TPMath User Guide
Delphi version

Jean Debord

October 29, 2003

Contents

Chapter 1

Installation and compilation

This chapter explains how to install the DMath library and how to compile a program which uses it.

1.1 Installation

To install DMath, simply extract the archive `dmath.zip` in a given directory, e. g. `\dmath`. When the archive is extracted, you should have the following directory structure:

- dmath
    - demo
        - fmath
        - fourier
        - fplot
        - matrices
        - mgs
        - minfunc
        - optim
        - quadrit
        - random
        - reg
        - regnlin
    - units
        - reg

Directory `units` contains all the DMath units. Its subdirectory `reg` contains a library of predefined regression models.

Directory `demo` contains all the demo programs, distributed into several subdirectories.

There are two types of demo programs: the command-line programs, which must be executed from a DOS box in Windows, and the GUI applications, which have their own graphical interface. Only the programs which draw graphics have been built as GUI applications. All others are command-line programs. We favor such programs because the executables are very compact and the code is easier to maintain ;-)

Command-line programs are standard `*.pas` files, while the main file of any GUI application is a Delphi project (`*.dpr`) file. In addition, every GUI application is located in its own subdirectory.

1.2 Compilation from the IDE

In order to compile a program from the Delphi IDE, you must specify the path to the DMath units (e.g. `\dmath\units;\dmath\units\reg`, assuming that you have installed the library in directory `dmath`).

In the Delphi menu, select *Project*, then *Options*, and select the *Directories/Conditionals* tab of the dialog box.

The path to the units must be entered in the *Search path* section of the dialog box.

You can also specify which type of real number you will use. DMath can use any of the three real types defined in Delphi: `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`. This symbol is entered in the *Conditional defines* section of the dialog box. If no symbol is defined, then type `Double` will be automatically selected.

You can select the *Default* checkbox to make the modifications permanent.

1.3 Compilation from the command line

To compile a program `prog.pas` or `prog.dpr` from the command line, you must use a command like this in a DOS box:

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

or:

```
dcc32 prog.dpr -u\dmath\units
```

The first line is for a command-line program (as indicated by the option `-cc`). The second line is for a GUI application. The option `-u` specifies the path to the units. You can of course add other options, especially the `-d` option which defines compiler directives. For instance, to compile in extended precision, use:

```
dcc32 prog.pas -cc -u\dmath\units -dEXTENDEDREAL
```

Note that the computer must be able to locate the `dcc32.exe` file. In other words, the complete path to this file must be included in the environment variable `PATH`, which is defined in the `AUTOEXEC.BAT` file located in the root directory of the computer. This file should therefore contain a line like:

```
PATH= ... C:\PROGRA~1\BORLAND\DELPHI6\BIN; ...
```

(*or equivalent*)

If the program uses the regression units contained in the `units\reg` subdirectory, you must add the reference to the `-u` option, for instance:

```
dcc32 prog.dpr -u\dmath\units;\dmath\units\reg
```

There are two batch files named `dcompil.bat`, located in the `units` and `demo` subdirectories, which compile all units and programs in extended precision. Each file must be run from the subdirectory in which it is located.

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 the three floating point types defined in Delphi: `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. See Chapter 1 (*Installation and compilation*) for more details.

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

If a program uses one or more units, it is necessary to compile the program and its unit(s) with the same numeric precision.

Also, 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 DMath program has been compiled with the same numeric precision than the reference program.

2.2 Type Float

The unit `fmath` defines a type `Float` for real numbers. 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
  fmath;
var
  X : Float;
```

2.3 Machine-dependent constants

DMath defines 8 constants which depend on the selected numeric precision. These constants are defined in the unit `fmath`.

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 a given 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.

2.4 Demo program

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

This program displays the selected floating point type with its size in bytes. It lists 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)`
	These values should be computed without overflow.

Here are the results obtained with Delphi 6 in `Extended` mode:

