNG_MWC, RNG_MT, RNG_UVAG`

15.1.3 Generic functions

These functions are defined in unit `urandom` and may be used with any of the three generators.

Choice of generator

The generator is chosen by using one of the statements: `SetRNG(RNG_MWC)`, `SetRNG(RNG_MT)` or `SetRNG(RNG_UVAG)`. A default initialization is performed at the same time.

Initialization

The selected generator can be initialized with the statement `InitGen(Seed)` where `Seed` is an integer.

Uniform random numbers

The following functions are available:

Function	Type	Bits	Domain
`IRanGen`	`Integer`	32	[-2147483648, 2147483647]
`IRanGen31`	`Integer`	31	[0, 2147483647]
`RanGen1`	`Float`	32	[0, 1]
`RanGen2`	`Float`	32	[0, 1)
`RanGen3`	`Float`	32	(0, 1)
`RanGen53`	`Float`	53	[0, 1)

15.1.4 Specific functions

The following functions are specific of a given generator.

MWC generator

These functions are defined in unit `uranmwc`

- Procedure `InitMWC(Seed)` initializes the MWC generator with an integer.

  The default initialization performed by `SetRNG` corresponds to `Seed` = 118105245.

- Function `IRanMWC` returns a random integer from the MWC generator.

MT generator

These functions are defined in unit `uranmt`

- Procedure `InitMT(Seed)` initializes the MT generator with an integer.
- Procedure `InitMTbyArray(InitKey, KeyLength)` initializes the MT generator with an array `InitKey[0..(KeyLength - 1)]` of unsigned integers, with `KeyLength` < 624.

  The default initialization performed by `SetRNG` corresponds to the vector `($123, $234, $345, $456)`.

- Function `IRanMT` returns a random integer from the MT generator.

UVAG generator

These functions are defined in unit `uranuvag`

- Procedure `InitUVAGbyString(KeyPhrase)` initializes the UVAG generator with a string `KeyPhrase`.

  The default initialization performed by `SetRNG` corresponds to the string `'abcd'`.

- Procedure `InitUVAG(Seed)` initializes the UVAG generator with an integer.
- Function `IRanUVAG` returns a signed integer from the UVAG generator.

15.1.5 Gaussian random numbers

These functions use the selected generator.

Normal distribution

These functions are defined in unit `urangaus`.

- Function `RanGaussStd` generates a random number from the standard normal distribution.

  The Box-Muller algorithm is used: if x_1 and x_2 are two uniform random numbers $\in (0, 1)$, the two numbers y_1 and y_2 defined by:

  $$y_1 = \sqrt{-2 \ln x_1} \cos 2\pi x_2 \qquad y_2 = \sqrt{-2 \ln x_1} \sin 2\pi x_2$$

  follow the standard normal distribution.

- Function `RanGauss(Mu, Sigma)` generates a random number from the normal distribution with mean `Mu` and standard deviation `Sigma`.

Multinormal distribution

These functions are defined in unit `uranmult`.

- Procedure `RanMult(M, L, Lb, Ub, X)` generates a random vector `X` from a multidimensional normal distribution. `M[Lb..Ub]` is the mean vector, `L[Lb..Ub, Lb..Ub]` is the Cholesky factor of the variance-covariance matrix.

  To simulate the n-dimensional multinormal distribution $\mathcal{N}(\mathbf{m}, \mathbf{V})$, where $\mathbf{m}$ is the mean vector and $\mathbf{V}$ the variance-covariance matrix, the following algorithm is used:

  1. Let $\mathbf{u}$ be a vector of n independent random numbers following the standard normal distribution,
  2. Let $\mathbf{L}$ be the lower triangular matrix resulting from the Cholesky factorization of matrix $\mathbf{V}$,
  3. Vector $\mathbf{x} = \mathbf{m} + \mathbf{Lu}$ follows the multinormal distribution $\mathcal{N}(\mathbf{m}, \mathbf{V})$.

- Procedure `RanMultIndep(M, S, Lb, Ub, X)` generates a random number from an uncorrelated multidimensional distribution. Here `S` is simply the vector of standard deviations.

15.2 Markov Chain Monte Carlo

It is not always possible to simulate the distribution of a random variable with a direct algorithm such as the ones used for normal or multinormal distributions.

However, there exist iterative algorithms which generate a sequence of random variables for which the distribution tend towards the desired distribution, after starting from a standard distribution (e. g. uniform).

These random sequences are known as *Markov chains* and the iterative simulation method is therefore known as *Markov chain Monte-Carlo* (MCMC).

There are several MCMC variants. Here we will present the Metropolis-Hastings method.

Let $\mathbf{X}$ a vector of random variables and $P(\mathbf{X})$ its probability density function (p.d.f.), which is to be simulated. The classical formulation of the Metropolis-Hastings algorithm is the following:

1. Choose an initial parameter vector $\mathbf{X}_0$
2. At iteration n:
   (a) Draw a vector $\mathbf{u}$ from the multinormal distribution $\mathcal{N}(\mathbf{X}_{n-1}, \mathbf{V})$ where $\mathbf{V}$ is the variance-covariance matrix
   (b) If $r = P(\mathbf{u})/P(\mathbf{X}_{n-1}) > 1$, set $\mathbf{X}_n = \mathbf{u}$
   otherwise if $Random(0,1) < r$, set $\mathbf{X}_n = \mathbf{u}$
   where $Random(0,1)$ denotes a uniform random number in the interval [0,1]
3. Set $n = n + 1$; goto 2

It is convenient to introduce a function $F(\mathbf{X})$ such that:

$$P(\mathbf{X}) = C \exp\left[-\frac{F(\mathbf{X})}{T}\right] \qquad \Longleftrightarrow \qquad F(\mathbf{X}) = -T \ln \frac{P(\mathbf{X})}{C} \tag{15.1}$$

where C and T are positive constants. By analogy with statistical thermodynamics, T is known as the *temperature*.

From this equation, it may be seen that:

$$r = \frac{P(\mathbf{u})}{P(\mathbf{X}_{n-1})} = \exp\left(-\frac{\Delta F}{T}\right)$$

where

$$\Delta F = F(\mathbf{u}) - F(\mathbf{X}_{n-1})$$

so, the Metropolis-Hastings algorithm may be rewritten as:

1. Choose an initial parameter vector $\mathbf{X}_0$
2. At iteration n:
   (a) Draw a vector $\mathbf{u}$ from the multinormal distribution $\mathcal{N}(\mathbf{X}_{n-1}, \mathbf{V})$
   Set $\Delta F = F(\mathbf{u}) - F(\mathbf{X}_{n-1})$
   (b) if $\Delta F < 0$, set $\mathbf{X}_n = \mathbf{u}$
   otherwise if $Random(0, 1) < \exp(-\Delta F/2)$, set $\mathbf{X}_n = \mathbf{u}$
3. Set $n = n + 1$; goto 2

The initial variance-covariance matrix $\mathbf{V}$ may be diagonal and its elements may be given large values, so that the initial distribution spans a relatively large space. When the iterations progress, the matrix converges to the variance-covariance matrix of the simulated distribution. It is often useful to perform several cycles of the algorithm, with the variance-covariance matrix being re-evaluated at the end of each cycle.

The vector $\mathbf{X}$ corresponding to the lowest value of F is recorded; hence, the algorithm may be used as a stochastic optimization algorithm for minimizing the function F. The advantage of such an algorithm is that it can 'escape' from a local minimum (with a probability equal to $e^{-\Delta F/T}$) and has therefore more chances to reach the global minimum, unlike the deterministic optimizers studied in chapter 10, for which only decreases of the function are acceptable. This application is however restricted by the fact that the function F must be linked to a p.d.f. by means of eq. (15.1).

This method is implemented in unit `umcmc` as:

```
Hastings(Func, T, X, V, Lb, Ub, Xmat, X_min, F_min)
```

The user must provide :

- the function `Func` to be minimized (defined as in paragraph 10.2, p. 72)
- the temperature `T`
- a starting vector `X[Lb..Ub]`

- a starting variance-covariance matrix `V[Lb..Ub, Lb..Ub]`.

On output, `Hastings` returns:

- the mean of the simulated distribution in `X`
- its variance-covariance matrix in `V`
- a matrix of simulated vectors in `Xmat` (one vector by line)
- the vector which minimizes the function in `X_min`
- the value of the function at the minimum in `F_min` (corresponds to the mode of the simulated distribution).

The behavior of the algorithm can be controlled with the following procedure:

```
InitMHParams(NCycles, MaxSim, SavedSim)
```

where:

- `NCycles` is the number of cycles (default = 10)
- `MaxSim` is the maximum number of simulations at each cycle (default = 1000)
- `SavedSim` is the number of simulated vectors which are saved in matrix `Xmat`. Only the last `SavedSim` vectors from the last cycle are saved. (default = 1000)

The current values of these parameters can be retrieved with the procedure `GetMHParams(NCycles, MaxSim, SavedSim)`.

After a call to `Hastings`, function `MathErr` will return one of the following codes:

- `OptOk` if no error occurred
- `MatNotPD` if the variance-covariance matrix is not positive definite

The random number generator is re-initialized at the start of the algorithm, so that a different result will be obtained for each call of the subroutine.

15.3 Simulated Annealing

Simulated annealing (SA) is an extension of the Metropolis-Hastings algorithm which tries to find the global minimum of any function (not necessarily a p.d.f.). Here the temperature starts from a high value and is progressively decreased as the algorithm progresses towards the minimum. The optimized parameters may then be refined with a local optimizer (chapter 10).

The implementations used in DMath is a modification of a Fortran program written by B. Goffe (`http://www.netlib.org/simann`).

With the notations:

$F(\mathbf{X})$	:	function to be minimized
$\delta\mathbf{X}$	:	range of $\mathbf{X}$
F_{min}	:	minimum of $F(\mathbf{X})$
T	:	temperature
N_T	:	number of loops at constant T
N_S	:	number of loops before adjustement of $\delta\mathbf{X}$
R_T	:	temperature reduction factor
N_{acc}	:	number of accepted function increases

the algorithm may be described as follows:

```
• initialize $T, \mathbf{X}, \delta\mathbf{X}$
• repeat
    ◦ repeat $N_T$ times
        ⋆ repeat $N_S$ times
            for each parameter $X_i$ :
            ◇ pick a random value $X_i'$ in the interval $X_i \pm \delta X_i$
            ◇ compute $F(X_i')$
            ◇ accept or reject $X_i'$ according to Metropolis criterion
            ◇ update $N_{acc}$
            ◇ update $F_{min}$ if necessary
        ⋆ adjust step length $\delta X_i$ so as to maintain an acceptance ratio
          of about 50%
    ◦ $T \leftarrow T \cdot R_T$
• until $N_{acc} = 0$ or $T < T_{min}$ or $|F_{min}| < \epsilon$
```

The threshold values T_{min} and ϵ are fixed at 10^{-30} in our implementation.

At the beginning of the iterations, while we are away from the minimum, it makes sense to choose a high probability of acceptance, for instance $p = \frac{1}{2}$. It is then possible to perform a given number of random drawings and to compute the median M of the increases of function F, from which the initial temperature T_0 is deduced by:

$$p = \exp\left(-\frac{M}{T_0}\right) = \frac{1}{2} \qquad \Rightarrow \qquad T_0 = \frac{M}{\ln 2}$$

This procedure is implemented in unit `usimann` as:

```
SimAnn(Func, X, Xmin, Xmax, Lb, Ub, F_min)
```

where:

- `Func` is the function to be minimized (defined as in paragraph 10.2, p. 72)
- `X[Lb..Ub]` is the parameter vector
- `Xmin, Xmax` are the bound values of `X`

The optimized parameters are returned in `X` and the corresponding function value in `F_min`

The user must provide reasonable values of `Xmin` and `Xmax` as well as a starting value for `X`. It is convenient to pick a random value in the range specified by `Xmin` and `Xmax`.

The behavior of the algorithm can be controlled with the following procedure:

```
InitSAParams(NT, NS, NCycles, RT)
```

where:

- `NT, NS, RT` correspond to the variables N_T, N_S and R_T in the algorithm. Default values are 5, 15 and 0.9 respectively.
- `NCycles` is the number of cycles (default = 1).

In some difficult situations, it may be useful to perform several cycles of the algorithm. Each cycle will start with the optimized parameters `X` from the previous cycle and the temperature will be re-initialized (the bound values `Xmin, Xmax` remaining the same).

It is possible to record the progress of the iterations in a log file. This file is created with:

```
SA_CreateLogFile(LogFileName)
```

If the file is created, the following information will be stored:

- iteration number (each iteration corresponds to a single temperature)
- temperature value
- lowest function value obtained at this temperature
- number of function increases
- number of accepted increases

The file will be automatically closed upon return from `SimAnn`.

15.4 Genetic Algorithm

Genetic Algorithms (GA) are another class of stochastic optimization methods which try to mimick the law of natural selection in order to optimize a function $F(\mathbf{X})$.

There are several implementations of these algorithms. We use a method described by E. Perrin *et al.* (*Recherche operationnelle / Operations Research*, 1997, **31**, 161-201). In this version, the vector $\mathbf{X}$ is considered as the 'phenotype' of an 'individual' belonging to a 'population'. This phenotype is determined by two 'chromosomes' $\mathbf{C}_1$ and $\mathbf{C}_2$ and a vector of 'dominances' $\mathbf{D}$ such that:

$$X_i = D_i C_{1i} + (1 - D_i) C_{2i} \qquad (0 < D_i < 1) \tag{15.2}$$

A population is defined by a matrix $\mathbf{P}$, such that each row of the matrix corresponds to a vector $\mathbf{X}$.

The population is initialized by taking vectors $\mathbf{C}_1$ and $\mathbf{C}_2$ at random in a given interval, vector $\mathbf{D}$ at random in (0,1) then applying eq. (15.2) to obtain the corresponding $\mathbf{X}$ vectors.

At each step ('generation') of the algorithm:

1. The function values $F(\mathbf{X})$ are computed for each vector $\mathbf{X}$ and the N_S individuals having the lowest function values (the 'survivors') are selected.

2. The remaining individuals are discarded and replaced by new ones, generated as follows:

   (a) Two 'parents' are chosen at random in the selected sub-population and a 'child' is generated by:

      - taking the vectors $\mathbf{C}_1$ and $\mathbf{C}_2$ at random from the parents
      - generating a new vector $\mathbf{D}$
      - computing the new $\mathbf{X}$ according to eq. (15.2)

      This process is repeated until the function value for the child is lower than the lowest function value of the two parents.

   (b) The child is 'mutated' (i. e. its vectors are reinitialized at random) with a probability M_R

   (c) The child is made 'homozygous' (i. e. its vectors $\mathbf{C}_1$ and $\mathbf{C}_2$ are made identical to its vector $\mathbf{X}$) with a probability H_R

This procedure is implemented in unit `ugenalg` as:

```
GenAlg(Func, X, Xmin, Xmax, Lb, Ub, F_min)
```

where the parameters have the same meaning as in `SimAnn`.

The behavior of the algorithm can be controlled with the following procedure:

```
InitGAParams(NP, NG, SR, MR, HR)
```

where:

- `NP` is the population size (default = 200)
- `NG` is the number of generations (default = 40)
- `SR` is the survival rate (default = 0.5)
- `MR` is the mutation rate (default = 0.1)
- `HR` is the probability of homozygosis (default = 0.5)

A log file may also be created with:

```
GA_CreateLogFile(LogFileName)
```

The file will contain the iteration (generation) number and the optimized function value for this generation.

15.5 Demo programs

These programs are located in the `demo\console\random` subdirectory.

Test of MWC generator

Program `testmwc.pas` picks 20000 random numbers and displays the next 6 together with the correct values obtained with the default initialization.

Test of MT generator

Program `testmt.pas` writes 1000 integer numbers and 1000 real numbers from functions `IRanGen` and `RanGen2`, using the default initialization.

The output of this program should be similar to the contents of file `mt.txt`

Test of UVAG generator

Program `testuvag.pas` writes 1000 integer numbers from function `IRanGen`, using the default initialization. The output should be similar to the contents of file `uvag.txt`

File of random numbers

Program `randfile.pas` generates a binary file of 32-bit random integers to be used as input for the DIEHARD program. The user must specify the number of random integers to be generated (default is 3,000,000).

