Reference manual TRIP

version 1.4.120
24 October 2021

Gastineau - Laskar

TRIP is a general computer algebra system dedicated to celestial mechanics.

Authors :

J. Laskar
Astronomie et Systèmes Dynamiques
Institut de Mécanique Céleste
77 avenue Denfert-Rochereau
75014 PARIS
email : `laskar@imcce.fr`

M. Gastineau
Astronomie et Systèmes Dynamiques
Institut de Mécanique Céleste
77 avenue Denfert-Rochereau
75014 PARIS
email : `gastineau@imcce.fr`

With the contributions of E. Paviot, F. Thire, D. Acheroff, M. To, A. Ceyrac, G. Rouault, V. Kelsch, F. Darricau, N. Brucy, F. Boschet.

TRIP version 1.4.120

1 Introduction

In order to execute TRIP, you must
- Under Unix or MacOs X, enter trip in a shell window.
- Under Windows, click on trip icon for the console version.
- Under Windows, click on tripstudio icon for the graphic version.

You will see on screen :

```
                **** BIENVENUE SUR TRIP v1.0.0 ****

Taper 'help;' ou '?;' pour obtenir de l'aide

>
```

TRIP is ready to receive your commands. TRIP has a case sensitive interpreter.

In order to compute $S = (X + Y)^2$, you must just enter :

```
                **** BIENVENUE SUR TRIP v1.0.0 ****

Taper 'help;' ou '?;' pour obtenir de l'aide

> S=(X+Y)^2;
S(X,Y) = 1*Y**2 + 2*Y*X + 1*X**2
```

Use the up-arrow key, you could bring back the previous command. Command-line input can be edited using arrow keys.

To exit TRIP, you enter `quit` or `exit`.

The reserved keywords for TRIP language are specified in the Annexes. TRIP contains several keyword categories :
- TRIP global variables, TRIP constants and mathematic symbol.
- Functions : command returns a value
- Procedures : command doesn't return a value.

When TRIP starts or the commands reset or @@ are executed, it reads these files in the following order :

'\$TRIPDIR/etc/trip.ini'	where \$TRIPDIR is the installation directory of TRIP.
'\$HOME/.trip'	where \$HOME is the user home directory.
'trip.ini'	in the current directory.

These file may contain any valid instructions.

Recommendations :
- The following environment variables should be set and exported:
  - TMPDIR : pathname of a directory in which temporary files may be written.
  - LC_MESSAGES : language used to display messages.

2 Semantic

2.1 expression

An expression can contain all defined operators on object identifiers and can call functions or commands. An expression must be always ended with `;` or `$` . The character `;` implies the execution of the expression and displays the result. The character`$` implies only the execution of the expression. Several instructions separated with the character `;` or `$` could follow. They will be executed sequentially.

A comment on one or more lines is defined as `/* .... */`, like in the C language. A single-line comment starts with the characters `//`, like in the C++ language.

```
Example :
> // compute 1+t and display the result
> s=1+t;
s(t) =
                         1
 +                       1*t

> // compute 1+t but does not display the result
> s1=(1+x+y)^2$
> // display the content of s1
> s1;
s1(x,y) =
                         1
 +                       2*y
 +                       1*y**2
 +                       2*x
 +                       2*x*y
 +                       1*x**2

>
```

2.2 object identifier

An object identifier must begin with a letter and must contain only letters (lower-case or upper-case) and/or number (09) and the characters _ or ' .

By default, an object identifier is global, so it will be viewed by any instruction. An object identifier could be local to a macro, so it will be only viewed from this macro and it will be destroy each time the execution of this macro finished. A TRIP reserved keyword can't be used as the name of an object identifier.

The object identifier can have the following type :

- variable
- serie
- constant
- array of series
- array of variables
- numerical vector
- numerical matrix
- string of characters

– structure

```
Example :
> ch="file1"$
> s=1+x$
> dim t[1:2];
> z=1+2*i;
z = (1+i*2)
> bilan;
ch  CHAINE
s   SERIE
t   TAB
x   VAR
```

The definition of the function, subroutines and variables use the following types :

<object identifier>	object identifier
<integer>	natural integer number or an operation which returns an integer
<real>	real floating-point number or an operation which returns a real floating-point number
<complex>	complex number or an operation which returns a complex number
<variable>	variable
<constant>	an operation which returns an integer, a real number or a complex number
<serie>	serie
<operation>	an operation which returns a number (constant) or a serie
<array>	array of series
<array of variables>	array of variables
<(array of) variables>	variable or array of variables
<num. vec.>	numerical vector
<real vec.>	numerical vector of real numbers
<complex vec.>	numerical vector of complex numbers
<array of real vec.>	array of numerical vectors of real numbers
<array of complex vec.>	array of numerical vectors of complex numbers
<array of num. vec.>	array of numerical vectors
<(array of) real vec.>	(array of) numerical vectors of real numbers
<(array of) num. vec.>	(array of) numerical vectors
<constant or num. vec.>	a constant or a numerical vector
<constant or matrix>	a constant or a numerical matrix
<real or real vec.>	real floating-point number or numerical vector of real numbers
matrixtype	numerical matrix
matrixrtype	real matrix
matrixctype	complex matrix
<filename>	filename
<file>	file
<macro>	macro
<dimension of an array>	two integers separated with the character :
<list of dimension>	list of <dimension of an array> separated with a comma
<two dimensions of a matrix>	list of two dimensions separated with a comma. Each dimension consists of two integers separated with the character :