```
Float type = Extended       Size = 10 bytes

MACHEP          =  1.08420217248550E-0019
MINNUM          =  3.36210314311210E-4932
Exp(MINLOG)     =  3.36210314311229E-4932
MINLOG          = -1.13551371119330E+0004
Ln(MINNUM)      = -1.13551371119330E+0004
MAXNUM          =  1.18973149535723E+4932
Exp(MAXLOG)     =  1.18973149535717E+4932
MAXLOG          =  1.13565234062941E+0004
Ln(MAXNUM)      =  1.13565234062941E+0004
MAXFAC          =  1754
Fact(MAXFAC)    =  1.97926189010501E+4930
MAXGAM          =  1.75545500000000E+0003
Gamma(MAXGAM)   =  5.92404938334683E+4931
MAXLGM          =  1.04848146839019E+4928
LnGamma(MAXLGM) =  1.18962664721039E+4932
```

Chapter 3

Elementary functions

This chapter describes the constants, types and functions available in the unit `fmath` for real variables. Functions of complex variables will be considered in the next chapter.

3.1 Constants

The following mathematical constants are defined:

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 defined internally with 20 to 21 significant digits. So, they will match the highest degree of precision available (i.e. type `Extended`).

3.2 Functional type

DMath defines a special type for a function of one real variable:

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

This type is used mainly to pass a function to a subroutine.

Note : The type `TFuncNVar` for a function of several variables is defined in the unit `matrices`.

3.3 Error handling

The global variable `MathError`, also defined in unit `fmath`, 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 MathError = FN_OK then ...
```

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

Error code	Value	Meaning
`FN_OK`	0	No error
`FN_DOMAIN`	-1	Argument domain error
`FN_SING`	-2	Function singularity
`FN_OVERFLOW`	-3	Overflow range error
`FN_UNDERFLOW`	-4	Underflow range error
`FN_TLOSS`	-5	Total loss of precision
`FN_PLOSS`	-6	Partial loss of precision

3.4 Min, max and exchange

For the following subroutines, X and Y may be 2 integers or 2 reals:

- Function `Min(X, Y)` returns the lowest number.
- Function `Max(X, Y)` returns the highest number.
- Procedure `Swap(X, Y)` exchanges the 2 numbers.

3.5 Sign

For the following functions, the arguments must be real:

- Function `Sgn(X)` returns the sign of X, i. e. 1 if $X > 0$, -1 if $X < 0$.

  The value of `Sgn(0)` is determined by the global boolean variable `SgnZeroIsOne`. If this variable is `True` (which is the default), `Sgn(0)` returns 1, otherwise it returns 0.

- Function `DSgn(A, B)` transfers the sign of B to A, so that:

  ```
  DSgn(A, B) = Sgn(B) * Abs(A)
  ```

3.6 Logarithms and exponentials

The functions `Expo` and `Log`, also defined in `fmath`, 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 necessary tests and calls the standard function if the argument is within the acceptable limits (for instance, $X > 0$ for `Log(X)`); otherwise, the function returns a default value and the variable `MathError` is set to 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 `fmath`:

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.

3.7 Power function

The function `Power(X, Y)` returns X^Y. X and Y may be integer or real, but if Y is real then X cannot be negative.

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

3.8 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)` 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`).

3.9 Trigonometric functions

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

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

3.10 Hyperbolic functions

The following functions are available:

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)`	$\ln(X + \sqrt{X^2 - 1})$	$X > 1$
`ArcTanh(X)`	$\frac{1}{2} \ln \frac{X+1}{X-1}$	$-1 < X < 1$

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

3.11 Demo programs

3.11.1 Program `testfunc.pas`

The program `testfunc.pas` located in the `demo\fmath` subdirectory 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 correspond to the numeric precision used (e. g. at least 10^{-18} in `Extended` precision).

3.11.2 Program `testw.pas`

The program `testw.pas` located in the `demo\fmath` subdirectory 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 (e. g. 18 to 19 digits in `Extended` precision).

Since the output of this program is very long, you may wish to redirect it to a file, by typing e. g.

```
testw > testw.out
```

and then open the output file in a text editor.

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

Chapter 4

Complex functions

This chapter describes the constants, types and functions available in the unit `fmath` for complex variables.

4.1 Complex type

Type `Complex` is defined as:

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

So, a complex variable Z is declared as follows:

```
uses
  fmath;
var
  Z : Complex;
```

Its real and imaginary parts are then `Z.X` and `Z.Y`

4.2 Error handling

Errors encountered while computing a complex function are handled as for the real case, i. e. the global variable `MathError` is set to the error code, and a default value is attributed to the function (see previous chapter, p. 16).

4.3 Constants

The following complex constants are defined:

Constant	Meaning
`C_infinity`	`MAXNUM`
`C_zero`	0
`C_one`	1
`C_i`	i
`C_pi`	π
`C_pi_div_2`	$\pi/2$

Note: `MAXNUM` is the highest representable number. See chapter 2, p. 12.