Gaussian random numbers

Program `testnorm.pas` picks a random sample of size N from a gaussian distribution with known mean and standard deviation (SD), estimates mean (m) and SD (s) from the sample, and computes a 95% confidence interval for the mean (i.e. an interval which has a probability of 0.95 to include the true mean), using the formula:

$$\left[m - 1.96\frac{s}{\sqrt{N}}, m + 1.96\frac{s}{\sqrt{N}}\right]$$

This formula is valid for $N > 30$.

Multinormal distribution

Program `ranmul.pas` simulates a multi-normal distribution. The example is a 3-dimensional distribution with the following means, standard deviations, and correlation matrix:

$$\mathbf{m} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \qquad \mathbf{s} = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} 1 & 0.25 & 0.5 \\ 0.25 & 1 & -0.25 \\ 0.5 & -0.25 & 1 \end{bmatrix}$$

These data are stored in the input file `ranmul.dat`. The results of the simulation are stored in file `ranmul.out`

Multi-lognormal distribution

A vector $\mathbf{x}$ is said to follow a multi-lognormal distribution $\mathcal{LN}(\mathbf{m}, \mathbf{V})$ if the vector $\mathbf{x}^\circ$ defined by:

$$x_i^\circ = \ln(x_i)$$

follows a multinormal distribution $\mathcal{N}(\mathbf{m}^\circ, \mathbf{V}^\circ)$

It may be shown that:

$$m_i^\circ = \ln(x_i) - V_{ii}^\circ$$

$$V_{ij}^\circ = \ln\left(1 + \frac{V_{ij}}{m_i m_j}\right)$$

So, if $\mathbf{x}^\circ$ is a random vector drawn from $\mathcal{N}(\mathbf{m}^\circ, \mathbf{V}^\circ)$, $\mathbf{x} = \exp(\mathbf{x}^\circ)$ will be a vector from $\mathcal{LN}(\mathbf{m}, \mathbf{V})$

Program `ranmull.pas` simulates a multi-lognormal distribution. The example is a 2-dimensional distribution with the following means, standard deviations, and correlation coefficient:

$$\mathbf{m} = \begin{bmatrix} 17.4178 \\ 5.3173 \end{bmatrix} \qquad \mathbf{s} = \begin{bmatrix} 6.1259 \\ 2.5158 \end{bmatrix} \quad r = 0.5672$$

These data are stored in the input file `ranmull.dat`. The results of the simulation are stored in file `ranmull.out`

Markov Chain Monte-Carlo

Although MCMC methods are best suited when there is no direct simulation algorithm available, we will use the Metropolis-Hastings method to simulate the previous multinormal distribution (program `testmcmc.pas`).

First, we have to define the function to be optimized. The probability density for a n-dimensional normal distribution $\mathcal{N}(\mathbf{m}, \mathbf{V})$ is:

$$P(\mathbf{X}) = \frac{1}{\sqrt{(2\pi)^n |\mathbf{V}|}} \exp\left[-\frac{1}{2}(\mathbf{X} - \mathbf{m})^\top \mathbf{V}^{-1}(\mathbf{X} - \mathbf{m})\right]$$

So, according to eq. 15.1, $T = 2$ and:

$$F(\mathbf{X}) = (\mathbf{X} - \mathbf{m})^\top \mathbf{V}^{-1}(\mathbf{X} - \mathbf{m})$$

Then, we have to define a starting vector $\mathbf{X}_{sim}$ and variance-covariance matrix $\mathbf{V}_{sim}$. In order to show that the algorithm can converge from a point chosen relatively far away from the optimum, we have chosen $\mathbf{X}_{sim} = 3\mathbf{m}$ and $\mathbf{V}_{sim} = diag(10V_{ii})$.

With the default initializations (10 cycles of 1000 simulations each), the results of a typical run were:

$$\hat{\mathbf{m}} = \begin{bmatrix} 1.01 \\ 2.02 \\ 3.01 \end{bmatrix} \qquad \hat{\mathbf{s}} = \begin{bmatrix} 0.099 \\ 0.210 \\ 0.320 \end{bmatrix} \qquad \hat{\mathbf{R}} = \begin{bmatrix} 1 & 0.286 & 0.467 \\ 0.286 & 1 & -0.299 \\ 0.467 & -0.299 & 1 \end{bmatrix}$$

Simulated Annealing

Program `simann.pas` uses simulated annealing to minimize a set of 10 notoriously difficult functions (most of them presenting multiple minima). Several successive runs of the program may be necessary to have all functions minimized (the random number generator being reinitialized at each call of the `SimAnn` procedure).

Genetic Algorithm

Program `genalg.pas` optimizes the same functions than the previous program but with genetic algorithm. Here, too, it may be necessary to run the program several times.

Chapter 16

Statistics

This chapter describes some of the statistical functions available in DMath. The specific problem of curve fitting will be considered in subsequent chapters.

16.1 Descriptive statistics

16.1.1 Minimum, maximum, mean and standard deviation

The following functions are defined in unit `umeansd` :

- Function `Min(X, Lb, Ub)` returns the minimum of sample `X[Lb..Ub]`
- Function `Max(X, Lb, Ub)` returns the maximum of sample `X[Lb..Ub]`
- Function `Mean(X, Lb, Ub)` returns the mean of sample `X[Lb..Ub]`, defined by:

  $$m = \frac{1}{n} \sum_{i=1}^{n} x_i$$

  where n is the size of the sample.
- Function `StDev(X, Lb, Ub, M)` returns the estimated standard deviation of the population from which sample `X` is extracted, `M` being the mean of the sample. This standard deviation is defined by:

  $$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - m)^2}$$

  These estimated standard deviations are used in statistical tests.

- Function `StDevP(X, Lb, Ub, M)` returns the standard deviation of `X`, *considered as a whole population.* This standard deviation is defined by:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2}$$

16.1.2 Median

Function `Median(X, Lb, Ub, Sorted)`, defined in unit `umedian`, returns the median of `X`, defined as the number x_{med} which has equal numbers of values above it and below it. If the array `X` has been sorted, the median is:

$$\begin{aligned} x_{med} &= x_{\frac{n+1}{2}} && (n \text{ odd}) \\ x_{med} &= \tfrac{1}{2}\left(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}\right) && (n \text{ even}) \end{aligned}$$

The parameter `Sorted` indicates if array `X` has been sorted before calling function `Median`. If not, it will be sorted within the function (the array `X` will therefore be modified).

Sorting (in ascending order) is performed by calling a procedure `QSort(X, Lb, Ub)`, defined in unit `uqsort`, which implements the 'Quick Sort' algorithm. Of course, this procedure may be called outside function `Median`. There is also a `DQSort` for sorting in descending order.

16.1.3 Correlation coefficient

Function `Correl(X, Y, Lb, Ub)`, defined in unit `ucorrel`, returns the correlation coefficient between `X` and `Y`:

$$r = \frac{\sum_{i=1}^{n}(x_i - m_x)(y_i - m_y)}{\sqrt{\sum_{i=1}^{n}(x_i - m_x)^2 \sum_{i=1}^{n}(y_i - m_y)^2}}$$

where m_x and m_y denote the means of the samples.

16.1.4 Skewness and kurtosis

The following functions are defined in unit `uskew` :

- Function `Skewness(X, Lb, Ub, M, Sigma)` returns the skewness of `X`, with mean `M` and standard deviation `Sigma`. This parameter is defined by:

  $$\gamma_1 = \frac{1}{n\sigma^3}\sum_{i=1}^{n}(x_i - m)^3$$

  Skewness is an indicator of the symmetric nature of the distribution. It is zero for a symmetric distribution (e. g. Gaussian), and positive (resp. negative) for an assymetric distribution with a tail extending towards positive (resp. negative) x values.

- Function `Kurtosis(X, Lb, Ub, M, Sigma)` returns the kurtosis of `X`, with mean `M` and standard deviation `Sigma`. This parameter is defined by:

  $$\gamma_2 = \frac{1}{n\sigma^4}\sum_{i=1}^{n}(x_i - m)^4 - 3$$

  Kurtosis is an indicator of the flatness of the distribution. It is zero for a Gaussian distribution, and positive (resp. negative) if the distribution is more (resp. less) 'sharp' than the Gaussian.

16.2 Comparison of means

16.2.1 Student's test for independent samples

We have 2 independent samples with sizes n_1, n_2, means m_1, m_2, standard deviations s_1, s_2. It is assumed that the samples are taken from gaussian populations with means μ_1, μ_2 and equal variances. The sample means are compared by computing the t-statistic:

$$t = \frac{m_1 - m_2}{s\sqrt{1/n_1 + 1/n_2}}$$

where s^2 is the estimation of the common variance:

$$s^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}$$

If $n_1 \geq 30$ and $n_2 \geq 30$, the conditions of normality and equal variances are no longer required and the formula becomes:

$$t = \frac{m_1 - m_2}{\sqrt{s_1^2/n_1 + s_2^2/n_2}}$$

The null hypothesis is $(H_0) : \mu_1 = \mu_2$

The alternative hypothesis (H_1) depends on the test:

One-tailed test	$(H_1) : \mu_1 > \mu_2$	$\Rightarrow$ reject (H_0) if $t > t_{1-\alpha}$
	$(H_1) : \mu_1 < \mu_2$	$\Rightarrow$ reject (H_0) if $t < t_{1-\alpha}$
Two-tailed test	$(H_1) : \mu_1 \neq \mu_2$	$\Rightarrow$ reject (H_0) if $\|t\| > t_{1-\alpha/2}$

where $t_{1-\alpha}$ is the value of the Student variable such that the cumulative probability function $\Phi_\nu(t) = 1 - \alpha$ at $\nu = n_1 + n_2 - 2$ d.o.f. (cf. chap. 5).

If H_0 is rejected, the difference of the means is considered significant at risk α

This test is implemented in the following procedure (defined in unit `ustudind`):

```
StudIndep(N1, N2, M1, M2, S1, S2, T, DoF)
```

where (`N1, N2`) are the sizes of the samples, (`M1, M2`) their means and (`S1, S2`) the estimated standard deviations (computed with `StDev`). The procedure returns Student's t in `T` and the number of degrees of freedom in `DoF`.

16.2.2 Student's test for paired samples

If the samples are paired (e. g. the same patients before and after a treatment), the t-statistic becomes:

$$t = \frac{m_d}{s_d}\sqrt{n}$$

where m_d and s_d are, respectively, the mean and standard deviations of the differences $(x_{1i} - x_{2i})$ between the paired values in the two samples, and n is the common size of the samples.

Apart from this, the test is carried out as with the independent case, with $(n - 1)$ d. o. f.

This test is implemented in the following procedure (defined in unit `ustdpair`):

```
StudPaired(X, Y, Lb, Ub, T, DoF)
```

where `X[Lb..Ub]`, `Y[Lb..Ub]` are the two samples. The procedure returns Student's t in `T` and the number of degrees of freedom in `DoF`.

After a call to this procedure, function `MathErr` returns one of the following error codes:

- `FOk` (0) if no error occurred
- `FSing` (-2) if $s_d = 0$
- `MatErrDim` (-3) if `X` and `Y` have different sizes.

16.2.3 One-way analysis of variance (ANOVA)

We have k independent samples with sizes n_i, means m_i, standard deviations s_i. It is assumed that the samples are taken from gaussian populations with means μ_i and equal variances. The goal is to compare the k means.

The following equation holds:

$$SS_t = SS_f + SS_r \tag{16.1}$$

with:

$$SS_t = \sum_{i=1}^{k}\sum_{j=1}^{n_i}(x_{ij} - \bar{x})^2 \qquad SS_f = \sum_{i=1}^{k} n_i(m_i - \bar{x})^2 \qquad SS_r = \sum_{i=1}^{k}(n_i - 1)s_i^2$$

- $\bar{x}$ is the global mean:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{k} n_i m_i \qquad n = \sum_{i=1}^{k} n_i$$

- SS_t is the *total sum of squares*; it has $(n-1)$ degrees of freedom
- SS_f is the *factorial sum of squares*; it has $(k-1)$ degrees of freedom.
- SS_r is the *residual sum of squares*; it has $(n-k)$ degrees of freedom

Note that the degrees of freedom (d.o.f.) are additive, just like the sums of squares:

$$(n-1) = (k-1) + (n-k)$$

The *variances* are defined by dividing each sum of squares by the corresponding number of d.o.f.

$$V_t = \frac{SS_t}{n-1} \qquad V_f = \frac{SS_f}{k-1} \qquad V_r = \frac{SS_r}{n-k}$$

These are the total, factorial, and residual variances, respectively. Note that the variances, unlike the sum of squares, are *not* additive!

The comparison of means is performed by computing the F-statistic:

$$F = \frac{V_f}{V_r}$$

The null hypothesis is $(H_0) : \mu_1 = \mu_2 = \cdots = \mu_k$

(H_0) is rejected if $F > F_{1-\alpha}$ where $F_{1-\alpha}$ is the value of the Fisher-Snedecor variable such that the cumulative probability function $\Phi_{\nu_1,\nu_2}(F) = 1 - \alpha$ at $\nu_1 = k - 1$ and $\nu_2 = n - k$ d.o.f. (cf. chap. 5).

This algorithm is implemented in the following procedure (defined in unit `uanova1`):

```
AnOVa1(Ns, N, M, S, V_f, V_r, F, DoF_f, DoF_r)
```

where `Ns` is the number of samples, `N[1..Ns]` are the sizes of the samples, `M[1..Ns]` their means and `S[1..Ns]` the estimated standard deviations (computed with `StDev`).
The procedure returns the factorial and residual variances in `V_f` and `V_r`, their ratio in `F` and their numbers of d. o. f. in `DoF_f` and `DoF_r`.

After a call to this procedure, function `MathErr` returns one of the following error codes:

- `FOk` (0) if no error occurred
- `FSing` (-2) if $n - k \leq 0$
- `MatErrDim` (-3) if the arrays have non-compatible dimensions

16.2.4 Two-way analysis of variance

We assume here that the means of the samples depend on two factors A and B, such that the sample corresponding to the i-th level of A and the j-th level of B has mean m_{ij} and standard deviation s_{ij}.

It is also assumed that all samples are taken from gaussian populations with equal variances, and that they have the same size n.

The previous equations become:

$$\bar{x} = \frac{1}{npq}\sum_{i=1}^{p}\sum_{j=1}^{q} n m_{ij}$$

$$SS_t = \sum_{i=1}^{p}\sum_{j=1}^{q}(x_{ij}-\bar{x})^2 \qquad SS_f = \sum_{i=1}^{p}\sum_{j=1}^{q} n(m_{ij}-\bar{x})^2 \qquad SS_r = \sum_{i=1}^{p}\sum_{j=1}^{q}(n-1)s_{ij}^2$$

with $npq-1$, $pq-1$, and $(n-1)pq$ d.o.f., respectively.

In addition, the factorial sum of squares can be splitted into three terms:

$$SS_A = qn\sum_{i=1}^{p}(m_{i.}-\bar{x})^2 \quad ; \quad (p-1) \text{ d.o.f.}$$

$$SS_B = pn\sum_{j=1}^{q}(m_{.j}-\bar{x})^2 \quad ; \quad (q-1) \text{ d.o.f.}$$

$$SS_{AB} = n\sum_{i=1}^{p}\sum_{j=1}^{q}(m_{ij}-m_{i.}-m_{.j}+\bar{x})^2 \quad ; \quad (p-1)(q-1) \text{ d.o.f.}$$

where $m_{i.}$ and $m_{.j}$ are the conditional means:

$$m_{i.} = \frac{1}{q}\sum_{j=1}^{q} m_{ij} \qquad m_{.j} = \frac{1}{p}\sum_{i=1}^{p} m_{ij}$$

that is, the means of the lines and columns of matrix $[m_{ij}]$

These sums of squares represent, respectively, the influence of factor A, the influence of factor B, and the interaction of the two factors (that is, the fact that the influence of one factor depends on the level of the other factor).