<condition>	comparison between two operations
<string>	an operation which returns a string of characters
<(array of) string>	an operation which returns a string of characters or an array of string of characters
<list of variables>	list of variables or an array of variables
<list of parameters>	list of parameters of a macro
<body>	list of trip expression
<scscp client>	connection to a SCSCP server
<remote object>	object stored on a remote SCSCP server
<Maple client>	connection to a local Maple session
<Mathematica client>	connection to a local Mathematica session

2.2.1 serie

A serie is a polynomial of degree n (n is an integer). It depends on one or more variables.

```
Example :
> s=1+x;
s(x) = 1 + 1*x
> s2=(1+x+y);
s2(x,y) = 1 + 1*y + 1*x
> bilan;
s   SERIE
s2  SERIE
x   VAR
y   VAR
```

2.2.2 constant

A constant is an integer, a real, a complex, a rational number or an interval. In rational numerical mode (`_modenum = NUMRAT` or `_modenum = NUMRATMP`), the input of 0.5 creates a floating-point number and not the rational number 1/2.

```
Example :
>x = 2;
2
>y = 2.25;
9/4
>z = 1+2*i;
(1+i*2)
>bilan;
x     CONST
y     CONST
z     CONST
> _modenum=NUMDBLINT;
            _modenum   =    NUMDBLINT
> s = <1|2>;
  s = [+1.00000000000000E+00,+2.00000000000000E+00]
```

2.2.3 string

It's a list of characters surrounded with `" "`. It's used to specify the `_path`, to define the name of the files, to display messages, ... There is no limit on the size. But, for the global variable `_path`, it's length is truncated to 256 (this value depends on the system). To produce a double-quotes (`"`) in a string, two double-quotes must be used.

```
Example :
```

```
> ch = "file" + "1";
ch = "file1"
> sch = "../" + ch;
sch = "../file1"
> sch1 = sch + "." + str(12);
sch1 = "../file1.12"
> ch3="nomsuivide""fin";
ch3 = "nomsuivide"fin"
```

2.2.4 real

<nom> = <real> ;

A real floating-point number is always displayed or inputed in base 10. The symbol `d`, `e`, `E` or `D` is the exponentiation symbol.

```
Example :
> a=2.125E6;
a = 2125000
> q=3D2;
q = 300
> r=0.123554545;
r = 0.123554545
```

2.2.5 complex

<nom> = <real> + i * <real> ;

A complex number is allowed. The lower-case i or upper-case I specifies the imaginary part of the complex number.

```
Example :
>  x=2+i*3;
x = (2+i*3)
> y=-5+4*i;
y = (-5+i*4)
```

2.2.6 filename

It's a string of characters. This string must be surrounded with `" "` if the string contains the character space " [ ] { } () or more than one points. A file name could be the content of an object identifier of type string. These file names are relative to the variable *_path*, excepted for the file names build with the function `file_fullname` (see Section 18.20 [file_fullname], page 205).

```
Example :
> read("fichier.1.dat",T);
> s="ell."+str(10)+".asc";
s = "ell.10.asc"
> read(s,T);
```

2.2.7 file

It's an object identifier which specifies a file, which is opened in reading or writing mode.

```
Example :
> f = file_open("fichier.1.dat",read);
> file_close(f);
```

2.3 Visibility

2.3.1 private

`private` [Commande]

`private` <object identifier> `x` ,...;

The specified object identifiers will be local to the macro, to the loop (for, while, sum) or to a file. The local object identifiers are destroyed at the end of the execution of the macro or at the end of each iteration of loop (for, while, sum).

If a local object identier is local to a file, this object is visible only to the macros of this file or during the execution of the file.

If a local object identifier has the same name as a global object identifier, then the local object identifier will be used. In this case, the local object identifier hides the global object identifier during the execution of the macro.

`private _ALL` [Commande]

`private _ALL;`

All object identifiers used in a macro and a file will be local to the macro or to a file.

The behavior is similar to the previous definition.