4.4 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 $R \cdot \exp(i\theta)$

4.5 Sign and exchange

- Function `Sgn(Z)` returns the sign of the complex Z, such that:

  $$\mathrm{Sgn}(Z) = \begin{cases} 1 & \text{if} \quad \Re(Z) > 0 \quad \text{or} \quad \Re(Z) = 0 \quad \text{and} \quad \Im(Z) > 0 \\ -1 & \text{if} \quad \Re(Z) < 0 \quad \text{or} \quad \Re(Z) = 0 \quad \text{and} \quad \Im(Z) < 0 \end{cases}$$

  where $\Re(Z)$ and $\Im(Z)$ denote the real and imaginary parts of Z, respectively.

  This function is used to determine in which half-plane (‘left’ or ‘right’) of the complex plane the number Z lies.

  The sign of 0 is set according to the global variable `SgnZeroIsOne` (See previous chapter).

- Procedure `Swap(W, Z)` exchanges the two complex numbers W and Z.

4.6 Modulus and argument

The functions `CAbs(Z)` and `CArg(Z)` give, respectively, the modulus (or absolute value) and the argument of the complex number Z, i. e. the numbers R and θ such that $Z = R \cdot \exp(i\theta)$ with $-\pi \leq \theta \leq \pi$.

If $Z = X + iY$, these functions are equivalent to `Pythag(X, Y)` and `ArcTan2(Y, X)`, respectively.

4.7 Arithmetic functions

The following functions are available (A and B are complex numbers):

Function	Meaning	Formula
`CNeg(A)`	$-A$	
`CConj(A)`	A^*	$A = X + iY \ \Rightarrow\ A^* = X - iY$
`CAdd(A, B)`	$A + B$	
`CSub(A, B)`	$A - B$	
`CMult(A, B)`	$A \times B$	
`CDiv(A, B)`	A/B	

4.8 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 $k = 0 \cdots (n-1)$

The function `CRoot(Z, K, N)` returns the K-th N-th root of the complex number Z (K and N are integers).

The function `CSqrt(Z)` returns the first square root of the complex number Z. It is therefore equivalent to `CRoot(Z, 0, 2)`.

4.9 Logarithms and exponentials

It is obvious from the relationship below that the complex logarithm is a multi-valued function:

$$Z = R \cdot \exp(i\theta) = R \cdot \exp\left[i\left(\theta + 2k\pi\right)\right] \;\Rightarrow\; \ln Z = \ln R + i\left(\theta + 2k\pi\right)$$

The function `Log(Z)` returns the *principal part* of the logarithm, i. e. the value corresponding to $k = 0$ and $-\pi \leq \theta \leq \pi$.

The function `Expo(Z)` returns the exponential of Z, according to:

$$\exp(X + iY) = e^X(\cos Y + i \sin Y)$$

4.10 Power function

The function `Power(Z, X)` returns Z^X where Z is complex and X may be integer, real or complex.

If X is integer of real, the following formula is used:

$$Z = R \cdot \exp(i\theta) \;\Rightarrow\; Z^X = R^X \cdot \exp(i\theta X)$$

If X is complex, the power function is computed by $Z^X = \exp(X \cdot \ln Z)$.

In any case, only the principal part of the function is returned. For instance, if N is integer, `Power(Z, 1/N)` is equivalent to `CRoot(Z, 0, N)`.

4.11 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)`	$\cos X \cosh Y - i \sin X \sinh Y$
`Tan(Z)`	$\frac{\sin X \cos X + i \sinh Y \cosh Y}{\cos^2 X + \sinh^2 Y} \qquad Z \neq \frac{\pi}{2} + k\pi$
`ArcSin(Z)`	$\arcsin(P - Q) + i \ \mathrm{csgn}(Y - iX) \ln(P + Q + \sqrt{(P + Q)^2 - 1})$ $P = \frac{1}{2}\sqrt{X^2 + 2X + 1 + Y^2} \quad Q = \frac{1}{2}\sqrt{X^2 - 2X + 1 + Y^2}$
`ArcCos(Z)`	$\frac{\pi}{2} - \arcsin(Z)$
`CArcTan(Z)`	$\frac{1}{2}[\arctan(X, 1 - Y) - \arctan(-X, 1 + Y)] + \frac{1}{4} i \ln \frac{X^2 + (Y+1)^2}{X^2 + (Y-1)^2} \qquad Z \neq \pm i$

4.12 Hyperbolic functions

The following functions are available (where $Z = X + iY$):

Function	Formula
`Sinh(Z)`	$\sinh X \cos Y + i \cosh X \sin Y$
`Cosh(Z)`	$\cosh X \cos Y + i \sinh X \sin Y$
`Tanh(Z)`	$\frac{\sinh X \cosh X + i \sin Y \cos Y}{\sinh^2 X + \cos^2 Y} \qquad Z \neq i\left(\frac{\pi}{2} + k\pi\right)$
`ArcSinh(Z)`	$-i \arcsin(iZ)$
`ArcCosh(Z)`	$\mathrm{csgn}[Y + i(1 - X)] \cdot i \arccos(Z)$
`ArcTanh(Z)`	$-i \arctan(iZ) \qquad Z \neq \pm 1$

4.13 Demo program

The program `testcomp.pas` located in the `demo\fmath` subdirectory checks the accuracy of the complex functions. It is a slight modification of a program written by E. Glynn (`http://www.efg2.com/Lab`). In addition to the functions defined in this chapter, the program uses the Gamma function (from unit `fspec`) and some number-formatting functions from unit `pastring`.