The variances are computed as before:

$$V_A = \frac{SS_A}{p-1} \qquad V_B = \frac{SS_B}{q-1} \qquad V_{AB} = \frac{SS_{AB}}{(p-1)(q-1)} \qquad V_r = \frac{SS_r}{(n-1)pq}$$

There are three null hypotheses:

$(H_0)_A$	:	The populations means do not depend on factor A
$(H_0)_B$	:	The populations means do not depend on factor B
$(H_0)_{AB}$	:	There is no interaction between the two factors

Each hypothesis is tested by computing the corresponding F-statistic (for instance, $F_A = V_A/V_r$ for testing $(H_0)_A$) and comparing with the critical value $F_{1-\alpha}$

Special case: n = 1. If there is only one observation per sample, the residual variance is zero. The null hypotheses $(H_0)_A$ and $(H_0)_B$ are tested with $F_A = V_A/V_{AB}$ and $F_B = V_B/V_{AB}$. The interaction of the factors cannot be tested.

This algorithm is implemented in the following procedure (defined in unit `uanova2`):

```
AnOVa2(NA, NB, Nobs, M, S, V, F, DoF)
```

where `NA` and `NB` are the number of levels of the factors A and B, `Nobs` the common number of observations, `N` the common size of the samples, `M[1..NA, 1..NB]` the matrix of means and `S[1..NA, 1..NB]` the matrix of standard deviations, such that the rows correspond to factor A and the columns to factor B.

The procedure returns the variances in vector `V[1..4]` $= [V_A, V_B, V_{AB}, V_r]$, the variance ratios in `F[1..3]` $= [F_A, F_B, F_{AB}]$, and the degrees of freedom in `DoF[1..4]`. If `N` $= 1$, the last element of each vector disappears.

After a call to this procedure, function `MathErr` returns one of the following error codes:

- `FOk` (0) if no error occurred
- `MatErrDim` (-3) if the arrays have non-compatible dimensions.

16.3 Comparison of variances

16.3.1 Comparison of two variances

We have 2 independent samples with sizes n_1, n_2, standard deviations s_1, s_2. It is assumed that the samples are taken from gaussian populations with variances σ_1^2, σ_2^2.

Snedecor's test uses the following statistic:

$$F = \frac{\max(s_1^2, s_2^2)}{\min(s_1^2, s_2^2)}$$

which is compared with the critical value $F_{1-\alpha/2}$ (two-tailed test).

This test is implemented in the following procedure (defined in unit `usnedeco`):

```
Snedecor(N1, N2, S1, S2, F, DoF1, DoF2)
```

where (`N1, N2`) are the sizes of the samples and (`S1, S2`) the estimated standard deviations. The procedure returns the variance ratio in `F` and the numbers of d. o. f. in `DoF1` and `DoF2`.

16.3.2 Comparison of several variances

We have k independent samples with sizes n_i, standard deviations s_i. It is assumed that the samples are taken from gaussian populations with variances σ_i^2. The goal is to compare the k variances.

Bartlett's test uses the following statistic:

$$B = \frac{1}{\lambda}\left[(n-k)\ln V_r - \sum_{i=1}^{k}(n_i - 1)\ln s_i^2\right]$$

$$\lambda = 1 + \frac{1}{3(k-1)}\left[\sum_{i=1}^{k}\frac{1}{n_i - 1} - \frac{1}{n-k}\right]$$

where $n = \sum n_i$ and V_r is the residual variance, as defined previously (§ 16.2.3).

The null hypothesis is:

$$(H_0) : \sigma_1^2 = \sigma_2^2 = \cdots = \sigma_k^2$$

Under (H_0), B follows approximately the χ^2 distribution with $(k-1)$ d. o. f. The hypothesis is tested by comparing B with the value $\chi^2_{1-\alpha}$ such that the cumulative probability function $\Phi_\nu(\chi^2) = 1 - \alpha$ at $\nu = k - 1$ d.o.f. (cf. chap. 5).

This test is implemented in the following procedure (defined in unit `ubartlet`):

```
Bartlett(Ns, N, S, Khi2, DoF)
```

where `Ns` is the number of samples, `N[1..Ns]` are the sizes of the samples and `S[1..Ns]` the estimated standard deviations. The procedure returns Bartlett's statistic in `Khi2` and the number of d. o. f. in `DoF`. The error codes are the same than for `AnOVa1`

16.4 Non-parametric tests

Non-parametric tests are used when the assumptions needed by the classical tests (gaussian populations with equal variances) are not fulfilled. They are also called *rank tests* because they work with the ranks of the values, rather than the values themselves.

The relevant procedures are all defined in unit `unonpar`.

16.4.1 Mann-Whitney test

This test compares the means of two independent samples. It is the non-parametric analog of Student's test for independent samples.

The test uses the following statistic:

$$U = \min(u_1, u_2)$$

with:

$$u_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - r_1 \quad ; \quad u_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - r_2$$

where (n_1, n_2) are the sample sizes, (r_1, r_2) the sums of the ranks of the two samples.

If $n_1 \geq 20$ and $n_2 \geq 20$, the variable:

$$\epsilon = \frac{U - \mu}{\sigma}$$

with:

$$\mu = \frac{n_1 n_2}{2} \quad ; \quad \sigma = \sqrt{\frac{n_1 n_2 (n_1 + n_2 + 1)}{12}}$$

follows the standard normal distribution under (H_0).

This test is implemented in the following procedure :

```
Mann_Whitney(N1, N2, X1, X2, U, Eps)
```

where `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`.

16.4.2 Wilcoxon test

This test compares the means of two paired samples. It is the non-parametric analog of Student's test for paired samples.

The test uses the following statistic:

$$T = \min(T_+, T_-)$$

where T_+ and T_- are the sums of the ranks of the positive and negative differences between the values of the two samples.

If the sample size is $N > 25$, the variable:

$$\epsilon = \frac{T - \mu}{\sigma}$$

with:

$$\mu = \frac{N(N+1)}{4} \qquad ; \qquad \sigma = \sqrt{\frac{N(N+1)(2N+1)}{24}}$$

follows the standard normal distribution under (H_0).

This test is implemented in the following procedure :

```
Wilcoxon(X, Y, Lb, Ub, Ndiff, T, Eps)
```

where `X[Lb..Ub]` and `Y[Lb..Ub]` are the two samples. The procedure returns the number of non-zero differences in `Ndiff`, Wilcoxon's statistic in `T` and the associated normal variable in `Eps`.

16.4.3 Kruskal-Wallis test

This test compares the means of several independent samples. It is the non-parametric analog of one-way ANOVA.

The test uses the following statistic:

$$H = \frac{12}{n(n+1)} \sum_{i=1}^{k} \frac{r_i^2}{n_i} - 3(n+1)$$

where k is the number of samples, n_i the size of sample i, r_i the sum of the ranks for sample i and n the total size.

If $n_i > 5 \ \forall i$, H follows the χ^2 distribution with $k-1$ d.o.f.

This test is implemented in the following procedure :

```
Kruskal_Wallis(Ns, N, X, H, DoF)
```

where `Ns` is the number of samples, `N[1..Ns]` is the vector of sizes and `X` the sample matrix (with the samples as columns). The procedure returns the Kruskal-Wallis statistic in `H` and the number of d. o. f. in `DoF`.

16.5 Statistical distribution

A statistical distribution is generated by binning data into a set of statistical classes $]x_i, x_{i+1}]$. Each class is characterized by the following parameters:

- its bounds x_i, x_{i+1}
- the number of values n_i contained in the class
- the frequency $f_i = n_i/N$ where N is the total number of values
- the density $d_i = f_i/(x_{i+1} - x_i)$

This structure is implemented in DMath as:

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

A distribution is generated with the following procedure (defined in unit `udistrib`):

```
Distrib(X, Lb, Ub, A, B, H, C)
```

where `X[Lb..Ub]` is the original set of values, `A` and `B` the lower and upper bounds of the distribution and `H` the common width of the classes, according to the following scheme:

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

The distribution is returned in `C` which is an array of statistical classes, allocated by `DimStatClassVector(C, M)`.

16.6 Comparison of distributions

16.6.1 Observed and theoretical distributions

An observed distribution may be compared to a theoretical one by using the following statistics:

- Pearson's χ^2 :

$$\chi^2 = \sum_{i=1}^{p} \frac{(O_i - C_i)^2}{C_i}$$

- Woolf's G :

$$G = 2 \sum_{i=1}^{p} O_i \ln \frac{O_i}{C_i}$$

where O_i and C_i denote the observed and theoretical numbers of values in class i, and p the number of classes.

The null hypothesis is (H_0): the observed distribution conforms to the theoretical one (it is a test for conformity)

Under (H_0), both statistics follow the χ^2 distribution with $(p - 1 - N_e)$ d. o. f., where N_e is the number of parameters which have been estimated to compute the C_i values (e. g. $N_e = 2$ if the mean and standard deviation of the distribution have been estimated).

(H_0) is rejected if the chosen statistic is higher than the critical value $\chi^2_{1-\alpha}$ for the chosen risk α.

Pearson's statistic is an approximation of Woolf's statistic. It is usually recommended to use it only if $C_i \geq 5 \ \forall i$.

The following procedures are defined in units `ukhi2` and `uwoolf`, respectively:

```
Khi2_Conform(N_cls, N_estim, Obs, Calc, Khi2, DoF)

Woolf_Conform(N_cls, N_estim, Obs, Calc, G, DoF)
```

where `N_cls` denotes 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` or `G` and the number of d. o. f. in `DoF`.

16.6.2 Several observed distributions

To compare several observed distributions, we can group them into a *contingency table* **O** such that O_{ij} denotes the number of values for class i in the j-th distribution.

The Pearson and Woolf statistics may then be computed as:

$$\chi^2 = \sum_{i=1}^{p} \sum_{j=1}^{q} \frac{(O_{ij} - C_{ij})^2}{C_{ij}}$$

$$G = 2 \sum_{i=1}^{p} \sum_{j=1}^{q} O_{ij} \ln \frac{O_{ij}}{C_{ij}}$$

where p the number of classes, q the number of distributions, and C_{ij} the theoretical value of O_{ij}, computed as:

$$C_{ij} = \frac{N_{i.} N_{.j}}{N}$$

where $N_{i.}$ is the sum of terms in line i, $N_{.j}$ is the sum of terms in column j, and N the global sum of all terms in the matrix ($N = \sum_i N_{i.} = \sum_j N_{.j}$).

The null hypothesis is (H_0): the observed distributions come from the same population (it is a test for homogeneity or independence).

Under (H_0), both statistics follow the χ^2 distribution with $(p-1)(q-1)$ d. o. f.

The following procedures are defined in units `ukhi2` and `uwoolf`, respectively:

```
Khi2_Indep(N_lin, N_col, Obs, Khi2, DoF)

Woolf_Indep(N_lin, N_col, Obs, G, DoF)
```

where `N_lin` and `N_col` are the numbers of lines and columns (i. e. p and q), and `Obs[1..N_lin, 1..N_col]` is the matrix of observed distributions. The statistic is returned in `Khi2` or `G` and the number of d. o. f. in `DoF`.

16.7 Demo programs

16.7.1 Console programs

These programs are located in the `demo\console\stat` subdirectory.

Descriptive statistics, comparison of means and variances

Program `stat.pas` performs a statistical analysis of hemoglobin concentrations in two samples of 30 men and 30 women. The computed parameters are the mean, standard deviation, skewness and kurtosis. The means are compared by Student's test (two-tailed) and Mann-Whitney's test, and the variances are compared by Snedecor's test.

Student's test for paired samples

Program `student.pas` compares the means of two paired samples, using Student's and Wilcoxon's two-tailed tests.

One-way analysis of variance

Program `av1.pas` compares the means of 5 independent samples, each with 12 observations, using one-way ANOVA and the Kruskal-Wallis test. In addition, the variances of the samples are compared with Bartlett's test.

Two-way analysis of variance

- Program `av2.pas` compares the means of 4 samples, depending on two factors, using two-way ANOVA. Each sample contains 12 observations.
- Program `av2a.pas` performs two-way ANOVA with one observation per sample.

Comparison of distributions

Program `khi2.pas` performs both χ^2 and Woolf's tests, first to compare an observed distribution with a theoretical one, and then to analyse a contingency table.

16.7.2 BGI program

Program `histo.pas` (located in the `demo\bgi\stat` subdirectory) uses the hemoglobin data from program `stat.bas` to generate a statistical distribution.

The first step determines a suitable range for the data. This is done by calling procedure `Interval` (defined in unit `uinterv`):

```
    Interval(X[1], X[N], 5, 10, XMin, XMax, XStep);
```

The arguments 5 and 10 represent the minimal and maximal number of classes which is desired.

The second step generates the distribution, using the ranges determined in the previous step:

```
Ncls := Round((Xmax - Xmin) / XStep);
DimStatClassVector(C, Ncls);
Distrib(X, 1, N, Xmin, Xmax, XStep, C);
```

This distribution is then compared with the normal distribution, using both χ^2 and Woolf's tests. The theoretical C_i values are computed from the cumulative probability function for the normal distribution having the same mean and standard deviation than the observed distribution.

The program plots an histogram of the observed distribution, together with the curve corresponding to the normal distribution. This curve is generated from the probability density function:

```
function PltFunc(X : Float) : Float;
begin
  PltFunc := DNorm((X - M) / S) / S;
end;
```

where `M, S` are the mean and standard deviation of the observed distribution, and `DNorm` is the probability density of the standard normal distribution (see chapter 5). Note that the histogram is constructed with the class densities as ordinates, so that a comparison with the plotted curve can be made.

Chapter 17

Linear regression

This chapter describes the routines available in DMath for fitting a straight line by linear regression. Other types of curve fitting will be described in subsequent chapters.

17.1 Straight line fit

The problem is to determine the equation of the line which comes closest to a set of points.

The model is defined by the equation:

$$y = a + bx$$

- x is the independent (or 'explicative') variable
- y is the dependent (or 'explained') variable
- a and b are the model parameters

Assume that the n points $(x_1, y_1), (x_2, y_2), \cdots (x_n, y_n)$ are perfectly lined, so that each of them verifies the equation of the straight line:

$$y_1 = a + bx_1$$

$$y_2 = a + bx_2$$

$$\cdots\cdots\cdots$$

$$y_n = a + bx_n$$

Or in matrix form:

$$\mathbf{y} = \mathbf{X}\beta \qquad \Longleftrightarrow \qquad \mathbf{y} - \mathbf{X}\beta = \mathbf{0}$$

where:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_n \end{bmatrix} \qquad \mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdots & \cdots \\ 1 & x_n \end{bmatrix} \qquad \beta = \begin{bmatrix} a \\ b \end{bmatrix}$$

In the general case, the points are not exactly lined, so that:

$$\mathbf{y} - \mathbf{X}\beta = \mathbf{r}$$

where $\mathbf{r}$ is the vector of residuals:

$$\mathbf{r} = [r_1, r_2 \cdots r_n]^\top = \mathbf{y} - \hat{\mathbf{y}}$$

where $\hat{\mathbf{y}} = \mathbf{X}\beta$

It is possible to compute β so that $\| \mathbf{r} \|$ is minimal (*least squares criterion*).

$$\| \mathbf{r} \|^2 = \mathbf{r}^\top\mathbf{r} = r_1^2 + r_2^2 + \cdots + r_n^2 = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = SS_r$$

where $\hat{y}_i = a + bx_i$ and SS_r is the *sum of squared residuals.*

Several methods allow the determination of β under the least squares criterion. The QR and SVD algorithms have been described previously. Here we will study the *method of normal equations.*

It may be shown that β is the solution of the system:

$$\mathbf{A}\beta = \mathbf{c}$$

with:

$$\mathbf{A} = \mathbf{X}^\top\mathbf{X} \qquad \mathbf{c} = \mathbf{X}^\top\mathbf{y}$$

so:

$$\beta = \mathbf{A}^{-\mathbf{1}}\mathbf{c} = (\mathbf{X}^\top\mathbf{X})^{-\mathbf{1}}(\mathbf{X}^\top\mathbf{y})$$

The matrices may be expressed in terms of statistical sums:

$$\mathbf{A} = \begin{bmatrix} n & \Sigma x_i \\ \Sigma x_i & \Sigma x_i^2 \end{bmatrix} \qquad \mathbf{c} = \begin{bmatrix} \Sigma y_i \\ \Sigma x_i y_i \end{bmatrix}$$

$$\mathbf{A}^{-\mathbf{1}} = \frac{1}{n\Sigma x_i^2 - (\Sigma x_i)^2} \begin{bmatrix} \Sigma x_i^2 & -\Sigma x_i \\ -\Sigma x_i & n \end{bmatrix}$$

$$\beta = \frac{1}{n\Sigma x_i^2 - (\Sigma x_i)^2} \begin{bmatrix} \Sigma x_i^2 \Sigma y_i - \Sigma x_i \Sigma x_i y_i \\ -\Sigma x_i \Sigma y_i + n\Sigma x_i y_i \end{bmatrix}$$

17.2 Analysis of variance

The following equation holds:

$$SS_t = SS_e + SS_r \tag{17.1}$$

with:

$$SS_t = \sum_{i=1}^{n}(y_i - \bar{y})^2 \qquad SS_e = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 \qquad SS_r = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- $\bar{y}$ is the mean of the y values:

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

- SS_t is the *total sum of squares*; it has $(n-1)$ degrees of freedom
- SS_e is the *explained sum of squares*; it has 1 degree of freedom.
- SS_r is the *residual sum of squares*; it has $(n-2)$ degrees of freedom

Note that the degrees of freedom (d.o.f.) are additive, just like the sums of squares:

$$(n-1) = 1 + (n-2)$$

The *variances* are defined by dividing each sum of squares by the corresponding number of d.o.f.

$$V_t = \frac{SS_t}{n-1} \qquad V_e = SS_e \qquad V_r = \frac{SS_r}{n-2}$$

These are the total, explained, and residual variances, respectively. Note that the variances, unlike the sum of squares, are *not* additive!