The command `public` (see Section 2.3.2 [public], page 8) allow to access to a global object.

```
Example :
> // S, SY, SXY, SXX, DEL are private objects
> macro least_ab[TX,TY,a,b] {
 private S, SY, SXY, SXX, DEL;
 S=size(TX)$
 SX=sum(TX)$
 SY=sum(TY)$
 SXY=sum(TX*TY)$
 SXX=sum(TX*TX)$
 DEL=S*SXX-SX*SX$
 a = (S*SXY-SX*SY)/DEL$
 b = (SXX*SY-SX*SXY)/DEL$
};
>
> tx=1,10;
 tx  double precision real vector : number of elements =10
> ty=3*tx;
 ty  double precision real vector : number of elements =10
> %least_ab[tx,ty,[a],[b]];
> bilan;
SX CONST
a CONST
b CONST
tx VNUMR
ty VNUMR
>
> // T2 is a private object
> macro detval[T] {
 private _ALL;
 dim T2[1:3,1:3]$
 T2[:,::]=0$
 T2[1,1]=T**2$
 T2[2,2]=1-T$
 T2[3,3]=T$
 return det(T2);
};
> %detval[5];
                      -500
> bilan;
SX CONST
a CONST
b CONST
tx VNUMR
ty VNUMR
>
```

2.3.2 public

`public` [Commande]

`public` <object identifier> `x ,...;`

This command must be only used after the command `private _ALL;`. It specifies that the identifiers of this list will be global whereas the other objects will be local to the macro or to the file.

```
Example :
> // w is local and z is global
> macro myfunc[T] {
  private _ALL;
  public z;
  w = cos(T);
  z = T;
};
> %myfunc[5];
w =          0.2836621854632262
z =                           5
> bilan;
z CONST
>
```

3 Global variables

The content of a global variable is performed by entering its name followed with `;` . The content of all global variables is displayed with the command `vartrip` (see Section 18.12 [vartrip], page 201). All global variables, containing a real number, are stored in a double-precision real number.

3.1 _affc

`integer _affc` [Variable]

`_affc` = {1, 2, 3, 4, 5, 6, 7, 8, 9};

Set the format to display the numerical coefficients of series or constants.

Default value = 4.

= 1 : %G (default short format)

= 2 : %.8G (8 digits after the decimal points)

= 3 : %.10G (10 digits after the decimal points)

= 4 : %20.15G (15 digits or more after the decimal points)

= 5 : rational approximation (double-precision real numbers are converted to rationals number using the continued fraction, else %.8G)

= 6 : same as 5 but %.15G (15 digits after the decimal points)

= 7 : rational approximation (fixed length format)

= 8 : %15.6E (6 digits after the decimal points)

= 9 : same as 8 but displays the diameter of intervals

```
Example :
> _affc=1;
    _affc = 1
> 1/3;
    0.333333
> _affc=2;
    _affc = 2
> 1/3;
    0.33333333
> _affc=3;
    _affc = 3
> 1/3;
    0.3333333333
> _affc=4;
    _affc = 4
> 1/3;
       0.333333333333333
> _affc=5;
    _affc = 5
> 1/3;
    1/3
> _affc=6;
    _affc = 6
> 1/3;
    1/3
> _affc=7;
```

```
    _affc = 7
> 1/3;
         1/3
> _affc=8;
    _affc = 8
> 1/3;
       3.333333E-01
> _affc=9; _modenum=NUMDBLINT;
    _affc = 9
    _modenum   =    NUMDBLINT
> 1/3;
    [+3.333333333333333E-1,+3.333333333333334E-1 (   5.551115E-17)]
```

3.2 _affdist

`integer _affdist` [Variable]

`_affdist` = {0, 1, 2, 3, 4, 5, 6, 7, 8};

Set the format to display the series

Default value = 2.

= 0 : recursive.

= 1 : distributed.

= 2 : distributed with one term by line.

= 3 : distributed with (co)sinus .

= 4 : distributed with (co)sinus and one term by line.

= 5 : distributed and aligned with (co)sinus and one term by line.

= 6 : distributed and factorized for variables specified in `_affvar`.

= 7 : distributed and factorized for variables specified in`_affvar`, with one term by line for the two parts.

= 8 : distributed and factorized for variables specified in`_affvar`, with one term by line for the distributed part and the factorized part is on the same line.

Remarks : if `_affvar` is empty, then the format of `_affdist` = 6, respectively 7 or 8, is the same as `_affdist`=1, respectively 2.

```
Example :
>_affdist = 2;
>x = 1+y;
x(y) =
1
+ 1*y
```

3.3 _comment

`boolean _comment` [Variable]

`_comment` {`on`,`off`};

Enable or disable the display of comments.

Default value = `off`.

```
Example :
> _comment;
```

```
_comment OFF
> /* série à deriver : /* s=1+x */  : */;
> _comment on;
> /* série à deriver : /* s=1+x */  : */;
série à deriver :  s=1+x   :
> _comment off;
```

3.4 _cpu

`integer _cpu` [Variable]

`_cpu` = <integer> ;

Specify the number of processors which will be used for later computations.

The specified value can't be greater than the number of available processors or cores on the computer.