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, within the precision of the chosen floating type.

Chapter 5

Special functions

This chapter describes the special functions available in unit `fspec`. Most of them are Pascal translations of C codes from the Cephes library by S. Moshier (`http://www.moshier.net`). They are used mainly to compute the probability distributions, which will be discussed in the next chapter.

5.1 Factorial

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

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

To avoid unnecessary calculations, the factorials of the first 33 integers are stored in a global array named `FactArray`. So, if $N \leq 33$, it is faster to call `FactArray[N]` instead of `Fact(N)`.

The constant `MAXFAC` defines the highest integer for which the factorial can be computed. It depends on the chosen floating type (See chapter 2, p. 12).

5.2 Binomial coefficient

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

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

5.3 Gamma function

- For X real, 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. Here X may be real or complex.

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. 12).

- 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.

5.4 Beta function

- Function `Beta(X, Y)` 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)` returns the incomplete Beta function, defined by:

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

5.5 Error function

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

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

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

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

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

5.6 Demo program

The program `specfunc.pas` located in the `demo\fmath` subdirectory checks the accuracy of the special functions. The program has been adapted 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.

The program computes the special functions for a set of predefined arguments stored in the file `specfunc.dat` and compares the results to the reference values, stored in the same file. The relative differences should be within the precision range of the chosen floating point type.

Chapter 6

Probability distributions

This chapter describes the functions available in unit `fspec` to compute probability distributions.

6.1 Binomial distribution

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

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

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

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

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

  where $I_{\mathcal{B}}$ denotes the incomplete Beta function (see previous chapter).

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

6.2 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)` returns the probability of observing the value K if the mean is μ. It is defined by:

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

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

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

  where J_Γ denotes the complement of the incomplete Gamma function.

6.3 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)` 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)` returns the cumulative probability function:

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

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

- Function `PNorm(X)` returns the probability that the standard normal variable exceeds X in absolute value, i. e. $\mathrm{Prob}(|U| > X)$.
- Function `InvNorm(P)` returns the value X such that $\mathrm{Prob}(U \leq X) = P$.

6.4 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)` 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)` returns the cumulative probability function:

  $$\Phi_\nu(X) = \mathrm{Prob}(t \leq X) = \int_{-\infty}^{X} f_\nu(x)dx = 1 - I_{\mathcal{B}}\left(\frac{\nu}{2}, \frac{1}{2}, \frac{\nu}{\nu + X^2}\right)$$

  where t denotes the Student variable.

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

6.5 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)` 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)` 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)` returns the probability that the χ^2 variable exceeds X, i. e. $\text{Prob}(\chi^2 > X)$.

6.6 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)` 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)` returns the cumulative probability function:

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

  where F denotes the Snedecor variable.

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

6.7 Exponential distribution

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

- 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:

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

6.8 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)` 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)` returns the cumulative probability function:

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

6.9 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)` 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)` 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 $B = 1/2$.

Chapter 7

Matrices and linear equations

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

7.1 Using vectors and matrices

Unit `matrices` defines the following dynamic array types:

Vector type	Matrix type	Base variable
`TVector`	`TMatrix`	Floating point number (type `Float`[1])
`TIntVector`	`TIntMatrix`	Integer
`TCompVector`	`TCompMatrix`	Complex number (type `Complex`[1])
`TBoolVector`	`TBoolMatrix`	Boolean
`TStrVector`	`TStrMatrix`	String

[1] These types are defined in unit `fmath`

To use these arrays in your programs, you must:

1. Declare variables of the appropriate type, e.g.

   ```
   var
     V : TVector;
     A : TMatrix;
   ```

2. Allocate each array *before* using it:

```
DimVector(V, N);     { creates vector V[0..N] }
DimMatrix(A, N, M);  { creates matrix A[0..N, 0..M] }
                     { N, M are integer variables }
```

If the allocation succeeds, all array elements are initialized to zero (for numeric arrays), `False` (for boolean arrays), or the null string (for string arrays). Otherwise, the array is initialized to `nil`. So, it is possible to test if the allocation has succeeded:

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

The `Dim...` procedure may be used again to redimension the array.