The following quantities are derived from the above equations:

- the **coefficient of determination** r^2

$$r^2 = \frac{SS_e}{SS_t}$$

  r^2 represents the percentage of the variations of y which are 'explained' by the independent variable. It is always comprised between 0 and 1. A value of 1 indicates a perfect fit.

- the **correlation coefficient** r

  It is the square root of the coefficient of determination, with the sign of the slope b. It is therefore comprised between -1 and 1.

- the **residual standard deviation** s_r

  It is the square root of the residual variance ($s_r = \sqrt{V_r}$). It is an estimate of the error made on the measurement of the dependent variable y. It should be 0 for a perfect fit.

- the **variance ratio** F

  It is the ratio of the explained variance to the residual variance ($F = V_e/V_r$). It should be infinite for a perfect fit.

17.3 Precision of parameters

The matrix:

$$\mathbf{V} = V_r \cdot \mathbf{A}^{-1} = V_r \cdot (\mathbf{X}^\top \mathbf{X})^{-1}$$

is called the **variance-covariance matrix** of the parameters. It is a symmetric matrix with the following structure:

$$\mathbf{V} = \begin{bmatrix} \mathrm{Var}(a) & \mathrm{Cov}(a,b) \\ \mathrm{Cov}(a,b) & \mathrm{Var}(b) \end{bmatrix}$$

The diagonal terms are the variances of the parameters, from which the standard deviations are computed by:

$$s_a = \sqrt{\mathrm{Var}(a)} \qquad s_b = \sqrt{\mathrm{Var}(b)}$$

The off-diagonal term is the covariance of the two parameters, from which the correlation coefficient r_{ab} is computed by:

$$r_{ab} = \frac{\mathrm{Cov}(a,b)}{s_a s_b}$$

17.4 Probabilistic interpretation

It is assumed that the residuals ($y_i - \hat{y}_i$) are identically and independently distributed according to a normal distribution with mean 0 and standard deviation σ (estimated by s_r).

It may be shown that the regression parameters (a, b) are distributed according to a Student distribution with $(n-2)$ d.o.f.

It is therefore possible to compute a confidence interval for each parameter, for instance:

$$\left[a - t_{1-\alpha/2} \cdot s_a \quad , \quad a + t_{1-\alpha/2} \cdot s_a\right]$$

where $t_{1-\alpha/2}$ is the value of the Student variable corresponding to the chosen probability α (usually $\alpha = 0.05$). This interval has a probability $(1-\alpha)$ to contain the 'true' value of the parameter.

It is also possible to compute a 'critical' value $F_{1-\alpha}$ from the Fisher-Snedecor distribution with 1 and $(n-2)$ d.o.f . The fit is considered satisfactory if the variance ratio F exceeds 4 times the critical value.

Note : for the straight line fit, $F_{1-\alpha} = \left(t_{1-\alpha/2}\right)^2$

17.5 Weighted regression

It is assumed here that the variance $v_i = \sigma_i^2$ of the measured value y_i is not constant.

The sums of squares become:

$$SS_t = \sum_{i=1}^{n} w_i (y_i - \bar{y})^2 \qquad SS_e = \sum_{i=1}^{n} w_i (\hat{y}_i - \bar{y})^2 \qquad SS_r = \sum_{i=1}^{n} w_i (y_i - \hat{y}_i)^2$$

where w_i denotes the 'weight', equal to $1/v_i$, and $\bar{y}$ denotes the weighted mean:

$$\bar{y} = \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i}$$

The regression parameters $\mathbf{b}$ are estimated by:

$$\mathbf{b} = (\mathbf{X}^\top \mathbf{W} \mathbf{X})^{-\mathbf{1}} (\mathbf{X}^\top \mathbf{W} \mathbf{y})$$

where $\mathbf{W}$ is the diagonal matrix of weights:

$$\mathbf{W} = diag(w_1, w_2, \cdots, w_n) = \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & w_n \end{bmatrix}$$

The values of r_2, s_r and F, as well as the variance-covariance matrix, are computed as above (§ 17.2). The normalized residual for the i-th observation is:

$$\frac{y_i - \hat{y}_i}{\sigma_i} = (y_i - \hat{y}_i)\sqrt{w_i}$$

These normalized residuals should follow the standard normal distribution.

17.6 Programming

17.6.1 Regression procedures

The following subroutines are defined in unit `ulinfit`:

- `LinFit(X, Y, Lb, Ub, B, V)` for unweighted linear regression
- `WLinFit(X, Y, S, Lb, Ub, B, V)` for weighted linear regression
- `SVDLinFit(X, Y, Lb, Ub, SVDTol, B, V)`

  Same than `LinFit` but uses singular value decomposition instead of normal equations. `SVDTol` is the threshold under which a singular value is considered zero. It is expressed as a fraction of the highest singular value (see paragraph 9.8 for details).
- `WSVDLinFit(X, Y, S, Lb, Ub, SVDTol, B, V)`

  Same than `WLinFit` but uses singular value decomposition.

The input parameters are:

- `X[Lb..Ub], Y[Lb..Ub]` : coordinates of points
- `S[Lb..Ub]` : standard deviations of Y values (noted σ_i in paragraph 17.5)

The output parameters are:

- `B[0..1]` : regression parameters
- `V[0..1, 0..1]` : inverse of the matrix of normal equations (noted $\mathbf{A}^{-1}$ in paragraph 17.3). This is **not** the variance-covariance matrix. This one will be computed by the routines described in the next paragraph.

After a call to one of these procedures, function `MathErr` returns one of the following error codes:

- `MatOk` if no error occurred
- `MatSing` if the matrix of normal equations is quasi-singular

17.6.2 Quality of fit

The parameters used to test the quality of the fit are grouped in a user-defined type:

```
type
  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;
```

They are computed by the following subroutines (defined in unit `uregtest`):

- `RegTest(Y, Ycalc, LbY, UbY, V, LbV, UbV, Test)` for unweighted regression
- `WRegTest(Y, Ycalc, S, LbY, UbY, V, LbV, UbV, Test)` for weighted regression

The input parameters are:

- `Y[LbY..UbY]` : ordinates of points
- `Ycalc[LbY..UbY]` : Y values computed from the regression equation, using the fitted parameters `B`. This computation must be done before calling `RegTest` or `WRegTest`.
- `V[LbV..UbV, LbV..UbV]` : the inverse matrix of the normal equations, as returned by the regression procedures.

The output parameters are:

- `V` : the variance-covariance matrix of the fitted parameters
- `Test` : variable of type `TRegTest`, as defined above.

17.7 Demo programs

17.7.1 BGI programs

These programs are located in `demo\bgi\curfit`.

Unweighted linear regression

Program `reglin.pas` performs the least squares fit of a straight line, according to the following equation:

```
Y = B[0] + B[1] * X
```

The parameter vector and the variance-covariance matrix are therefore declared as:

```
DimVector(B, 1);
DimMatrix(V, 1, 1);
```

The program calls procedure `LinFit`, then computes the theoretical Y values:

```
for I := 1 to N do
  Ycalc[I] := B[0] + B[1] * XX[I];
```

Note that this computation must be done before calling procedure `RegTest`

The critical values of Student's t and Snedecor's F are computed for the chosen probability `Alpha` by using the functions from chapter 5.

```
Tc := InvStudent(N - 2, 1 - 0.5 * Alpha);
Fc := InvSnedecor(1, N - 2, 1 - Alpha);
```

The ouput shows the standardized residuals, equal to $(y_i - \hat{y}_i)/\sigma$, where σ is estimated by s_r. They should be distributed according to the standard normal distribution.

Weighted linear regression

Program `wreglin.pas` performs the weighted least squares fit of a straight line. Here the standard deviations σ_i of the observed y values are stored in a vector `S` defined by the user.

The computations involve the same steps as with the previous program, except that procedures `WLinFit` and `WRegTest` are used, and that the standardized residuals are computed as $(y_i - \hat{y}_i)/\sigma_i$

The plot shows the error bars, corresponding to $y_i \pm \sigma_i$ for each point.

17.7.2 GUI program

Program `curfit.dpr`, located in `demo\gui\curfit`, performs linear and nonlinear regression.

The options are selected by means of a menu. The main options relevant to linear regression are the following:

- *File / Open* : reads data from a text file.

  The file must have the following structure:

  - line 1 : title of study
  - line 2 : number of variables (should be 2 here)
  - line 3 - 4 : names of variables (one name by line)
  - line 5 : number of observations
  - following lines : tabulated data (one variable by column, one observation by line)

  The file `reglin.dat` is an example data file.

- *Compute / Select Model* : selects the regression model.

  Here we need only the linear regression model, which is the default.

- *Compute / Select Algorithm* : selects the regression algorithm.

  Both the Gauss-Jordan method and the Singular Value Decomposition are available for linear regression. SVD is the default method. It is possible to set the tolerance under which the singular values are considered null (see paragraph 9.8).

- *Compute / Fit Model* : fits the model and writes the results in an output file.

  The output file has the same name than the data file but with the extension 'out'.

- *Compute / View Results* : displays the contents of the output file.

- *Graph / Options* : sets all graph options, as described in chapter 7

- *Graph / Axes and Curves* : selects the type of plot.

  This option can be used for instance to get a plot of residuals. It is also possible to display error bars as a given multiple of the residual standard deviation.

Chapter 18

Multilinear regression and principal component analysis

This chapter describes the procedures available in DMath for multilinear regression, polynomial regression and principal component analysis.

18.1 Multilinear regression

18.1.1 Normal equations

The regression model is:

$$y = a + bx_1 + cx_2 + \cdots$$

where the x_i are m independent variables.

The method of normal equations, studied in chapter 17, is still applicable with:

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

There are $p = m + 1$ parameters. The number of observations n must be such that $n > p$.

Special case: The x_i may be functions of another variable x, as long as these functions do not contain parameters. For instance:

- Polynomial: $y = a + bx + cx^2 + \cdots$

- Fourier series: $y = a + b \sin x + c \sin 2x + \cdots$

In such cases, the matrix $\mathbf{X}$, the matrix of normal equations $\mathbf{A} = \mathbf{X}^\top \mathbf{X}$ and the constant vector $\mathbf{c} = \mathbf{X}^\top \mathbf{y}$ will have special forms. For instance with polynomial regression, if d is the degree of the polynomial:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_n & x_n^2 & \cdots & x_n^d \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} n & \Sigma x_i & \Sigma x_i^2 & \cdots & \Sigma x_i^d \\ \Sigma x_i & \Sigma x_i^2 & \Sigma x_i^3 & \cdots & \Sigma x_i^{d+1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \Sigma x_i^d & \Sigma x_i^{d+1} & \Sigma x_i^{d+2} & \cdots & \Sigma x_i^{2d} \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} \Sigma y_i \\ \Sigma x_i y_i \\ \cdots \\ \Sigma x_i^d y_i \end{bmatrix}$$

It is possible to use these special forms to simplify the computations. For instance, only the first line and the last column of the above matrix $\mathbf{A}$ need to be computed; the others terms are deduced by shifting.

18.1.2 Analysis of variance

Equation 17.1 still holds with the following modifications:

- the explained sum of squares SS_e has $(p-1)$ degrees of freedom.
- the residual sum of squares SS_r has $(n-p)$ degrees of freedom

Note that the degrees of freedom are still additive:

$$(n-1) = (p-1) + (n-p)$$

The explained and residual variances become:

$$V_e = \frac{SS_e}{p-1} \qquad V_r = \frac{SS_r}{n-p}$$

The quantities r^2, s_r, F are derived as in § 17.1, but here the correlation coefficient r is always positive.

In multilinear regression, the use of r^2 may be misleading because it is always possible to artificially increase its value by adding more independent variables or using a higher degree polynomial. To overcome this drawback, the **adjusted coefficient of determination** may be used instead:

$$r_a^2 = 1 - (1 - r^2)\frac{n-1}{n-p}$$

18.1.3 Precision of parameters

The variance-covariance matrix $\mathbf{V}$ is computed as in chapter 17. It is a $p \times p$ symmetric matrix such that:

- the diagonal term V_{ii} is the variance of the i-th parameter
- the off-diagonal term V_{ij} is the covariance of the i-th and j-th parameters

The correlation coefficient r_{ij} is computed by:

$$r_{ij} = \frac{V_{ij}}{\sqrt{V_{ii}V_{jj}}}$$

18.1.4 Probabilistic interpretation

Assuming that the residuals are identically and independently distributed according to a normal distribution, the regression parameters are distributed according to a Student distribution with $(n - p)$ d.o.f. Confidence intervals may be computed as in chapter 17.

The 'critical' value $F_{1-\alpha}$ is computed from the Fisher-Snedecor distribution with $(p-1)$ and $(n-p)$ d.o.f. However, the relationship $F_{1-\alpha} = \left(t_{1-\alpha/2}\right)^2$ does not hold if $p > 2$.

18.1.5 Weighted regression

Weighted multilinear regression may be performed as for the simple linear case (chap. 17).

18.1.6 Programming

Multilinear regression

The following subroutines are available in unit `umulfit` :

- `MulFit(X, Y, Lb, Ub, Nvar, ConsTerm, B, V)` for unweighted multilinear regression

  `X[Lb..Ub, 1..Nvar]` is the matrix of independent variables, `Y[Lb..Ub]` is the vector of dependent variable, and `ConsTerm` is a boolean parameter which indicates the presence of a constant term b_0. The regression parameters are returned in `B` and the inverse matrix in `V`.

- `WMulFit(X, Y, S, Lb, Ub, Nvar, ConsTerm, B, V)` for weighted multilinear regression

  The additional parameter `S` is a vector containing the standard deviations of the observations.

Two alternative procedures are available in unit `usvdfit` for using singular value decomposition instead of normal equations:

- `SVDFit(X, Y, Lb, Ub, Nvar, ConsTerm, SVDTol, B, V)`

  Same than `MulFit` but uses singular value decomposition instead of normal equations. `SVDTol` is the threshold under which a singular value is considered zero. It is expressed as a fraction of the highest singular value (see paragraph 9.8 for details).

- `WSVDFit(X, Y, Lb, Ub, Nvar, ConsTerm, SVDTol, B, V)`

  Same than `WMulFit` but uses singular value decomposition.

Polynomial regression

The following procedures are available in unit `upolfit` :

- `PolFit(X, Y, Lb, Ub, Deg, B, V)` for unweighted polynomial regression

  Here `X[Lb..Ub]` and `Y[Lb..Ub]` are the point coordinates and `Deg` is the degree of the polynomial.

- `WPolFit(X, Y, S, Lb, Ub, Deg, B, V)` for weighted polynomial regression

- `SVDPolFit(X, Y, Lb, Ub, Deg, SVDTol, B, V)`

  Same than `PolFit` but uses singular value decomposition.

- `WSVDPolFit(X, Y, S, Lb, Ub, Deg, SVDTol, B, V)`

  Same than `WPolFit` but uses singular value decomposition.