Default value : number of active processors or cores on the computer.

```
Example :
> _mode=POLP;
_mode      =    POLP
> _cpu=1;
_cpu = 1
> time_s; s=(1+x+y+z+t+u)^30$ time_t(usertime, realtime);
03.321s  -  ( 99.44% CPU)
> msg("the real time is %g\n", realtime);
the real time is 3.33976
> _cpu=4;
_cpu = 4
> time_s; s=(1+x+y+z+t+u)^30$ time_t(usertime, realtime);
03.339s  -  (322.89% CPU)
> msg("the real time is %g\n", realtime);
the real time is 1.03414
```

3.5 _echo

`boolean _echo` [Variable]

`_echo` = {0, 1};

`_echo` {on,off};

Enable or disable the display of the echo of the executed commands.

Default value = `off`.

= 1 : Display the echo of the commands (same as on).

= 0 : Don't Display the echo of the commands (same as off)

```
Example :
>_echo = 1;
>msg "exemple";
exemple
cde>>
msg "exemple";
```

3.6 _endian

`name _endian` [Variable]

`_endian` = {little , big};

Specify the byte order (big-endian or little-endian) in binary files.

Default value = endianness in the computer.

= big : big-endian

= little : little-endian

```
Example :
> _endian=big;
                   _endian    =    big
> vnumR ieps;
> readbin(file1.dat,"%u",ieps);
```

3.7 _graph

`name _graph` [Variable]

`_graph = gnuplot`;

`_graph = grace`;

Specify if the plot, replot, plotps, plotf commands will use the grace or gnuplot (see Chapter 12 [Graphiques], page 111) software to display graphics.

Default value = `gnuplot`.

= gnuplot : gnuplot will display the graphics.

= grace : grace will display the graphics.

```
Example :
> _graph = grace;
                   _graph      = grace
> _graph = gnuplot;
                   _graph      = gnuplot
```

3.8 _hist

`boolean _hist` [Variable]

`_hist` {`on`,`off`};

Specify if the file `history.trip` is opened or not.

If the file is opened, all commands are stored in this file.

Default value = `on`.

```
Example :
>_hist;
_hist = ON
```

3.9 _history

`string _history` [Variable]

`_history` = <string> ;

Specify the folder where the file `history.trip` will be written.

Default value = `""`.

```
Example :
> _history="/users/toto/";
        _history = /users/toto/
```

3.10 _info

`booleen _info` [Variable]

`_info` {`on`,`off`};

Enable or disable the display of the message information.

Default value = `on`.

```
Example :
> _modenum=NUMQUAD;
_modenum   =    NUMQUAD
> x1=0,2*pi,2*pi/59$
> x2=0,3,0.5$
> _info off;
_info off
> sl=interpol(LINEAR,x1,cos(x1),x2)$
> _info on;
_info on
> sl=interpol(LINEAR,x1,cos(x1),x2)$
 Information :  interpol perform computations in double precision.
```

3.11 _integnum

`booleen _integnum` [Variable]

`_integnum` = {`DOPRI8`,`ODEX1`,`ADAMS`};

Defines the numerical integrator used by the numerical integration performed by `integnum` (see Section 17.4 [integnum], page 183) and `integnumfcn` (see Section 17.5 [integnumfcn], page 188).

Default value = `DOPRI8`.

= DOPRI8 : Runge-Kutta 8(7) ("A family of embedded Runge-Kutta formulae", Journal of Computational and Applied Mathematics, Dormand and Prince, 1980).

= ODEX1 : first-order extrapolation method ("Solving ordinary differential equations I", Hairer et al., 1987).

= ADAMS : Adams PECE method of order 12 ("Solving ordinary differential equations I", Hairer et al., 1987).

```
Example :
> _integnum=ODEX1;
_integnum  =    ODEX1
> _integnum=DOPRI8;
_integnum  =    DOPRI8
```

3.12 _language

`_language` [Variable]

`_language` = {fr, en };

Select the language to display messages.

Default value =

- On Unix, Linux and MacOS X, depends on the environment variable LC_MESSAGES. If it doesn't exist, the french language is used.
- On Windows, depends on the system preferences.

```
Example :
> _language = en ;
                _language    =    en
> Exp(x, y);
TRIP[ 3 ] : This parameter must be an integer
        Command :' Exp(x, y);'
                          ^

> _language = fr;
                _language    =    fr
> Exp(x, y);
TRIP[ 3 ] : Cet argument doit etre un entier
        Commande :'Exp(x, y);'
```

3.13 _mode

`_mode` [Variable]

`_mode` = {`POLY`, `POLP`, `POLYV`, `POLPV` };

Specify the internal representation of the series for their storage in the memory and during the computations.

The internal representation could be changed during the computations.

Default value = `POLY`.