Note that this allocation step is mandatory, because these dynamic arrays are, in fact, pointers. Unlike standard Pascal arrays, it is not sufficient to declare the variables!

3. Use arrays as in standard Pascal, noting that:
   - (a) You cannot use the assignment operator (`:=`) to copy the contents of an array into another array. Writing `B := A` simply makes `B` point to the same memory block than `A`. You must use one of the provided `Copy...` procedures described below (see section 7.3).
   - (b) All arrays begin at index 0, so that the 0-indexed element is always present, even if you don't use it.
   - (c) A matrix is declared as an array of vectors, so that `A[I]` denotes the I-th vector of matrix `A` and may be used as any vector.
   - (d) 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.
4. To deallocate an array, assign the value `nil`:

```
V := nil;
```

7.2 Programming conventions

The following conventions have been adopted for the procedures of the unit `matrices`:

- Parameters `Lbound` and `Ubound` denote the lower and upper bounds of the indices, for a vector `V[Lbound..Ubound]` or a square matrix `A[Lbound..Ubound, Lbound..Ubound]`.
- Parameters `Lbound1, Ubound1` and `Lbound2, Ubound2` denote the lower and upper bounds of the indices, for a rectangular matrix `A[Lbound1..Ubound1, Lbound2..Ubound2]`.
- 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.

7.3 Copying arrays

The following procedures are available, for vectors and matrices of the floating point type:

- procedure `SwapRows(I, K, A, Lbound, Ubound)` exchanges lines `I` and `K` of matrix `A`. Here `Lbound` and `Ubound` are the bounds in the second dimension.
- procedure `SwapCols(J, K, A, Lbound, Ubound)` exchanges columns `J` and `K` of matrix `A`. Here `Lbound` and `Ubound` are the bounds in the first dimension.
- procedure `CopyVector(Dest, Source, Lbound, Ubound)` copies vector `Source` into vector `Dest`.
- procedure `CopyMatrix(Dest, Source, Lbound1, Lbound2, Ubound1, Ubound2)` copies matrix `Source` into matrix `Dest`.
- procedure `CopyRowFromVector(Dest, Source, Lbound, Ubound, Row)` copies vector `Source` into the row `Row` of matrix `Dest`.
- procedure `CopyColFromVector(Dest, Source, Lbound, Ubound, Col)` copies vector `Source` into the column `Col` of matrix `Dest`.

- procedure `CopyVectorFromRow(Dest, Source, Lbound, Ubound, Row)` copies the row `Row` of matrix `Source` into vector `Dest`.
- procedure `CopyVectorFromCol(Dest, Source, Lbound, Ubound, Col)` copies the column `Col` of matrix `Source` into vector `Dest`.

7.4 Minima and maxima

If `X` is a real or integer vector:

- function `Min(X, Lbound, Ubound)` returns the lowest element in `X`.
- function `Max(X, Lbound, Ubound)` returns the highest element in `X`.

7.5 Matrix transposition

If `A` is a real or integer Matrix, procedure `Transpose(A, Lbound1, Lbound2, Ubound1, Ubound2, A_t)` returns its transpose in `A_t`.

7.6 Gauss-Jordan elimination

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

This procedure is implemented in DMath as the following function:

```
GaussJordan(A, B, Lbound, Ubound1, Ubound2, A_inv, X, Det)
```

where:

- `Lbound` is the lowest index in the first dimension (usually 0 or 1). It must be the same for `A` and `B`.
- `Ubound1` is the highest index in the second dimension of `A`.
- `Ubound2` is the highest index in the second dimension of `B`.
- `A_inv` is the inverse matrix.
- `Det` is the determinant.

The function returns one of two error codes:

- `MAT_OK` (or 0) if there is no error.
- `MAT_SINGUL` (or -1) if `A` is a quasi-singular matrix.

Of course, **B** may also be a vector ($m = 1$). In this case, the function simplifies to:

```
GaussJordan(A, B, Lbound, Ubound, A_inv, X, Det)
```

In case you want only the inverse and/or the determinant, there are three additional functions:

- `InvMat(A, Lbound, Ubound, A_inv)` computes the inverse matrix and returns the same error code than `GaussJordan`.
- `InvDet(A, Lbound, Ubound, A_inv, Det)` computes the inverse matrix and the determinant, and returns the same error code than `GaussJordan`.
- `Det(A, Lbound, Ubound)` returns the determinant. This function returns 0 if the matrix is quasi-singular.