After a call to one of these procedures, function `MathErr` returns one of the following error codes:

- `MatOk` if no error occurred
- `MatSing` if the matrix of normal equations is quasi-singular
- `MatErrDim` if the array dimensions do not match

18.2 Principal component analysis

18.2.1 Theory

The goal of Principal Component Analysis (PCA) is to replace a set of m variables $\mathbf{x}_1, \mathbf{x}_2, \cdots \mathbf{x}_m$, which may be correlated, by another set $\mathbf{f}_1, \mathbf{f}_2, \cdots \mathbf{f}_m$, called the *principal components* or *principal factors*. These factors are independent (uncorrelated) variables.

Usually, the algorithm starts with the *correlation matrix* $\mathbf{R}$ which is a $m \times m$ symmetric matrix such that R_{ij} is the correlation coefficient between variable $\mathbf{x}_i$ and variable $\mathbf{x}_j$.

The eigenvalues $\lambda_1, \lambda_2, \cdots \lambda_m$ (in decreasing order) of matrix $\mathbf{R}$ are the variances of the principal factors. Their sum $\sum_{i=1}^{p} \lambda_i$ is equal to m. So, the percentage of variance associated with the i-th factor is equal to λ_i/m.

If $\mathbf{C}$ is the matrix of eigenvectors of $\mathbf{R}$, the correlation coefficient between variable $\mathbf{x}_i$ and factor $\mathbf{f}_j$ (sometimes called *loading*) is:

$$R_{C_{ij}} = C_{ij}\sqrt{\lambda_j}$$

The coordinates of the principal factors (sometimes called *scores*) are such that:

$$\mathbf{F} = \mathbf{ZC}$$

where $\mathbf{Z}$ denotes the matrix of scaled original variables:

$$Z_{ij} = \frac{X_{ij} - m_j}{s_j}$$

where m_j and s_j are the mean and standard deviation of the j-th variable.

Note that the reduced variables have means 0 and variances 1, while the principal factors have means 0 and variances λ_i.

In most cases, a limited number of principal factors represent the most part of the total variance. It is therefore possible to neglect the other factors and to replace the m original (partially correlated) variables by a smaller set of independent variables. These variables can then be used in a regression analysis instead of the original ones (*orthogonalized regression*).

18.2.2 Programming

The following subroutines are available in unit `upca` :

- `VecMean(X, Lb, Ub, Nvar, M)` computes the mean vector `M[1..Nvar]` from matrix `X[Lb..Ub, 1..Nvar]`.

- `VecSD(X, Lb, Ub, Nvar, M, S)` computes the standard deviations `S[1..Nvar]` from matrix `X` and mean vector `M`.

- `ScaleVar(X, Lb, Ub, Nvar, M, S, Z)` computes the scaled variables `Z[Lb..Ub, 1..Nvar]` from the original variables `X`, the means `M` and the standard deviations `S`.

- `MatVarCov(X, Lb, Ub, Nvar, M, V)` computes the variance-covariance matrix `V[1..Nvar, 1..Nvar]` from matrix `X` and mean vector `M`.

- `MatCorrel(V, Nvar, R)` computes the correlation matrix `R[1..Nvar, 1..Nvar]` from the variance-covariance matrix `V`.

- `PCA(R, Nvar, MaxIter, Tol, Lambda, C, Rc)` performs the principal component analysis of the correlation matrix `R`, which is destroyed. `MaxIter` and `Tol` are the maximum number of iterations and the requested tolerance for the Jacobi method (see paragraph 9.10.2). The eigenvalues are returned in vector `Lambda[1..Nvar]`, the eigenvectors in the columns of matrix `C[1..Nvar, 1..Nvar]`. The matrix `Rc[1..Nvar, 1..Nvar]` contains the correlation coefficients (loadings) between the original variables (rows) and the principal factors (columns).

- `PrinFac(Z, Lb, Ub, Nvar, C, F)` computes the principal factors (scores) `F[Lb..Ub, 1..Nvar]` from the scaled variables `Z` and the matrix of eigenvectors `C`.

After a call to these procedures, function `MathErr` returns one of the following error codes:

- `MatOk` if no error occurred
- `MatErrDim` if the array dimensions do not match
- `MatNonConv` if the iterative procedure (Jacobi method) did not converge in subroutine `PCA`

18.3 Demo programs

18.3.1 Console program

Program `pcatest.pas` (located in `demo\console\curfit`) performs a principal component analysis on a set of 4 variables (Example taken from: P. DAGNELIE, *Analyse statistique à plusieurs variables*, Presses Agronomiques de Gembloux, Belgique, 1982). The program prints:

- The mean vector and variance-covariance matrix of the original variables
- The correlation coefficients between the original variables
- The eigenvalues and eigenvectors of the correlation matrix
- The correlation coefficients between the principal factors and the original variables
- The values of the principal factors for each point

It may be seen that:

- High correlations exist between the original variables, which are therefore not independent
- According to the eigenvalues, the last two principal factors may be neglected since they represent less than 11 % of the total variance. So, the original variables depend mainly on the first two factors
- The first principal factor is negatively correlated with the second and fourth variables, and positively correlated with the third variable

- The second principal factor is positively correlated with the first variable
- The table of principal factors show that the highest scores are usually associated with the first two principal factors, in agreement with the previous results

18.3.2 BGI programs

These programs are located in `demo\bgi\curfit`

- Program `regmult.pas` performs a multilinear least squares fit with `Nvar` = 4 independent variables, according to the following equation:

  ```
  Y = B[0] + B[1] * X1 + B[2] * X2 + B[3] * X3 + B[4] * X4
  ```

  The data are stored in a matrix `X` and a vector `Y`.

  The parameter vector and variance-covariance matrix are declared as:

  ```
  DimVector(B, Nvar);
  DimMatrix(V, Nvar, Nvar);
  ```

  The program calls procedure `MulFit` or `SVDFit`, then computes the theoretical Y values:

  ```
  for I := 1 to N do
    begin
      if ConsTerm then Ycalc[I] := B[0] else Ycalc[I] := 0.0;
      for J := 1 to Nvar do
        Ycalc[I] := Ycalc[I] + B[J] * XX[I,J];
    end;
  ```

  Note that this computation must be done before calling procedure `RegTest`

  The critical values of Student's t and Snedecor's F are computed for the chosen probability `Alpha` by using the functions from chapter 5.

  ```
  Tc := InvStudent(Test.Nu2, 1 - 0.5 * Alpha);
  Fc := InvSnedecor(Test.Nu1, Test.Nu2, 1 - Alpha);
  ```

where `Test.Nu1` and `Test.Nu2` are the numbers of d.o.f., returned by procedure `RegTest`.

The ouput shows the standardized residuals, equal to $(y_i - \hat{y}_i)/\sigma$, where σ is estimated by s_r. They should be distributed according to the standard normal distribution.

Due to the multi-dimensional nature of the relationship, a plot of y as a function of the x's is not possible. Rather, the program plots a diagram of the observed and computed values of y, together with the theoretical line $\hat{y} = y$.

- Program `regpoly.pas` performs a polynomial least squares fit. The structure of the program is very similar to the previous one, with the degree of the polynomial (`Deg`) playing the role of the number of variables (`Nvar`).

  Here, only a vector `X` is needed to store the values of the independent variable, since the powers of x are computed by the polynomial regression routine `PolFit`.

  The theoretical Y values are computed by means of function `Poly`, studied in chapter 12.

  The program plots the fitted curve by calling the plotting subroutine `PlotFunc`. The function which is passed to this subroutine is defined as:

  ```
  function PltFunc(X : Float) : Float;
  begin
    PltFunc := Poly(X, B, Deg);
  end;
  ```

  The definition of procedure `PlotFunc` does not allow additional parameters for `PltFunc`. This is the only reason why the parameter vector `B` is declared as a global variable.

18.3.3 GUI program

The program `curfit.dpr`, which we have already described in the previous chapter, can also perform polynomial regression. This option is selected from the 'Compute / Select Model' menu. The file `regpol.dat`, located in `demo\gui\curfit`, is an example data file.

Chapter 19

Nonlinear regression

This chapter describes the procedures available in DMath for fitting models which are nonlinear with respect to their parameters. For instance, the exponential model $y = ae^{-bx}$ is nonlinear with respect to the parameter b.

19.1 Theory

The regression model is:

$$y = f(x; a, b \cdots)$$

where f is a nonlinear function of the parameters $a, b \cdots$

Assume that we have a first estimate $(a^0, b^0 \cdots)$ of the parameters. Let us write the Taylor series expansion of y in the vicinity of this estimate:

$$y = y^0 + y'_a \cdot (a - a^0) + y'_b \cdot (b - b^0) + \cdots$$

where:

$$\begin{aligned} y^0 &= f(x; a^0, b^0 \cdots) \\ y'_a &= \frac{\partial f}{\partial a}(x; a^0, b^0 \cdots) \\ y'_b &= \frac{\partial f}{\partial b}(x; a^0, b^0 \cdots) \\ &\cdots\cdots\cdots\cdots\cdots\cdots \end{aligned}$$

The equation may be rewritten as:

$$y - y^0 = y'_a \cdot (a - a^0) + y'_b \cdot (b - b^0) + \cdots$$

which corresponds to the linear regression problem:

$$\mathbf{z} = \mathbf{J} \cdot \delta$$

with:

$$\mathbf{z} = \begin{bmatrix} y_1 - y_1^0 \\ y_2 - y_2^0 \\ \cdots \\ y_n - y_n^0 \end{bmatrix} \qquad \mathbf{J} = \begin{bmatrix} y'_{a1} & y'_{b1} & \cdots \\ y'_{a2} & y'_{b2} & \cdots \\ \cdots & \cdots & \cdots \\ y'_{an} & y'_{bn} & \cdots \end{bmatrix} \qquad \delta = \begin{bmatrix} a - a^0 \\ b - b^0 \\ \cdots \end{bmatrix}$$

where $\mathbf{J}$ is the *Jacobian matrix*, such that $y'_{ai} = \partial f(x_i; a^0, b^0 \cdots)/\partial a$ etc.

Application of the linear regression relationships leads to:

$$\delta = (\mathbf{J}^\top \mathbf{J})^{-1}(\mathbf{J}^\top \mathbf{z}) \tag{19.1}$$

Knowing the correction vector δ, it is possible to compute better estimates a and b of the parameters. The process is repeated until convergence of the parameter estimates.

The method so described is known as the *Gauss-Newton* method. It is usually combined with nonlinear optimization, usually Marquardt's method, in order to minimize the sum of squared residuals:

$$SS_r = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \Phi(a, b \cdots)$$

In this case, the gradient vector $\mathbf{g}$ and hessian matrix $\mathbf{H}$ of function Φ are computed by the following relationships:

$$\mathbf{g} = -\mathbf{J}^\top \mathbf{z} \qquad \mathbf{H} = \mathbf{J}^\top \mathbf{J} \tag{19.2}$$

so that the Gauss-Newton formula (19.1) becomes equivalent to the Newton-Raphson formula for nonlinear optimization (p. 73).

Note that, in the previous formula:

1. $\mathbf{g}$ and $\mathbf{H}$ are scaled by a factor 1/2 since this factor cancels during the computations.

2. The expression of $\mathbf{H}$ is only approximate, since a factor containing the term $(y_i - \hat{y}_i)$ is neglected during the computation of the second partial derivatives:

$$\frac{\partial^2 \Phi}{\partial a \, \partial b} = \sum_{i=1}^{n} \left[\frac{\partial \hat{y}_i}{\partial a} \frac{\partial \hat{y}_i}{\partial b} - (y_i - \hat{y}_i) \frac{\partial^2 \hat{y}_i}{\partial a \, \partial b} \right]$$

The residual variance is:

$$V_r = \frac{SS_r}{n - p}$$

where p is the number of parameters in the model.

It is still possible to compute r^2 and F, as well as confidence intervals, but their interpretation is less straightforward since the ANOVA relationship (§ 17.1) does not hold for nonlinear models. In this case, r^2 may be > 1 ! Moreover, the distribution of the parameters is only approximately described by the Student distribution.

19.2 Monte-Carlo simulation

The distribution of the regression parameters may be simulated by the MCMC method discussed in § 15.2 (p. 107).

Let β denote the vector of model parameters. According to Bayes' theorem, the posterior probability density $\pi(\beta)$ of these parameters is given by:

$$\pi(\beta) = \frac{L(\beta)P(\beta)}{\int L(\beta)P(\beta)d\beta} = \frac{L(\beta)P(\beta)}{N}$$

where $P(\beta)$ denotes the prior probability density of the parameters and $L(\beta)$ denotes the likelihood, i.e. the probability of observing the experimental results (x_i, y_i) given the parameters.

The integral which appears in the denominator is usually too complex to be calculated and is therefore treated as a normalizing constant N.

Assuming that, for a given β, the residuals $(y_i - \hat{y}_i)$ are identically and independently distributed according to a normal distribution with variance σ^2, the likelihood is given by:

$$L(\beta) = \prod_{i=1}^{n} \left(\frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right] \right)$$

If we choose a uniform prior probability $P(\beta)$ over an interval $\mathcal{B}$, the posterior probability becomes:

$$\pi(\beta) = C \prod_{i=1}^{n} \exp\left[-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right]$$

where C is a constant.

In order to use the Metropolis-Hastings algorithm, as described in chapter 15.2, we define the function:

$$F(\beta) = \begin{cases} -2 \ln \frac{\pi(\beta)}{C} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 & \text{if } \beta \in \mathcal{B} \\ \\ \infty & \text{otherwise} \end{cases} \tag{19.3}$$

It is the same objective function than for the nonlinear regression algorithm, except that it is bounded on the interval $\mathcal{B}$.

19.3 Regression procedures

These procedures are defined in unit `unlfit`.

19.3.1 Optimization methods

DMath offers three deterministic optimizers: Marquardt, Simplex and BFGS (see chapter 10) and two stochastic optimizers: Simulated Annealing and Genetic Algorithm (see chapter 15)

The Marquardt method is selected by default. This selection can be changed with the statement `SetOptAlgo(Algo)` where `Algo` may have one of the following values:

`NL_MARQ`	for Marquardt
`NL_SIMP`	for Simplex
`NL_BFGS`	for BFGS
`NL_SA`	for Simulated Annealing
`NL_GA`	for Genetic Algorithm

The current algorithm is returned by function `GetOptAlgo`

19.3.2 Maximal number of parameters

By default, the maximal number of regression parameters is set to 10. This value may be changed with the statement `SetMaxParam(N)` where `N` is a number up to 255. Function `GetMaxParam` returns the current value.

19.3.3 Parameter bounds

It is assumed that each regression parameter varies within an interval $[a, b]$. By default, this interval is set to $[-10^6, 10^6]$ which is way too large for most applications. It is possible to change this interval with the statement:

```
SetParamBounds(I, ParamMin, ParamMax)
```

where `I` is the index of the parameter and `ParamMin` and `ParamMax` are the bounds.

Procedure `GetParamBounds(I, ParamMin, ParamMax)` returns the bounds for the parameter of index I.

Defining realistic intervals for the parameters is essential when using stochastic optimizers.

19.3.4 Check of parameters

Function `NullParam(B)` returns `True` if at least one of the components of vector `B` is null. This function can be used to check if the parameters heve been initialized properly.

19.3.5 Nonlinear regression

Nonlinear regression is performed by the two procedures:

- `NLFit(RegFunc, DerivProc, X, Y, Lb, Ub, MaxIter, Tol, B, FirstPar, LastPar, V)` for unweighted regression
- `WNLFit(RegFunc, DerivProc, X, Y, S, Lb, Ub, MaxIter, Tol, B, FirstPar, LastPar, V)` for weighted regression

where:

- `RegFunc` is the function to be fitted, defined as:

  ```
  function RegFunc(X : Float; B : TVector) : Float;
  ```

  where `X` is the independent variable and `B` the vector of regression parameters. This function is of type `TRegFunc`.

- `DerivProc` is the procedure used to compute the partial derivatives of the regression function with respect to the parameters. It is defined as:

  ```
  procedure DerivProc(X, Y : Float; B, D : TVector);
  ```

  where `D` is the vector of derivatives at point `(X, Y)` (one row of the Jacobian). This procedure is of type `TDerivProc`.