- POLY : series are stored using a recursive list (recursive sparse representation). This is the default mode. The leaf nodes use generic containers. This mode is efficient for the very degree.
- POLP : series are stored using a recursive vector (recursive dense representation). The leaf nodes use generic containers. It should be used if the degree of series are low.
- POLYV : series are stored using a recursive list (recursive leaf sparse representation). The leaf nodes use optimized containers. This mode is efficient for the very degree.
- POLPV : series are stored using a recursive vector (recursive leaf dense representation). The leaf nodes use optimized containers. It should be used if the degree of series are low.

```
Example :
> _mode=POLY;
_mode      =    POLY
> s1=(1+x)^100$
> s2=1+x^100$
> bilan mem;
        Nom            Memoire (octets)         Nb de termes
-------------------- -------------------- --------------------
                  s1                 4848                  101
                  s2                   96                    2
-------------------- -------------------- --------------------
                                     4944                  103

Memoire physique utilisee = 7118848 octets
Memoire maximale utilisee = 2783817728 octets
```

```
> _mode=POLP;
_mode      =    POLP
> s1=(1+x)^100$
> s2=1+x^100$
> bilan mem;
          Nom           Memoire (octets)         Nb de termes
-------------------- -------------------- --------------------
                  s1                 3232                  101
                  s2                 3232                    2
-------------------- -------------------- --------------------
                                     6464                  103

Memoire physique utilisee = 7208960 octets
Memoire maximale utilisee = 2784210944 octets

> _mode=POLY$ time_s; s1=(1+x)^100$ time_t;
00.240s  -  ( 47.74% CPU)
> _mode=POLP$ time_s; s1=(1+x)^100$ time_t;
00.013s  -  ( 40.57% CPU)
```

3.14 _modenum

`_modenum` [Variable]

`_modenum` = {`NUMDBL`, `NUMRAT`, `NUMRATMP`, `NUMQUAD`, `NUMFPMP`};

Specify the representation of the numerical coefficients in the series and the precision of the floating point numbers in the numerical vectors or matrices.

This mode could be changed during the session.

Default value = `NUMDBL`.

The numerical coefficients are encoded as followed :

- NUMDBL : double precision floating-point. (default)
- NUMQUAD : quadruple precision floating-point. This representation isn't supported on Windows operating system.
- NUMRAT : integer or rational number, otherwise double precision floating-point.

  The absolute value for the numerator and denomiator is between 0 et 2**62 on most computers. If the numerator or denomiator becomes too large, rational number is converted in a double precision floating-point.

  The real numbers are stored in double precision format. For example, the coefficient "2.0" will be stored as a double precision floating-point while the coefficient "2" is stored as a rational number.
- NUMRATMP : multiprecision rational, otherwise double precision real.

  The absolute value for the numerator and denomiator don't have any limits.

  The real numbers are stored in double precision format. For example, the coefficient "2.0" will be stored as a double precision floating-point while the coefficient "2" is stored as a rational number.

  This numerical representation uses the GMP library (type mpz_t or mpq_t). On the 64 bit operating systems, integers smaller than 2^63-1 (in absolute value) use the hardware instructions instead of the GMP library.

– NUMFPMP : multiprecision floating-point.

  The number of number of digits of precision is specified by the global variable `_modenum_prec` (see Section 3.15 [_modenum_prec], page 16). This numerical representation uses the MPFR and the MPC library.

In the mode NUMRAT and NUMRATMP, numerical vectors or matrices contain always double precision floating-points (real or complex numbers).

```
Example :
> _affc=4;
    _affc = 4
> _modenum=NUMDBL;
        _modenum   =    NUMDBL
> s=1/11;
s =   0.0909090909090909
> _modenum=NUMRAT;
        _modenum   =    NUMRAT
> s=1/11;
s =          1/11
> _modenum=NUMFPMP;
                _modenum   =    NUMFPMP
> pi;
        3.1415926535897932384626433832795028841971693993751058209749445924
```

3.15 _modenum_prec

`integer _modenum_prec` [Variable]

`_modenum_prec` = <integer> ;

Variable used if `_modenum = NUMFPMP` (see Section 3.14 [_modenum], page 15).

Specify the number of significant decimal digits for the multiprecision real numbers.

Default value = 64.

```
Example :
> _modenum=NUMFPMP;
_modenum   =    NUMFPMP
> _modenum_prec= 20;
_modenum_prec = 20
> pi;
3.14159265358979323846
> _modenum_prec= 40;
_modenum_prec = 40
> pi;
3.1415926535897932384626433832795028841975
```

3.16 _naf_dtour

`real _naf_dtour` [Variable]

`_naf_dtour` = <real> ;

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify the "length of the dial turn ".

Default value = 2*pi.

```
Example :
> _naf_dtour=360;
```

3.17 _naf_icplx

`integer _naf_icplx` [Variable]

`_naf_icplx` = {0, 1};

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify if the function is real or complex.

Default value = 1.

The available values are :

= 0 : real function

= 1 : complex function

```
Example :
> _naf_icplx=0;
```

3.18 _naf_iprt

`integer _naf_iprt` [Variable]

`_naf_iprt` = {-1 , 0, 1, 2};

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify the verbosity to display messages in the commands `naf` and `naftab`. Intermediate results are stored in a file.

Default value = -1.

The available values are :

= -1 : No display.

= 0 : Few messages are displayed.

= 1 : More messages are displayed.

= 2 : Many messages are displayed.

```
Example :
> _naf_iprt  = 2;
```

3.19 _naf_isec

`integer _naf_isec` [Variable]

`_naf_isec` = {0, 1};

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify if the algorithm "secantes" is used or not in the commands `naf` and `naftab`.