Note that both `InvMat` and `Det` perform the whole Gauss-Jordan elimination on `A`. So, if you need both the inverse matrix and the determinant, it would be a waste of time to use the two functions sequentially. Use `InvDet` instead.

7.7 LU decomposition

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

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

$$\mathbf{LY} = \mathbf{B} \tag{7.1}$$

$$\mathbf{UX} = \mathbf{Y} \tag{7.2}$$

System 7.1 is solved for vector **Y**, then system 7.2 is solved for vector **X**. The solutions are simplified by the triangular nature of the matrices.

This algorithm can also be used with complex matrices.

DMath provides the following functions:

- function `LU_Decomp(A, Lbound, Ubound)` performs the LU decomposition of matrix `A`. The matrix may be real or complex.

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

  The function returns the error code `MAT_OK` or `MAT_SINGUL`

- procedure `LU_Solve(A, B, Lbound, Ubound, X)` solves the system `AX = B`, where `X` and `B` are real or complex vectors, once the matrix `A` has been transformed by `LU_Decomp`.

7.8 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 $\mathbf{Q}$ by $\mathbf{Q}'$ and left-multiplying by this transpose, one obtains:

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

or:

$$\mathbf{RX} = \mathbf{Q}'\mathbf{B}$$

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, $\mathbf{Q}$ has dimensions $n \times m$ and $\mathbf{R}$ 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 functions:

- function `QR_Decomp(A, Lbound, Ubound1, Ubound2, R)` performs the QR decomposition on the input matrix A.

  The matrix `Q` is stored in `A`, which is therefore destroyed.

  The function returns the code `MAT_OK` or `MAT_SING`.

- procedure `QR_Solve(Q, R, B, Lbound, Ubound1, Ubound2, X)` solves the system `QRX = B`.

7.9 Singular value decomposition

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

$$\mathbf{A} = \mathbf{USV}'$$

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:

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

  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 functions:

- function `SV_Decomp(A, Lbound, Ubound1, Ubound2, S, V)` performs the singular value decomposition on the input matrix A.

  The matrix `U` (such that `A = USV'`) is stored in `A`, which is therefore destroyed.

  The function returns one of the following error codes:

  - `MAT_OK` (or 0) if all goes well.
  - `MAT_NON_CONV` (or -2) if the iterative process does not converge.

- procedure `SV_SetZero(S, Lbound, Ubound, Tol)` sets to zero the singular values S_{ii} which are lower than a threshold value `Tol` which is defined by the user. This procedure must be used when solving a system with a near-singular matrix.

- procedure `SV_Solve(U, S, V, B, Lbound, Ubound1, Ubound2, X)` solves the system `USV'X = B`.

- procedure `SV_Approx(U, S, V, Lbound, Ubound1, Ubound2, A)` approximates a matrix `A` by the product `USV'`, after the lowest singular values have been set to zero by `SV_SetZero`.

7.10 Cholesky decomposition

The symmetric matrix $\mathbf{A}$ is said to be *positive definite* if, for any vector $\mathbf{x}$, the product $\mathbf{x}'\mathbf{A}\mathbf{x}$ is positive.

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

$$\mathbf{A} = \mathbf{L}\mathbf{L}'$$

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

DMath provides the function `Cholesky(A, Lbound, Ubound, L)` which performs the Cholesky decomposition on `A` and returns one of the following error codes:

- `MAT_OK` (or 0) if there is no error.

- `MAT_NOT_PD` (or -3) if `A` is not positive definite.

7.11 Demo programs

7.11.1 Determinant and inverse of a square matrix

The demo program `detinv.pas` computes the determinant and inverse of a square matrix. The matrix is stored in an ASCII file with the following structure :

- Line 1 : size of matrix (N)

- Lines 2 to (N + 1) : matrix

The default file `matrix1.dat` is an example with N = 4.

Procedure `ReadMatrix` reads the data file. Note that the matrix is dimensioned inside the procedure, hence the `var` attribute.

```
procedure ReadMatrix(FileName : String;
                     var A    : TMatrix;
                     var N    : Integer);
var
  F    : Text;     { Data file }
  I, J : Integer;  { Loop variables }
begin
  Assign(F, FileName);
  Reset(F);
  Read(F, N);
  DimMatrix(A, N, N);
  for I := 1 to N do
    for J := 1 to N do
      Read(F, A[I,J]);
  Close(F);
end;
```

The determinant and inverse are computed with the `InvDet` function, then the inverse matrix is re-inverted with the `InvMat` function.

```
{ Compute inverse matrix and determinant using the InvDet function }
if InvDet(A, 1, N, A_inv, Det) <> MAT_OK then
  begin
    WriteLn('Singular matrix!');
    Halt;
  end;

{ Write results }

{ Reinvert inverse matrix using the InvMat function }
if InvMat(A_inv, 1, N, A) = MAT_OK then
  { Write results }
```

Note : It was considered that the matrix begins at index 1. If the matrix begun at index 0, we should write `InvDet(A, 0, N, A_inv, Det)` etc.