- `X[Lb..Ub]`, `Y[Lb..Ub]` are the point coordinates and `S[Lb..Ub]` are the standard deviations
- `MaxIter` is the maximum number of iterations for the optimization procedure
- `Tol` is the required precision for the regression parameters
- `B[FirstPar..LastPar]` is the vector of fitted parameters
- `V[FirstPar..LastPar, FirstPar..LastPar]` is the inverse matrix $(\mathbf{J}^\top\mathbf{J})^{-1}$

19.3.6 Monte-Carlo simulation

The statistical distribution of the regression parameters is simulated by the following procedures:

- `SimFit(RegFunc, X, Y, Lb, Ub, B, FirstPar, LastPar, V)` for unweighted regression
- `WSimFit(RegFunc, X, Y, S, Lb, Ub, B, FirstPar, LastPar, V)` for weighted regression

where the parameters have the same meaning than in the nonlinear regression procedures, except that here `V` is the variance-covariance matrix.

The results of the last simulation cycle are saved in an ASCII file. The name of this file may be defined by the statement `SetMCFile(FileName)`. The default file name is `mcmc.txt`

19.4 Demo programs

These BGI programs are located in the `demo\bgi\curfit` subdirectory.

- Program `regnlin.pas` performs a nonlinear least squares fit of the exponential model:

  $$y = ae^{-bx}$$

  The partial derivatives used to compute the Jacobian are:

  $$\frac{\partial y}{\partial a} = e^{-bx} \qquad \frac{\partial y}{\partial b} = -axe^{-bx}$$

Initial estimates of the parameters B are obtained by linearization:

$$\ln y = \ln a - bx$$

However, this transformation modifies the standard deviations of the independent variables:

$$\sigma(\ln y) \approx \mathrm{d} \ln y = \frac{\mathrm{d}y}{y} \approx \frac{\sigma(y)}{y}$$

It is therefore recommended to use weighted linear regression for this step.

Subroutine `ApproxFit` selects the data points for which the transformation is appropriate (i. e. $y > 0$) and stores the transformed coordinates and their standard deviations in 3 vectors `X1, Y1, S1` which are passed to the weighted linear regression subroutine `WLinFit`. The fitted parameters are returned in vector `A[0..1]`. They are then transformed back to the original form of the model:

```
B[1] := Exp(A[0]);
B[2] := - A[1];
```

Marquardt's method is then used to perform nonlinear minimization of the residual sum of squares, by means of subroutine `NLFit`. Function `RegFunc` and procedure `DerivProc` are defined as follows:

```
function RegFunc(X : Float; B : TVector) : Float;
begin
  RegFunc := B[1] * Exp(- B[2] * X);
end;

procedure DerivProc(X, Y : Float; B, D : TVector);
begin
  D[1] := Exp(- B[2] * X);
  D[2] := - B[1] * X * D[1];
end;
```

Since the parameter lists of these procedures cannot be modified, the other variables which they must access are made global.

The results of the minimization are printed as with the linear regression programs, except that the correlation coefficients are shown only if $r \leq 1$.

The program may be adapted to another regression model by changing the following parts:

- the function name (constant `FuncName`)
- the constants `FirstPar` and `LastPar` which define the bounds of the parameter array `B`
- the subroutine `ApproxFit` which computes the initial estimates of the parameters
- the definition of the regression model in function `RegFunc`
- the definition of derivatives in subroutine `DerivProc`

- Program `mcsim.pas` simulates the posterior distribution of the regression parameters for the previous exponential model.

  The settings for the MCMC procedure are defined as follows:

```
const
  NCycles  =   10;         { Number of cycles }
  MaxSim   = 1000;         { Max nb of simulations at each cycle }
  SavedSim = 1000;         { Nb of simulations to be saved }
  MCFile   = 'mcsim.txt';  { File for storing simulation results }
```

  The algorithm is initialized with:

```
InitMHParams(NCycles, MaxSim, SavedSim);
SetMCFile(MCFile);
```

  Proper intervals are defined for the two parameters:

```
SetParamBounds(1, 100, 1000);
SetParamBounds(2, 0.1, 1);
```

  The `SimFit` procedure is then called. After the computation is done, the program plots a graph showing the distribution of the parameters.

Chapter 20

Library of nonlinear regression models

DMath has a set of predefined nonlinear regression models which can be fitted just like the polynomial or multilinear models.

At this time, the following models are implemented:

- Rational fractions
- Sums of exponentials
- Increasing exponential
- Exponential + linear
- Logistic equations
- Power function
- Gamma distribution
- Michaelis equation
- Integrated Michaelis equation
- Hill equation
- Acid-base titration curve

20.1 Common features

20.1.1 Procedures

For each model, two fitting procedures are provided, for unweighted and weighted regression:

- `<model>Fit(X, Y, Lb, Ub, [...], MaxIter, Tol, B, V)`
- `W<model>Fit(X, Y, S, Lb, Ub, [...], MaxIter, Tol, B, V)`

where:

- `<model>` stands for model name
- `X[Lb..Ub]`, `Y[Lb..Ub]` are the point coordinates and `S[Lb..Ub]` are the standard deviations
- `[...]` stands for the additional parameters required by some models
- `MaxIter` is the maximum number of iterations for the optimization procedure
- `Tol` is the required precision for the regression parameters
- `B` is the vector of fitted parameters
- `V` is the inverse matrix

In addition, there exists a function `<model>Fit_Func(X, B)` which returns the value of the fitted function at point `X`, given the parameters `B`. This function should be called *after* the model has been fitted.

20.1.2 Optimization methods and initial parameters

As with the general nonlinear regression procedures studied in the previous chapter, the specific procedures can use the three local optimizers (Marquardt, BFGS, simplex) and the two global ones (simulated annealing and genetic algorithm), according to the choice performed by procedure `SetOptAlgo`.

Moreover, the maximal number of regression parameters and their bounds can still be modified by using procedures `SetMaxParam` and `SetParamBounds`, as described in the previous chapter.

If a local optimizer is selected, the fitting procedure will look at the current values of the regression parameters `B`. If *all* these values are nonzero, they will be used to start the algorithm. Otherwise, a built-in specific procedure will be called to generate approximate starting values (most often, using a linearized form of the model).

If a global optimizer is selected, the initial parameters will be randomly chosen within the parameter bounds.

20.2 Regression models

20.2.1 Rational fractions

The model has the form:

$$y = \frac{p_0 + p_1 x + p_2 x^2 + \cdots + p_{d_1} x^{d_1}}{1 + q_1 x + q_2 x^2 + \cdots + q_{d_2} x^{d_2}}$$

The fitted parameters are:

$$B_0 = p_0 \qquad B_1 = p_1 \qquad B_2 = p_2 \qquad \cdots \qquad B_{d1} = p_{d_1}$$

$$B_{d_1+1} = q_1 \qquad B_{d_1+2} = q_2 \qquad \cdots \qquad B_{d_1+d_2} = q_{d_2}$$

where d_1 and d_2 are the degrees of the numerator and denominator, respectively.

The fitting procedures are defined in unit `ufracfit` :

- `FracFit(X, Y, Lb, Ub, Deg1, Deg2, ConsTerm, MaxIter, Tol, B, V)`
- `WFracFit(X, Y, S, Lb, Ub, Deg1, Deg2, ConsTerm, MaxIter, Tol, B, V)`

where `Deg1, Deg2` are the degrees of the numerator and denominator, and `ConsTerm` is a boolean indicator which flags the presence of the constant term p_0.

The regression function is `FracFit_Func(X, B)`

20.2.2 Sums of exponentials

The model has the form:

$$y = Y_0 + A_1 e^{-a_1 x} + A_2 e^{-a_2 x} + \cdots$$

The fitted parameters are:

$$B_0 = Y_0$$

$$B_1 = A_1 \qquad B_2 = a_1$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$B_{2i-1} = A_i \qquad B_{2i} = a_i \qquad i = 1..N_{exp}$$

where N_{exp} is the number of exponentials.

The fitting procedures are defined in unit `uexpfit` :

- `ExpFit(X, Y, Lb, Ub, Nexp, ConsTerm, MaxIter, Tol, B, V)`
- `WExpFit(X, Y, S, Lb, Ub, Nexp, ConsTerm, MaxIter, Tol, B, V)`

where `Nexp` is the number of exponentials, and `ConsTerm` is a boolean indicator which flags the presence of the constant term Y_0.

The regression function is `ExpFit_Func(X, B)`

20.2.3 Increasing exponential

The model has the form:

$$y = Y_{min} + A(1 - e^{-kx})$$

The fitted parameters are:

$$B_0 = Y_{min} \qquad B_1 = A \qquad B_2 = k$$

The fitting procedures are defined in unit `uiexpfit` :

- `IncExpFit(X, Y, Lb, Ub, ConsTerm, MaxIter, Tol, B, V)`
- `WIncExpFit(X, Y, S, Lb, Ub, ConsTerm, MaxIter, Tol, B, V)`

where `ConsTerm` is a boolean indicator which flags the presence of the constant term Y_{min}.

The regression function is `IncExpFit_Func(X, B)`

20.2.4 Exponential + Linear

The model has the form:

$$y = A(1 - e^{-kx}) + Bx$$

The fitted parameters are:

$$B_0 = A \qquad B_1 = k \qquad B_2 = B$$

The fitting procedures are defined in unit `uexlfit` :

- `ExpLinFit(X, Y, Lb, Ub, MaxIter, Tol, B, V)`
- `WExpLinFit(X, Y, S, Lb, Ub, MaxIter, Tol, B, V)`

The regression function is `ExpLinFit_Func(X, B)`

20.2.5 Logistic functions

The model has the form:

$$y = A + \frac{B - A}{1 - e^{-ax+b}}$$

The fitted parameters are:

$$B_0 = A \qquad B_1 = B \qquad B_2 = a \qquad B_3 = b$$

The generalized logistic function has the form:

$$y = A + \frac{B - A}{(1 - e^{-ax+b})^n}$$

with the additional parameter $B_4 = n$

The fitting procedures are defined in unit `ulogifit` :

- `LogiFit(X, Y, Lb, Ub, ConsTerm, General, MaxIter, Tol, B, V)`
- `WLogiFit(X, Y, S, Lb, Ub, ConsTerm, General, MaxIter, Tol, B, V)`

where `ConsTerm` is a boolean indicator which flags the presence of the constant term A, and `General` is a boolean indicator which selects the generalized logistic.

The regression function is `LogiFit_Func(X, B)`

20.2.6 Power function

The model has the form:

$$y = Ax^n$$

The fitted parameters are:

$$B_0 = A \qquad B_1 = n$$

The fitting procedures are defined in unit `upowfit` :

- `PowFit(X, Y, Lb, Ub, MaxIter, Tol, B, V)`
- `WPowFit(X, Y, S, Lb, Ub, MaxIter, Tol, B, V)`

The regression function is `PowFit_Func(X, B)`

20.2.7 Gamma distribution

The model has the form:

$$y = a(x - b)^c \exp\left[-\frac{x - b}{d}\right]$$

The fitted parameters are:

$$B_1 = a \qquad B_2 = b \qquad B_3 = c \qquad B_4 = d$$

The fitting procedures are defined in unit `ugamfit` :

- `GammaFit(X, Y, Lb, Ub, MaxIter, Tol, B, V)`
- `WGammaFit(X, Y, S, Lb, Ub, MaxIter, Tol, B, V)`

The regression function is `GammaFit_Func(X, B)`

20.2.8 Michaelis equation

The model has the form:

$$y = \frac{Y_{max} x}{K_m + x}$$

The fitted parameters are:

$$B_0 = Y_{max} \qquad B_1 = K_m$$

where K_m is the Michaelis constant.

This equation is widely used in enzyme kinetics, with x being the substrate concentration and y the reaction rate. Therefore, Y_{max} is the maximal velocity, such that $Y_{max} = k_{cat} e_0$ where k_{cat} is the catalytic constant and e_0 the total enzyme concentration.

The fitting procedures are defined in unit `umichfit` :

- `MichFit(X, Y, Lb, Ub, MaxIter, Tol, B, V)`
- `WMichFit(X, Y, S, Lb, Ub, MaxIter, Tol, B, V)`

The regression function is `MichFit_Func(X, B)`

20.2.9 Integrated Michaelis equation

The integrated Michaelis equation is the solution to the differential equation:

$$\frac{dp}{dt} = \frac{Y_{max}(s_0 - p)}{K_m + s_0 - p}$$

with the initial condition: $p = 0$ at $t = 0$.

It is also used in enzyme kinetics, with p being the product concentration at time t and s_0 the initial substrate concentration.

The solution is expressed in terms of Lambert's W-function, studied in chapter 4:

$$p = s_0 - K_m \mathrm{W} \left[\frac{s_0}{K_m} \exp \left(\frac{s_0 - k_{cat} e_0 t}{K_m} \right) \right]$$

where e_0 is the total enzyme concentration. The independent variable may be t, s_0 or e_0.

The fitting procedures are defined in unit `umintfit` :

- `MintFit(X, Y, Lb, Ub, MintVar, Fit_S0, MaxIter, Tol, B, V)`

- `WMintFit(X, Y, S, Lb, Ub, MintVar, Fit_S0, MaxIter, Tol, B, V)`

where:

- `MintVar` denotes the independent variable (may be `Var_T`, `Var_S` or `Var_E`)
- `Fit_S0` indicates if s_0 must be fitted (for `Var_T` or `Var_E` only)

So, the fitted parameters are as follows:

Indep. var.	`MintVar`	B_0	B_1	B_2
t	`Var_T`	s_0	K_m	Y_{max}
s_0	`Var_S`		K_m	$Y_{max}t$
e_0	`Var_E`	s_0	K_m	$k_{cat}t$

The regression function is `MintFit_Func(X, B)`

20.2.10 Hill equation

The Hill equation can be viewed as an extension of the Michaelis equation:

$$y = A + \frac{B - A}{1 + (K/x)^n}$$

The fitted parameters are:

$$B_0 = A \qquad B_1 = B \qquad B_2 = K \qquad B_3 = n$$

The fitting procedures are defined in unit `uhillfit` :

- `HillFit(X, Y, Lb, Ub, ConsTerm, MaxIter, Tol, B, V)`
- `WHillFit(X, Y, S, Lb, Ub, ConsTerm, MaxIter, Tol, B, V)`

where `ConsTerm` is a boolean indicator which flags the presence of the constant term A

The regression function is `HillFit_Func(X, B)`

20.2.11 Acid-base titration curve

The model has the form:

$$y = A + \frac{B - A}{1 - 10^{pK_a - x}}$$

The fitted parameters are:

$$B_0 = A \qquad B_1 = B \qquad B_2 = pK_a$$

It is used in chemistry, where x is the pH, y is some property (e.g. absorbance) which depends on the ratio of the acidic and basic forms of a compound, A is the property for the pure acidic form, B is the property for the pure basic form and pK_a is the acidity constant.

The fitting procedures are defined in unit `upkfit` :

- `PKFit(X, Y, Lb, Ub, MaxIter, Tol, B, V)`
- `WPKFit(X, Y, S, Lb, Ub, MaxIter, Tol, B, V)`

The regression function is `PKFit_Func(X, B)`

20.3 Demo programs

20.3.1 BGI programs

- The following programs are located in `demo\bgi\regmodel`:

Program	Model	Data files	Ref.
`regfrac.pas`	Rational fraction	`frac.dat`	(^{1})
`regexpo.pas`	Sum of exponentials	`iv2.dat`	(^{2})
		`oral1.dat`	(^{2})
		`oral2.dat`	(^{2})
`regiexpo.pas`	Increasing exponential	`iexpo.dat`	(^{1})
`regexlin.pas`	Exponential + linear	`exlin.dat`	(^{1})
`reglogi.pas`	Logistic function	`logist.dat`	(^{3})
`regamma.pas`	Gamma distribution	`gamma.dat`	(^{4})
`regmich.pas`	Michaelis equation	`michael.dat`	(^{1})
`regmint.pas`	Integrated Michaelis equation	`michint.dat`	(^{1})
`reghill.pas`	Hill equation	`hill.dat`	(^{1})

The data were taken from the following references:

1. Enzyme kinetic data from the author's laboratory
2. Pharmacokinetic data from M. GIBALDI & D. PERRIER, Pharmacokinetics, 2nd edition, Dekker 1982
3. Biochemical data from S. HUET *et al.*, Statistical Tools for Nonlinear Regression, Springer 1996
4. Example provided by Chris Rorden

Each program reads the data file, then sets the indices of the parameters, the maximal number of regression parameters and the parameter bounds. For instance, for the rational fraction:

```
if ConsTerm then FirstPar := 0 else FirstPar := 1;
LastPar := Deg1 + Deg2;

SetMaxParam(LastPar);

for I := 0 to LastPar do
  SetParamBounds(I, -1000, 1000);
```

where `FirstPar` and `LastPar` are the indices of the first and last regression parameters.