Default value = 1.

The available values are :

= 0 : algorithm "secantes" is not used.

= 1 : algorithm "secantes" is used.

```
Example :
> _naf_isec=1;
```

3.20 _naf_iw

`integer _naf_iw` [Variable]

`_naf_iw` = <integer> ;

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify if a window filter is set.

Default value = 1.

The available values are :

- = -1 : eponential window $PHI(T) = 1/CE * \exp^{(-1/(1-T^2))}$
  avec CE= 0.22199690808403971891E0
- = 0 : no window
- = N > 0 : $PHI(T) = C_N * (1 + \cos(\pi T))^N$ avec $C_N = 2^N * (N!)^2/(2N)!$

```
Example :
> _naf_iw=-1;
```

3.21 _naf_nulin

`integer _naf_nulin` [Variable]

`_naf_nulin` = <integer> ;

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify the number of lines to be ignored at the beginning of the data file.

Default value = 1.

```
Example :
> _naf_nulin=0;
```

3.22 _naf_tol

`real _naf_tol` [Variable]

`_naf_tol` = <real> ;

Variable used by naf (see Section 17.1 [Analyse en frequence], page 177).

Specify the tolerance to check if two frencuencies are equal.

Default value = 1E-10.

```
Example :
> _naf_tol=1E-4;
```

3.23 _path

`string _path` [Variable]

`_path` = <string> ;

`_path` = "path directory";

Specify the folder used to save or load files. All TRIP commands or functions writing or reading files will prepend the filename with this path. loading trip programs (include), creating psotscript files(plotps) and plotting files (plotf) will use this path.

Default value = `""` (current folder).

Remarque :

- Don't miss the character `"/"` at the end of the path on UNIX or MACOS X.
- Don't miss the character `"\"` at the end of the path on WINDOWS.

```
Example :
_path = "/u/gram/trip/"; /* UNIX */
_path = "\u\gram\trip\"; /* WINDOWS */
```

3.24 _quiet

`boolean _quiet` [Variable]

`_quiet` {`on`,`off`};

Remove, respectively enable, the future display produced by the commands followed by `;`, only in the current macro. Default value = `off`.

```
Example :
> macro silence{ s=1; _quiet on; s=pi; };
> %silence;
s =                          1
> stat(s);
  constante has name s and has the value     3.141592653589793
```

3.25 _read

`string _read` [Variable]

`_read` = ( );

`_read` = ("skipbrokenlines", `"skipemptylines"`);

`_read` is a list of no, one or more strings which specify the behavior of the functions `read` and `file_read` when an empty or partial line is encountered.

`_read = ()` specifies that these functions will generate an error when such lines are encountered.

Les valeurs possibles sont :

- "skipbrokenlines" : the partial lines are ignored and a warning message is displayed with the indices of the ignored lines.
- "skipemptylines" : the empty lines are ignored and a warning message is displayed with the indices of the ignored lines.

Default value = `()`.

```
Example :
> t1=1,3;
 t1  double precision real vector : number of elements =3
> t2=11,13;
 t2  double precision real vector : number of elements =3
> f=file_open(temp001, write);
 f  = file "temp001"  opened in write mode
> file_writemsg(f,"# header line\n");
> file_write(f,t1);
> file_writemsg(f,"err...\n");
> file_write(f,t2);
> file_close(f);
> _read= ( "skipbrokenlines");
_read      =    ( "skipbrokenlines" )
> vnumR Q;
> read(temp001,  Q);
 Information : The line 1 is ignored : the line is incomplete or invalid
 Information : The line 5 is ignored : the line is incomplete or invalid
> writes(Q);
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
+1.1000000000000000E+01
+1.2000000000000000E+01
+1.3000000000000000E+01
```

3.26 _read_history

`string _read_history` [Variable]

`_read_history` = <string> ;

`_read_history` specifies the name of the file in which the skipped errors by *_read* are recorded. This file is emptied when this variable is assigned. Each line of this ascii file contains two columns: the line number and the name of the file where the error is produced.

`_read_history = ""` specifies that the skipped errors by *_read* are not recorded in a file.

Default value = `""`.

```
Example :
> t1=1,3;
 t1  double precision real vector : number of elements =3
> t2=11,13;
 t2  double precision real vector : number of elements =3
> f=file_open(temp001, write);
 f  = file "temp001"  opened in write mode
> file_writemsg(f,"# header line\n");
> file_write(f,t1);
> file_writemsg(f,"err...\n");
> file_write(f,t2);
> file_close(f);
> _read=( "skipbrokenlines");
_read      =    ( "skipbrokenlines" )
> _read_history="errors.dat";
_read_history = "errors.dat"
> vnumR Q;
> read(temp001,  Q);
 Information : The line 1 is ignored : the line is incomplete or invalid
 Information : The line 5 is ignored : the line is incomplete or invalid
> vnumR lerr;
> fmt="%g %s";
fmt = "%g %s"
> read("errors.dat", fmt, lerr, ferr);
> writes("%g\n",lerr);
1
5
> afftab(ferr);
ferr[1] =  "temp001"
ferr[2] =  "temp001"
```

3.27 _time

`boolean _time` [Variable]

`_time` {`on`, `off`};

enable or disable display of partial time after each executed command.