Results obtained with file `matrix1.dat` are the following:

```
Original matrix :

   1.000000    2.000000    0.000000   -1.000000
  -1.000000    4.000000    3.000000   -0.500000
   2.000000    2.000000    1.000000   -3.000000
   0.000000    0.000000    3.000000   -4.000000

Inverse matrix :

  -1.952381    0.190476    1.571429   -0.714286
   0.761905    0.047619   -0.357143    0.071429
  -1.904762    0.380952    1.142857   -0.428571
  -1.428571    0.285714    0.857143   -0.571429

Determinant =   -21.000000

Reinverted inverse matrix :

   1.000000    2.000000    0.000000   -1.000000
  -1.000000    4.000000    3.000000   -0.500000
   2.000000    2.000000    1.000000   -3.000000
   0.000000   -0.000000    3.000000   -4.000000
```

7.11.2 Gauss-Jordan method and Hilbert matrices

The demo program `syseq.pas` tests the Gauss-Jordan method by solving a series of Hilbert systems of increasing order. Such systems have matrices of the form:

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

Each element of the constant vector **B** is equal to the sum of the terms in the corresponding line of the matrix :

$$B_i = \sum_{j=1}^{N} A_{ij}$$

The solution of such a system is:

$$\mathbf{X} = [111 \cdots 1]'$$

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 15 in `Extended` precision.

The main program has the following form:

```
{ Initialize }
N := 1;
ErrCode := 0;

{ Main loop }
while ErrCode = 0 do
  begin
    { Set system order }
    Inc(N);

    { Allocate (or re-allocate) vectors and matrices }
    DimMatrix(A, N, N);
    DimVector(B, N);
    DimMatrix(A_inv, N, N);
    DimVector(X, N);

    { Generate Hilbert system of order N }
    Hilbert(A, B, N);

    { Solve Hilbert system }
    ErrCode := GaussJordan(A, B, 1, N, A_inv, X, Det);

    { Write solution }
  end;
```

7.11.3 Gauss-Jordan, LU, QR and SVD with multiple constant vectors

The demo programs `syseq_gj.pas`, `syseqlu.pas`, `syseq_qr.pas` and `syseqsvd.pas` solve a series of linear systems with the same system matrix and several constant vectors, by using the Gauss-Jordan, LU decomposition, QR decomposition and singular value decomposition algorithms.

The data are stored in an ASCII file with the following structure:

- Line 1 : size of matrix (N)
- Lines 2 to (N + 1) : matrix
- Line N + 2 : number of constant vectors (M)
- Lines (N + 3) to (2N + 2) : constant vector (one per column)

The default file `matrix2.dat` is an example file with N = 4 and M = 5.

These data are read by the following code:

```
Assign(F, 'matrix2.dat');
Reset(F);

{ Read matrix A }
Read(F, N);
DimMatrix(A, N, N);
for I := 1 to N do
  for J := 1 to N do
    Read(F, A[I,J]);

{ Read matrix B }
Read(F, M);
DimMatrix(B, N, M);
for I := 1 to N do
  for J := 1 to M do
    Read(F, B[I,J]);
```

With `GaussJordan` the whole matrix `B` can be passed to the function:

```
{ Dimension inverse matrix and solution matrix }
DimMatrix(A_inv, N, M);
DimMatrix(X, N, M);

{ Solve system }
case GaussJordan(A, B, 1, N, M, A_inv, X, D) of
  MAT_OK     : WriteMatrix('Solution vectors', X, N, M);
  MAT_SINGUL : Write('Singular matrix!');
end;
```

With other methods, only one vector can be passed to the function. Thus we use two auxiliary vectors `B1` and `X1` which receive, respectively, the current columns of `B` and `X`:

```
DimMatrix(X, N, M);

DimVector(B1, N);
DimVector(X1, N);

{ Perform LU decomposition of A }
if LU_Decomp(A, 1, N) <> MAT_OK then
  begin
    Write('Singular matrix!');
    Halt;
  end;

{ Solve system for each constant vector }
for I := 1 to M do
  begin
    CopyVectorFromCol(B1, B, 1, N, I);
    LU_Solve(A, B1, 1, N, X1);
    CopyColFromVector(X, X1, 1, N, I);
  end;
```