The program then fits the model and plots the resulting curve.

- Program `nist.pas`, located in `demo\bgi\nist`, performs a validation of some of the regression routines in DMath by using reference data from the NIST (National Institute of Standards and Technology, `http://www.itl.nist.gov/div898/strd/general/dataarchive.html`).

  The best results are obtained in extended precision. The program uses a specific unit, `umodels.pas` which defines the regression models and calls the appropriate regression routines.

  For each model, the precision of the fit is estimated, for each fitted parameter, by computing the relative difference ϵ_i between the value found by the program and the reference value given by the NIST. The highest relative difference $\epsilon_{max} = \max(|\epsilon_i|)$ is then selected to compute the precision as $p = -\log_{10} \epsilon_{max}$, which corresponds approximately to the number of correct digits found by the program. A plot of the precision obtained for the different models is displayed.

According to the NIST, a precision of at least 4 digits is acceptable. In our tests, precisions ranging from 4 to 14 digits were obtained by selecting the algorithms as follows:

- for multilinear and polynomial regressions, the Singular Value Decomposition (SVD) algorithm was used, with a threshold for the singular values set at `N * MachEp` where `N` is the number of points (see paragraph 9.8 for details).
- for nonlinear regression, the Simplex algorithm was applied first, followed by Marquardt's method. The initial values of the parameters were computed by the built-in DMath routines. The tolerance for the fitted parameters was the same than for SVD.

The results are saved in a file `nist.out`. For each model, the relative difference (ϵ_i) is computed for:

- the regression parameters b_i
- their standard deviations s_i
- the residual standard deviation s_r
- the coefficient of determination r^2
- the variance ratio F

Note that reference values for the last two tests are not available for all models.

20.3.2 GUI program

Program `curfit.dpr`, already studied in the previous chapters, can fit any of the above models by nonlinear regression. The options specific to these models are the following:

- *Compute / Select Model* : selects the regression model.
- *Compute / Select Algorithm* : selects the regression algorithm.

Example data files are provided in `demo\bgi\regmodel`.

Chapter 21

Complex numbers and functions

This chapter describes the complex functions available in DMath. Most of them are adapted from the Delphi library ComplexMath by E. F. Glynn (`http://www.efg2.com/Lab/Mathematics/Complex/index.html`). They are defined in the unit `ucomplex`.

21.1 Introduction to complex numbers

21.1.1 Definitions

A complex number Z is a pair of real numbers:

$$Z = (X, Y)$$

It is related to the coordinates of a point M in the plane (called the *complex plane* in this case).

Complex addition and subtraction are defined as for vectors:

$$Z_1 = (X_1, Y_1)$$

$$Z_2 = (X_2, Y_2)$$

$$Z_1 + Z_2 = (X_1 + X_2, Y_1 + Y_2)$$

$$Z_1 - Z_2 = (X_1 - X_2, Y_1 - Y_2)$$

A complex number may also be written as $Z = X + iY$ where i is the number (0,1). This notation defines the *rectangular form* of the complex.

The coordinates X and Y are often called the *real part* and the *imaginary part*, noted respectively $\Re(Z)$ and $\Im(Z)$.

The complex product is defined as:

$$Z_1 \cdot Z_2 = (X_1X_2 - Y_1Y_2, X_1Y_2 + Y_1X_2)$$

It is related to the rotation of vectors in the plane.

The complex product is commutative, i. e. $Z_1Z_2 = Z_2Z_1$

The number i is such that $i^2 = (0,1) \cdot (0,1) = (-1,0) = -1$

Every complex $Z \neq 0$ has an inverse Z^{-1} for the multiplication.

21.1.2 Polar form

A complex number $Z = X + iY$ can also be written as $R(\cos\theta + i\sin\theta)$, where:

- R is the norm of the vector $\overrightarrow{OM}$, also called the *modulus* or the absolute value of the complex number and noted $|Z|$
- θ is the angle between the Ox axis and the vector $\overrightarrow{OM}$, counted in the interval $]-\pi,\pi]$. It is often called the *argument* or the *phase* of the complex number.

This notation defines the *polar form* of the complex number.

We have the following relationships:

$$R = \sqrt{X^2+Y^2} \qquad \theta = \arctan\frac{Y}{X} = \arctan 2(Y,X)$$

$$X = R\cos\theta \qquad Y = R\sin\theta$$

21.1.3 Exponential form

It has been shown that the exponential function can be extended to complex numbers, and that:

$$\exp(i\theta) = \cos\theta + i\sin\theta$$

So, the notation $R(\cos\theta + i\sin\theta)$ is equivalent to $R\exp(i\theta)$, where all the classical properties of the exponential apply, for instance:

$$Z_1 = R_1\exp(i\theta_1) \quad , \quad Z_2 = R_2\exp(i\theta_2) \quad \Rightarrow \quad Z_1Z_2 = R_1R_2\exp[i(\theta_1+\theta_2)]$$

21.2 Type definition

Type `Complex` is defined in unit `utypes` as:

```
type Complex = record
  X, Y : Float;
end;
```

So, a complex variable `Z` is declared as follows:

```
var Z : Complex;
```

Its real and imaginary parts are `Z.X` and `Z.Y`

The complex variables can be initialized, for instance:

```
var Z : Complex = (X : 1; Y : 2);  { Z = 1 + 2i }
```

You can also declare arrays of complexes and initialize them, for instance:

```
var Z : array[1..2] of Complex =
  ((X : 1; Y : 2),
   (X : 2; Y : 1));
```

21.3 Error handling

Function `MathErr()` returns the error code from the last function evaluation, as described in paragraph 3.1.

21.4 Number construction

The following functions create a complex number from either its rectangular or polar coordinates:

- Function `Cmplx(X, Y)` returns the complex number $X + iY$
- Function `Polar(R, Theta)` returns the complex number $R(\cos\theta + i\sin\theta)$

Functions `CReal(Z)` and `CImag(Z)` return the real part and the imaginary part of their complex argument Z.

21.5 Sign and exchange

- Function `CSgn(Z)` returns the sign of the complex Z, such that:

$$\mathrm{CSgn}(Z) = \begin{cases} 1 & \text{if} \quad \Re(Z) > 0 \quad \text{or} \quad \Re(Z) = 0 \quad \text{and} \quad \Im(Z) > 0 \\ -1 & \text{if} \quad \Re(Z) < 0 \quad \text{or} \quad \Re(Z) = 0 \quad \text{and} \quad \Im(Z) < 0 \end{cases}$$

  This function is used to determine in which half-plane ('left' or 'right') of the complex plane the number Z is located.

- Procedure `CSwap(Z1, Z2)` exchanges the two complex numbers Z_1 and Z_2.

21.6 Modulus and argument

- Functions `CAbs(Z)` and `CArg(Z)` give, respectively, the modulus and the argument of the complex Z, i. e. the numbers R and θ such that $Z = R\exp(i\theta)$ with $\theta \in]-\pi, \pi]$.

- Function `CAbs2(Z)` returns the squared modulus of Z, i. e. $X^2 + Y^2$.

21.7 Other functions

- Function `CConj(Z)` returns the conjugate of Z i.e. $\bar{Z} = X - iY$

- Function `CNeg(Z)` returns the opposite $-Z$.

- Function `CInv(Z)` returns the inverse $1/Z$.

21.8 Arithmetic functions

- Functions `CAdd(Z1, Z2)`, `CSub(Z1, Z2)`, `CMul(Z1, Z2)`, `CDiv(Z1, Z2)` return respectively the sum $Z_1 + Z_2$, difference $Z_1 - Z_2$, product $Z_1 * Z_2$ and quotient Z_1/Z_2 of their complex arguments.

- Function `CSqr(Z)` returns the square Z^2.

21.9 Polynomials

Function `CPoly(Z, Coef, Deg)` evaluates the polynomial:

$$P(Z) = \mathrm{Coef}[0] + \mathrm{Coef}[1] \cdot Z + \mathrm{Coef}[2] \cdot Z^2 + \cdots + \mathrm{Coef}[\mathrm{Deg}] \cdot Z^{\mathrm{Deg}}$$

where Z is complex and the coefficients are real.

21.10 Logarithm and exponential

It is obvious from the relationship below that the complex logarithm is a multi-valued function:

$$Z = R \cdot \exp(i\theta) = R \cdot \exp\left[i\left(\theta + 2k\pi\right)\right] \ \Rightarrow\ \ln Z = \ln R + i\left(\theta + 2k\pi\right)$$

- Function `CLn(Z)` returns the *principal part* of the logarithm, i. e. the value corresponding to $k = 0$ and $\theta \in]-\pi, \pi]$.

- Function `CExp(Z)` returns the exponential of Z, according to:

$$\exp(X + iY) = e^X(\cos Y + i \sin Y)$$

21.11 Power functions

- Function `CIntPower(Z, N)` computes Z^N, where N is integer, by repeated multiplications with Legendre's algorithm to minimize the number of operations. For instance, Z^8 is computed as $Z^2 = Z \cdot Z$, $Z^4 = Z^2 \cdot Z^2$ and $Z^8 = Z^4 \cdot Z^4$, hence 3 multiplications instead of 7.

- Function `CRealPower(Z, X)` computes Z^X, where X is real, by DeMoivre's theorem :

$$Z^X = [R \exp(i\theta)]^X = R^X \exp(Xi\theta) = R^X(\cos X\theta + i \sin X\theta)$$

  If the fractional part of X is null, the function `CIntPower` is called.

- Function `CPower(Z1, Z2)` computes $(Z_1)^{Z_2}$, where both Z_1 and Z_2 are complexes, by the formula:

$$(Z_1)^{Z_2} = \exp(Z_2 \ln Z_1)$$

21.12 Complex roots

According to the following relationship:

$$Z = R \cdot \exp(i\theta) = R \cdot \exp\left[i\left(\theta + 2k\pi\right)\right] \;\Rightarrow\; Z^{1/n} = R^{1/n} \cdot \exp\left[i\left(\frac{\theta}{n} + \frac{2k\pi}{n}\right)\right]$$

a complex number has n distinct n-th roots, corresponding to $k = 0 \cdots (n-1)$

- Function `CRoot(Z, K, N)` returns the K-th N-th root of the complex number Z (K and N are integers).
- Function `CSqrt(Z)` returns the first square root of the complex number Z. It is therefore equivalent to `CRoot(Z, 0, 2)`.

21.13 Trigonometric functions

The following functions are available (where $Z = X + iY$):

Function	Formula
`CSin(Z)`	$\sin X \cosh Y + i \cos X \sinh Y$
`CCos(Z)`	$\cos X \cosh Y - i \sin X \sinh Y$
`CTan(Z)`	$\frac{\sin X \cos X + i \sinh Y \cosh Y}{\cos^2 X + \sinh^2 Y} \qquad Z \neq \frac{\pi}{2} + k\pi$
`CASin(Z)`	$\arcsin(P - Q) + i \ \mathrm{csgn}(Y - iX) \ln(P + Q + \sqrt{(P+Q)^2 - 1})$ $P = \frac{1}{2}\sqrt{X^2 + 2X + 1 + Y^2} \quad Q = \frac{1}{2}\sqrt{X^2 - 2X + 1 + Y^2}$
`CACos(Z)`	$\frac{\pi}{2} - \arcsin(Z)$
`CATan(Z)`	$\frac{1}{2}[\arctan(X, 1 - Y) - \arctan(-X, 1 + Y)] + \frac{1}{4} i \ln \frac{X^2 + (Y+1)^2}{X^2 + (Y-1)^2} \qquad Z \neq \pm i$

In addition, procedure `CSinCos(Z, SinZ, CosZ)` allows to calculate the sine and cosine simultaneously, saving some computations if both functions are needed.

21.14 Hyperbolic functions

The following functions are available (where $Z = X + iY$):

Function	Formula
`CSinh(Z)`	$\sinh X \cos Y + i \cosh X \sin Y$
`CCosh(Z)`	$\cosh X \cos Y + i \sinh X \sin Y$
`CTanh(Z)`	$\frac{\sinh X \cosh X + i \sin Y \cos Y}{\sinh^2 X + \cos^2 Y}$ $\quad Z \neq i\left(\frac{\pi}{2} + k\pi\right)$
`CASinh(Z)`	$-i \arcsin(iZ)$
`CACosh(Z)`	$\mathrm{csgn}[Y + i(1 - X)] \cdot i \arccos(Z)$
`CATanh(Z)`	$-i \arctan(iZ) \quad Z \neq \pm 1$

In addition, procedure `CSinhCosh(Z, SinhZ, CoshZ)` allows to calculate the hyperbolic sine and cosine simultaneously, saving some computations if both functions are needed.

21.15 Gamma function

Function `CLnGamma(Z)` returns the natural logarithm of the Gamma function for the complex argument Z.

21.16 Demo program

Program `testcomp.pas`, located in `demo\console\complex`, checks the accuracy of the complex functions. It is a slight modification of a Pascal program written by E. Glynn.

The program defines an array of 20 complex numbers. The tests consist mostly of applying a function to each number, then applying the reciprocal function to the result, in order to retrieve the original number.

Chapter 22

Fractals and chaos

We will study here two examples of chaotic systems and fractal graphics:

- the Mandelbrot and Julia sets
- the quadratic iterator

22.1 Mandelbrot and Julia sets

22.1.1 Introduction

These sets are created by iteration of a function of complex variable $z \mapsto f(z) + c$, where c is a complex constant. The present discussion is limited to the power function $f(z) = z^p$, where p is an integer or real exponent > 1.

22.1.2 The Mandelbrot set

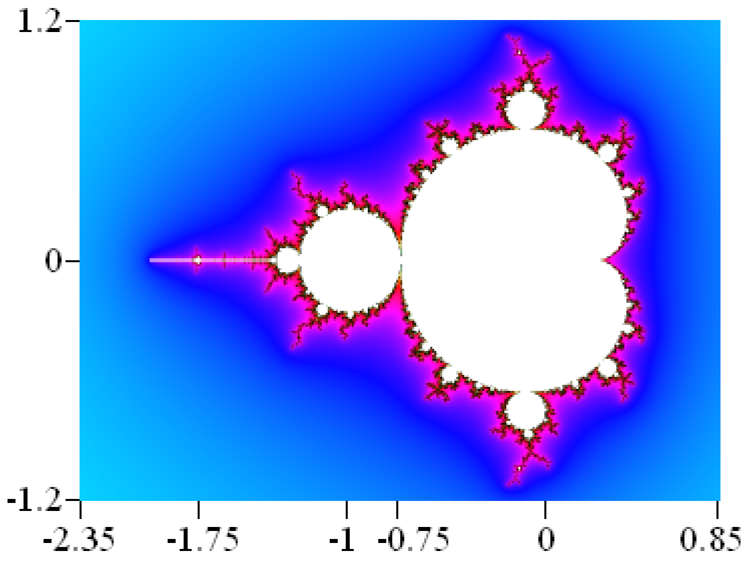
This set, named after the mathematician Benoît Mandelbrot, corresponds to the case $p = 2$. More precisely, it is defined as the set of complex numbers c for which the sequence:

$$z_0 = c \qquad z_n = z_{n-1}^2 + c$$

converges to a finite value.

In practice, the sequence is iterated at each point c in the complex plane. The points belonging to the set are given the same color, and the points outside the set are given a color which depends on their divergence rate.

A global view of the Mandelbrot set is shown here:



- The Mandelbrot set is in white. It is comprised of several parts:
  - a cardioid (heart-shaped curve, at right) which constitutes the main part of the set. The 'cusp' of the cardioid is at (0.25, 0)
  - a main disk of radius 0.25 centered at (-1, 0) and tangent to the cardioid at (-0.75, 0)
  - a straight 'tip' (leftmost part of the set), showing a mini-Mandelbrot set at the abscissa -1.75

  The set is covered with 'bulbs' of various sizes emitting filaments. The major bulbs are located at the poles of the cardioid. Each bulb is itself covered with other bulbs, while the filaments give rise to small copies (more or less distorted) of the set. All these structures are repeated at all scales.