When the command `_time on` is performed, an implicit call to `time_s` and the time 0.00s is always displayed after it.

Default value = `off`.

```
Example :
> macro temps
{
s=(1+x+y+z)**20$
_time on;
s=(1+x+y+z)**20$
q=s$
_time off;
};
> %temps;
00.0s
08.10s
00.12s
>
```

3.28 _userlibrary_path

`string _userlibrary_path` [Variable]

`_userlibrary_path = `<string> ;

`_userlibrary_path = `"path to the user functions";

When this variable is assigned, TRIP reads recursively the specified directory all the file with the extension .t. It loads all the macros which are preceded by a comment starting by `//!trip extern_function` and renders them as functions to the user.

The format of this comment :

`//!trip extern_function name_of_the_function attribute="parameter_attribute" "parameter_inout"`

The string `parameter_attribute` may contain the following keywords separated by a comma. if this string is empty, then the the function is visible by the user.

- private : the function is private and is not visible by the user.

`parametres_inout` must contain the same number of elements as the number of parameter of the function. Each element are separated with a comma. An element could have the following values :

- in : the parameter is an input only.
- out : the parameter is an output only.
- inout : the parameter is an input and output.

Several macros may have the same value `name_of_the_function` if they have a different number of parameters.

Default value = `""`.

```
Example :

_userlibrary_path="./myuserlibrary/";
userfunc1(3);
userfunc2(3,y);
```

The directorty myuserlibrary contains the following file :

```
Example :
//!trip extern_function userfunc1  "in"
macro userfunc1[x]
{
    return 2*x+1;
};

 //!trip extern_function userfunc2  "in,out"
macro userfunc2[x,y]
{
    y=2*x+1;
};
```

3.29 I

`complex i` [constant]

`complex I` [constant]

$i^2 = -1$

```
Example :
>z = x + y*i;
z(x,y) = (0+i*1)*y+1*x
```

3.30 PI

`real pi` [constant]

`real PI` [constant]

The mathematical symbol π

```
Example :
>z = pi;
 3.14159265358979
```

4 Variables

4.1 crevar

`crevar` [Function]

`crevar` (<string or name> *radical*, <integer> *indice1*, <integer> *indice2*,...);

Create and return a variable with the name radical+"_"+ indice1 + ... + "_" + indicen.

`crevar` (<string or name> *radical*);

Create and return a variable with the name radical. This function allow to create variables with a non-standard name.

```
Example :
> dimvar t[1:3];
> t[1]:=crevar("t",1);
t[1] =  t_1 =                         1*t_1

> t[3]:=crevar("tab",3,2,1);
t[3] =  tab_3_2_1 =                         1*tab_3_2_1

> afftab(t);
t[1] =  t_1 =                         1*t_1

t[2] =
t[3] =  tab_3_2_1 =                         1*tab_3_2_1

> dimvar u[1:2];
> u[1]:=crevar("\bar{X}");
u[1] =  \bar{X} =                         1*\bar{X}

> u[1];
u[1] =  \bar{X} =                         1*\bar{X}

>
```

4.2 Operators

`:=` [Operator]

<object identifier> `[...] :=` <variable> ;

Create a reference to an existing variable.

```
Example :
> // w is a reference to x
> s=(1+x+y)**3$
> w:=x$
> stat(w);
Variable x type : 2     ordres :   2 2 2 2
 dependances :
 variables dependant de celle-ci :


> deriv(s,w);
```

```
                                3
   +                            6*y
   +                            3*y**2
   +                            6*x
   +                            6*x*y
   +                            3*x**2

  >
```

5 Series

5.1 Operators

`+ - * /` [Operator]

TRIP performs the addition (or unary plus), the subtraction (or unary minus), multiplication, or division between series, , variables, numbers and expressions.

To perform the addition, multiplication, subtraction and division of the arrays, the operators are +, *, - and /.

The operators +, *, -, / could be applied on numerical vectors.

The unary + or - could be applied to any type.

Remark : The division by a serie isn't available. You should use the function `div` to perform the euclidian division.

```
Example :
> s=1+x-y*z+t/2;
s(x,y,z,t) = 1 + 1/2*t - 1*y*z + 1*x
> _modenum=NUMRATMP;
_modenum   =    NUMRATMP
> p=5/3;
p = 5/3
> _modenum=NUMQUAD;
_modenum   =    NUMQUAD
> p=5/3;
p =   1.666666666666666666666666666666667
```

`** ^` [Operator]

<serie> `**`<real>

It performs the exponentiation.

It supports integer powers.

The exponentation **-1 on an array, with 2 same dimensions, performs the matrix inversion.