With SVD, the matrix is considered singular when some singular values are lower than a given fraction of the highest one. This fraction is defined in the program by the constant `TOL`, arbitrarily set to 10^{-8}. These singular values must be set to zero before calling procedure `SV_Solve`:

```
{ Perform SV decomposition of A
  Note that U is stored in place of A }
if SV_Decomp(A, 1, N, N, S, V) <> MAT_OK then
  begin
    Write('Non-convergence of singular value decomposition!');
    Halt;
  end;

{ Set the lowest singular values to zero }
SV_SetZero(S, 1, N, TOL);

{ Solve system for each constant vector }
```

```
for I := 1 to M do
  begin
    CopyVectorFromCol(B1, B, 1, N, I);
    SV_Solve(A, S, V, B1, 1, N, N, X1);
    CopyColFromVector(X, X1, 1, N, I);
  end;
```

With the file `matrix2.dat` the four programs give the same results:

```
System matrix :

   2.000000    1.000000    5.000000   -8.000000
   7.000000    6.000000    2.000000    2.000000
  -1.000000   -3.000000  -10.000000    4.000000
   2.000000    2.000000    2.000000    1.000000

Constant vectors :

   0.000000  -15.000000   14.000000  -13.000000    5.000000
  17.000000   50.000000    1.000000   84.000000   30.000000
 -10.000000   -5.000000  -12.000000  -51.000000  -15.000000
   7.000000   17.000000    1.000000   37.000000   10.000000

Solution vectors :

   1.000000    2.000000    1.000000    4.000000    0.000000
   1.000000    5.000000   -1.000000    5.000000    5.000000
   1.000000    0.000000    1.000000    6.000000    0.000000
   1.000000    3.000000   -1.000000    7.000000    0.000000
```

7.11.4 LU decomposition with complex matrix

The demo program `syseqc.pas` solves a system of linear equations with complex coefficients by the LU method. The system is stored in a data file with the following structure:

- Line 1 : size of matrix (N)
- Lines 2 to (N + 1) : matrix, followed by constant vector

Complex numbers are given in rectangular form: real part, followed by imaginary part.

The file `matrix3.dat` is an example data file with N = 2. It corresponds to the system:

$$\begin{bmatrix} 6+5i & -6 \\ -6 & 8-4i \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 10 \\ 0 \end{bmatrix}$$

The following code reads the matrix, making use of function `Cmplx` from unit `fmath` to build the complex numbers:

```
for I := 1 to N do
  begin
  { Read line I of matrix }
  for J := 1 to N do
    begin
      Read(F, X, Y);
      A[I,J] := Cmplx(X, Y);
    end;
    { Read element I of constant vector }
    Read(F, X, Y);
    B[I] := Cmplx(X, Y);
  end;
```

Then the system is solved by calling the complex version of the LU functions:

```
{ Perform LU decomposition of A. If successful, solve system }
if LU_Decomp(A, 1, N) = 0 then
  begin
    LU_Solve(A, B, 1, N, X);
    { Write solution }
  end
else
  Write('Singular matrix!');
```

With the file `matrix3.dat` the results are:

```
System matrix :

    6.0000 +     5.0000 * i   -6.0000 +     0.0000 * i
   -6.0000 +     0.0000 * i    8.0000 -     4.0000 * i
```

```
Constant vector :

   10.0000 +     0.0000 * i
    0.0000 +     0.0000 * i

Solution vector :

    1.5000 -     2.0000 * i
    1.5000 -     0.7500 * i
```

7.11.5 Cholesky decomposition

The demo program `cholesk.pas` performs the Cholesky decomposition of a positive definite symmetric matrix. The matrix is stored in an ASCII file, as for `detinv.pas` (§ 7.11.1 p. 44). The file `matrix4.dat` is an example file with N = 3. After reading the data, the matrix is decomposed then the program computes the product **LL'** which must give the original matrix:

```
case Cholesky(A, 1, N, L) of
  MAT_OK :     { Compute LL' and write results }
  MAT_NOT_PD : { Matrix is not positive definite }
end;
```

With the file `matrix4.dat` the following results are obtained:

```
Original matrix :

   60.000000   30.000000   20.000000
   30.000000   20.000000   15.000000
   20.000000   15.000000   12.000000

Cholesky factor (L) :

    7.745967    0.000000    0.000000
    3.872983    2.236068    0.000000
    2.581989    2.236068    0.577350

Product L * L' :

   60.000000   30.000000   20.000000
   30.000000   20.000000   15.000000
   20.000000   15.000000   12.000000
```