  Mathematicians have shown that:

  1. the Mandelbrot set is included in the circle of radius 2: the complex numbers having a modulus higher than 2 don't belong to the set.
  2. the set is connected: all its parts are linked by filaments which can be infinitely thin and therefore not visible on the pictures.

- The immediate vicinity of the set is plotted in dark colors, using the *distance estimator method* which will be described later.

- The outside of the set contains the points for which the sequence tends towards infinity. It is colored according to the number of iterations needed to reach a value called the *escape radius*. When a point gets closer to the set, the number of iterations increases, and it increases faster as the point approaches the set.

22.1.3 The Julia sets

These sets, named after the mathematician Gaston Julia, are defined in a manner similar to the Mandelbrot set, except that here the parameter c is constant and the sequence is initialized with the coordinates z_{pixel} of the point:

$$z_0 = z_{pixel} \qquad z_n = z_{n-1}^2 + c$$

So, there is a Julia set for each value of the parameter c.

It has been shown that:

- When the point c belongs to the Mandelbrot set, the Julia set is connected, i. e. as a whole.
- Otherwise, the Julia set is made of an infinity of similar parts.

22.1.4 Variation of the exponent

Integer exponent

For the integer values of p (> 2) the Mandelbrot set is centered at (0,0) and is comprised of $(p-1)$ symmetrical lobes. When p is even, one of these lobes is located along the negative Ox axis. There is no such lobe when p is odd.

The corresponding Julia sets have a symmetry of order p.

Non-integer exponent

The pictures display discontinuities. This is due to the fact that, for non-integer values of p, z^p is evaluated as $\exp(p \ln z)$. But the logarithm of a complex number is a multi-valued function. The discontinuities arise from the choice of a single value at each point.

It is interesting to study the transition from an odd integer p to an even integer while following the intermediate values. We can observe the progressive formation of the lobe located along the Ox axis. This lobe is formed by fusion of multiple 'buds' which develop progressively and are surrounded by complex structures.

22.1.5 Theoretical aspects

Iteration of the complex function

The sequence $z_n = (z_{n-1})^p + c$ is iterated until the modulus $|z_n|$ exceeds the escape radius, or the iteration number n exceeds a predefined value. In the last case, the point is considered as belonging to the set.

The distance estimator

As its name implies, this parameter estimates the distance between the tested point and the set. It is computed by using the fact that the iteration number required to reach the escape radius increases faster as the point approaches the set. This rate is estimated by the derivative of the iterated function.

For the Mandelbrot set, the function depends on the parameter c. We must then derive with respect to this parameter:

$$z'_n = p \cdot (z_{n-1})^{p-1} \cdot z'_{n-1} + 1 \qquad z'_0 = 0$$

For the Julia set, the parameter c is constant. The derivative reduces to:

$$z'_n = p \cdot (z_{n-1})^{p-1} \cdot z'_{n-1} \qquad z'_0 = 1$$

At the end of the iterations, the distance estimator is given by:

$$D = \frac{p|z_n| \ln |z_n|}{|z'_n|}$$

The weaker this value, the closer we are to the set.

The *continuous dwell* method

As we have shown previously, points outside the set are colored according to the iteration number required to reach the escape radius. At the vicinity of the set, this number varies rapidly from one point to another, resulting in a color mixing which makes the picture look 'fuzzy'. To avoid this effect, it is possible to 'smooth' the variation of iteration numbers by means of a logarithmic transform. The formula used here is the following:

$$\text{Dwell} = n + \log_p \frac{\ln \text{Esc}}{\ln |z_n|}$$

where Esc denotes the escape radius. The function $\log_p$ is the base-p logarithm, such that $\log_p z = \ln z / \ln p$

22.1.6 Programming aspects

We will describe here some of the techniques used in the demo program `mandel`.

Picture format

The picture is 640 $\times$ 480, 32 bits. It is centered at (x_0, y_0). The zoom is defined by the `ZoomFact` parameter, such that the value `ZoomFact` = 1 corresponds to a vertical scale of 4, resulting in a horizontal scale of $4 \times (640/480) \approx 5,33$. The vertical scale for a given `ZoomFact` value is therefore (4 / `ZoomFact`). If `Scale` denotes the pixel scale, we can define a scale factor:

```
ScaleFact := 4 / (Scale * ZoomFact);
```

This factor represents the distance between 2 pixels. So, the pixel coordinates (`Nx`, `Ny`) of a point are converted to the algebraic coordinates (`x`, `y`) by:

```
x := x0 + ScaleFact * (Nx - HalfPicWidth);
y := y0 - ScaleFact * (Ny - HalfPicHeight);
```

where `HalfPicWidth` and `HalfPicHeight` denote the half-width and half-height of the picture, i. e. 320 and 240 in our case.

A complete scan of the picture (`Image1`) is performed with two loops:

```
for Ny := 0 to Pred(PicHeight) do
  begin
    yt := y0 - ScaleFact * (Ny - HalfPicHeight);
    for Nx := 0 to Pred(PicWidth) do
      begin
        xt := x0 + ScaleFact * (Nx - HalfPicWidth);
        Image1.Canvas.Pixels[Nx, Ny] := Mandelbrot(xt, yt);
      end;
  end;
```

where `PicWidth` and `PicHeight` denote the width and height of the picture, i. e. 640 and 480 in our case. The `Mandelbrot` function returns the color of the point.

Computing the iterations

The `Mandelbrot` function performs the iterations at the point of algebraic coordinates (`xt`, `yt`) by computing simultaneously the complex function z_n and its derivative z'_n, according to the following code:

```
if Julia then
  begin
    c  := Cmplx(c_X, c_Y);
    z  := Cmplx(xt, yt);
    dz := Cmplx(1, 0);
  end
else
  begin
    c  := Cmplx(xt, yt);
    z  := Cmplx(0, 0);
    dz := z;
  end;

Iter := 0;
Module := CAbs(z);

while (Iter < MaxIter) and (Module < Esc) do
  begin
    zp1 := CRealPower(z, p1);  { z^(p-1) }

    zn   := CMul(z, zp1);
    zn.X := zn.X + c.X;
    zn.Y := zn.Y + c.Y;        { z(n+1) = z(n)^p + c }

    dzn   := CMul(zp1, dz);    { Derivative : dz(n+1)/dc }
    dzn.X := p * dzn.X;
    dzn.Y := p * dzn.Y;

    if not Julia then dzn.X := dzn.X + 1.0;

    Module := CAbs(zn);

    z := zn;
    dz := dzn;
    Inc(Iter);
 end;
```

where:

- `Julia` indicates if we are computing a Julia set
- `cJulia` corresponds to the parameter c for the Julia set
- the functions `Cmplx`, `CAbs`, `CMul` have been described in the previous chapter
- `p1 = p - 1`

The iterations can be terminated in two ways:

- If the iteration number reaches `MaxIter`, the point is considered as belonging to the Mandelbrot set, and plotted in white:

  ```
  if Iter = MaxIter then
    begin
      Result := clWhite;
      Exit;
    end;
  ```

  This method does not ensure that the sequence converges since rigorously we should perform an infinite number of iterations! We take the risk to include in the set some points located very closely to its frontier. This will result a 'fuzzy' aspect on the graphic. In this case, the solution is to increase `MaxIter`, at the expense of the computation time.

- If the modulus of the complex number reaches the value of the escape radius, the sequence diverges and the point does not belong to the set. In this case, we compute the distance estimator and the continuous dwell :

  ```
  LnMod := Ln(Module); DMod := CAbs(dzn);
  if DMod > 0.0 then Dist := p * Module * LnMod / DMod;
  Dwell := Iter - Ln(LnMod) / Lnp + LLE;
  ```

  where `Lnp` denotes the natural logarithm of p and `LLE` $= \log_p(\ln \mathrm{Esc})$

  The value of the escape radius has been fixed at 10^{10}. Such a high value is required by the distance estimator algorithm.

Color attribution

We use a simplified version of an algorithm described by Robert Munafo (`http://mrob.com/pub/muency/color.html`). Its main features are the following:

- The colors are defined in the HSV (Hue, Saturation, Value) space.
- The value V is attributed according to the distance estimator, so that the points close to the set appear in dark shades. This facilitates the detection of the tiniest parts of the set, which may not be visible at a given zooming.
- The hue H and saturation S are attributed according to the continuous dwell.

22.1.7 Demo program

Program `mandel.dpr`, located in `\demo\gui\mandel`, plots the Mandelbrot or Julia sets by using the algorithms described previously.

Compilation

It is recommended to build the program in extended precision to allow magnifications up to 10^{17} (instead of 10^{14} in double precision). This can be done:

- from the IDE, by defining the symbol `EXTENDEDREAL` in the project options, then build the program.
- from the command line, by:

  ```
  dcc32 mandel.dpr -b -dEXTENDEDREAL
  ```

  Choosing the 'build' option (`-b`) instead of 'make' will ensure that all the relevant units are compiled with the same precision.

Creating the standard picture

When the program is launched for the first time, it displays the default parameters for the classical Mandelbrot set:

p	:	exponent	= 2
(X, Y)	:	coordinates of the center of the picture	= (-0.75, 0)
Max. Iter.	:	maximal number of iterations	= 200
Zoom Fact.	:	zoom factor	= 1.75
Dist. Fact.	:	Distance estimator factor (for coloring)	= -1
Color Fact.	:	factor which controls the color range	= -2

Click on the 'Graph' button to start the computation. You will have to wait until the picture is displayed.

Coloring methods

The picture is colored in the HSV (Hue, Saturation, Value) color space.

The value (luminosity) V is determined by the distance between the pixel and the set. The points near the set are dark, while the set itself is white. So, the contour of the set is always visible, even if the set itself is too tiny to appear at the picture resolution. The luminosity is controlled by the `DistFact` parameter and increases when the parameter increases.

The hue H and saturation S are determined by the number of steps needed for the modulus of the complex number z to reach a predetermined value. The color density is controlled by the `ColorFact` parameter : a high (absolute) value means more colors on the picture. For a black and white picture, set `ColorFact` to zero.

If `ColorFact` is positive, the alternance of color stripes surrounding the set is made more obvious by darkening every other stripe. If `ColorFact` is negative, this effect is suppressed, so that the colors appear more like a continuous gradient.

Creating a new picture of the Mandelbrot set

1. Click on a point of the displayed picture: this point will become the center of the new picture, and its coordinates will appear in the 'X' and 'Y' boxes.

2. If necessary, modify the `MaxIter`, `ZoomFact`, `DistFact` or `ColorFact` values

3. Click on the 'Graph' button to start the picture generation.

Creating a new picture of a Julia set

1. Click on the 'Mandelbrot/Julia' button. The coordinates of the current Mandelbrot picture will become the c parameter of the Julia set, appearing in the 'Julia c parameter' box. The old coordinates will become zero, to ensure that the Julia set is centered at (0, 0).
2. If necessary, modify the `MaxIter`, `ZoomFact`, `DistFact` or `ColorFact` values
3. Click on the 'Graph' button to start the picture generation.

Saving the picture

Click on the 'Save file' button and enter a name for the parameter file (`*.par`). The parameters of the current picture will be saved in this file, and the displayed picture will be saved in a bitmap file (`*.bmp`) having the same file name.

The parameter file is a text file containing the following lines:

```
p
X
Y
MaxIter
ZoomFact
DistFact
ColorFact
c_X
c_Y
```

where `c_X` and `c_Y` are the coordinates of the c parameter for the Julia set. These coordinates are always 0 for a Mandelbrot set, while the coordinates `(X, Y)` are always 0 for a Julia set.

Loading a picture

Click on the 'Open file' button and select a parameter file. The parameters will be displayed in the relevant boxes. If a matching bitmap exists it will be displayed, otherwise the current picture will be erased.

22.1.8 Examples

A series of parameter files is provided to demonstrate some interesting aspects of the Mandelbrot and Julia sets.

Mandelbrot sets with variable exponent

`mandel_2`	Classical Mandelbrot set, $p = 2$
`mandel_3`	Odd integer exponent: $(p - 1)$ symmetry, no lobe on Ox
`mandel_4`	Even integer exponent: $(p - 1)$ symmetry, lobe on Ox
`mandel_3_5`	Noninteger exponent: discontinuities and incomplete Ox lobe
`mandel_3_85`	Transition from odd exponent to even exponent, showing the complex structures which give rise to the Ox lobe
`mandel_3_85_a`	Zoom on the complex structures of the previous set

Julia sets

`julia_2_a`	Connected Julia set: $p = 2, c = (-0.75, 0)$
`julia_2_b`	Disconnected Julia set: $p = 2, c = (-0.75, 0.1)$
`julia_5`	Julia set: $p = 5$, symmetry of order p

Exploration of the *seahorse valley*

This region is located between the cardioid and the main disk of the classical Mandelbrot set. The parameter files correspond to successive magnifications.

`seahorse_01`	The two sides of the valley
`seahorse_02a`	West side (story ?)
`seahorse_02b`	East side, showing a seahorse
`seahorse_03`	Zoom on the 'tail' of the seahorse
`seahorse_03a`	Mini Mandelbrot set at the junction of two spirals
`seahorse_04`	Zoom on the miniset
`seahorse_05`	The seahorse valley of the miniset
`seahorse_06`	Spirals in the seahorse valley
`seahorse_07`	An 'Embedded Julia Set' (EJS) at the junction of 2 spirals
`seahorse_08`	Zoom on the EJS
`seahorse_09`	At the center of the EJS: the 'nucleus'
`seahorse_10`	At the center of the nucleus: the 'nucleolus' ...
`seahorse_11`	... showing a mini Mandelbrot set ...
`seahorse_12`	... which is magnified here

Note: these pictures take a long time to generate, so be patient !

22.2 The quadratic iterator

The three programs located in `demo\gui\quadrit` demonstrate a chaotic system: the *quadratic iterator*, also known as the *logistic equation.*

The system is defined by the relationship:

$$x_{n+1} = ax_n(1 - x_n)$$

According to the value of a the sequence may be:

- periodic (e.g. for $a = 3.5$ the period is 4)
- chaotic (e.g. $a = 4$)
- presenting an alternance of periodic and chaotic phases, a phenomenon known as *intermittency* (e.g. $a = 3.82812$)

The programs can plot three types of diagrams:

- an *orbit diagram* (program `orbit.dpr`): the value of x_n is plotted vs. n. The initial value x_0 can be specified by the user or chosen at random ('Rnd' checkbox).
- a *bifurcation diagram* (program `bifur.dpr`): the values of x_n are plotted for each value of a (after an initial run of some hundred iterations). The bifurcation diagram shows the progressive transition from order to chaos via a period-doubling route as a increases. It also shows the existence of periodic regions inside the chaotic domain. Magnifying these regions reveals a structure similar to the whole plot (i.e. the fractal nature of the bifurcation diagram).
- a *FFT diagram* (program `fft.dpr`): the modulus of the Fast Fourier Transform of the sequence x_n is plotted as a function of the frequency. The period of the sequence is equal to the inverse of the lowest frequency giving a peak in the FFT.

Some examples of periodic/chaotic behavior are given here:

- The period-doubling route begins at $a = 3$:

a	Period	Frequency
3	2	0.5
3.5	4	0.25
3.55	8	0.125
3.566	16	0.0625

- The chaotic domain begins at $a \approx 3.57$

- Some periodic regions inside the chaotic domain:

a	Period	Frequency
3.63	6	0.1666...
3.74	5	0.2
3.83	3	0.333...

22.3 Links

- `http://hypertextbook.com/chaos/` : The Chaos Hypertextbook, by Glenn Elert.
- `http://shiny3d.de/mandelbrot-dazibao/Main/Main.htm` : The Mandelbrot Dazibao, by 'Jark'. This site gives programs written in QBasic. One of them has been the model of our Delphi program.
- `http://mrob.com/pub/muency.html` : Encyclopedia of the Mandelbrot Set, by Robert Munafo. This site describes the coloring algorithm used in our program.
- `http://en.wikipedia.org/wiki/Mandelbrot_set` : The Wikipedia page for the Mandelbrot set.
- `http://en.wikipedia.org/wiki/Julia_set` : The Wikipedia page for the Julia sets.