Remarks : The exponentiation, term by term, on arrays isn't available.

```
Example :
> u=2**3;
8
> z=(1+i)**3;
(-2+i*2)

> s=(1+x+y)**2;
s(y,x) = 1 + 2*x + 1*x**2 + 2*y + 2*y*x + 1*y**2
```

5.2 Standard functions

`size` [Function]

`size(`<serie> *s* `)`

It returns the number of terms in the serie *s*.

```
Example :
> s=1+x*y+z*t-i*p;
s(x,y,z,t,p) = 1 + (-0-i*1)*p + 1*t*z + 1*y*x
> size(s);
    4
```

5.3 Differentiation and integration

5.3.1 deriv

`deriv` [Function]

`deriv(`<serie> , <variable> `)`

It computes the total derivative of the serie with respect to one variable .

```
Example :
> s=(1+x+y)**2;
s(x,y) = 1 + 2*y + 1*y**2 + 2*x + 2*x*y + 1*x**2
> deriv(s,x);
    2 + 2*y + 2*x
```

5.3.2 integ

`integ` [Function]

`integ(`<serie> , <variable> `)`

It integrates the serie with respect to one variable .

```
Example :
>  s=(1+x+y)**2;
s(x,y) = 1 + 2*y + 1*y**2 + 2*x + 2*x*y + 1*x**2
> integ(s,x);
    1*x + 2*x*y + 1*x*y**2 + 1*x**2 + 1*x**2*y + 1/3*x**3
```

5.4 Euclidian division

`div` [Procedure]

`div(`<serie> *f*, <serie> *g*, <object identifier> *q*, <object identifier> *r* `)`

It computes the quotient and the remainder of f divided by g such that $f = q \times g + r$ and degree(r)<degree(g). The quotient is stored in *q* and the remainder in *r*.

```
Example :
> div(x^3+x+1, x^2+x+1, q,r);
 > q;
q(x) =
 -                       1
 +                       1*x

> r;
r(x) =
                         2
 +                       1*x
```

5.5 Selection

5.5.1 coef_ext

`coef_ext` [Function]

`coef_ext(`<serie> `, (`<variable> `,`<integer> `),`...)

Return the coefficient of the variable raised to the specified value from the serie.

`coef_ext`(S,(X,n),(Y,m)) returns the coefficient of X^n*Y^m in the serie S.

```
Example :
>  S= (1+x+y+z)**4 $
> coef_ext(S,(x,1),(y,2));
  12 + 12*z
```

5.6 Evaluation

5.6.1 coef_num

`coef_num` [Function]

`coef_num(<serie> , (<variable> ,<constant> ),...)`

Substitute in a serie one (or more) variable(s) by one (or more) numerical value(s).

```
Example :
> S= (1+x+y+z)**4 $
> coef_num(S,(x,0.1),(y,2));
    923521/10000 + 29791/250*z + 2883/50*z**2 + 62/5*z**3 + 1*z**4
```

5.6.2 evalnum

`evalnum` [Function]

`evalnum(<serie> , {REAL/COMPLEX} , (<variable> ,<num. vec.> ),...)`

Evaluates the serie with substituting all variables by their value in the associated numerical vectors.

The function returns a numerical vector of real numbers if `REAL` is specified or else a numerical vector of complex numbers `COMPLEX` is specified.

All numerical vectors must have the same size.

```
Example :
> serie=sin(x+y)-2*y;
serie(y,_EXy1,_EXx1) =
   (                      -0+i*                        0.5)*_EXy1**-1*_EXx1**-1
 + (                       0-i*                        0.5)*_EXy1*_EXx1
 -                        2*y

> TABX=0,pi,pi/6;
 TABX  double precision real vector : number of elements =7
> TABY=-pi,0,pi/6;
 TABY  double precision real vector : number of elements =7
> TABRES=evalnum(serie,REAL,(x,TABX),(y,TABY));
 TABRES  double precision real vector : number of elements =7
> writes(TABRES);
+6.2831853071795862E+00
+4.3699623521985504E+00
+3.3227648010019526E+00
+3.1415926535897931E+00
+2.9604205061776341E+00
+1.9132229549810371E+00
+1.2246467991473532E-16
> writes(TABX);
+0.0000000000000000E+00
+5.2359877559829882E-01
```

```
+1.0471975511965976E+00
+1.5707963267948966E+00
+2.0943951023931953E+00
+2.6179938779914940E+00
+3.1415926535897931E+00
> TABRES=evalnum(serie,COMPLEX,(x,TABX),(y,TABY));
 TABRES  Double precision complex vector : number of elements =7
> writes(TABRES);
+6.2831853071795862E+00 +1.4997597826618576E-32
+4.3699623521985504E+00 +0.0000000000000000E+00
+3.3227648010019526E+00 +0.0000000000000000E+00
+3.1415926535897931E+00 +0.0000000000000000E+00
+2.9604205061776341E+00 -5.5511151231257827E-17
+1.9132229549810371E+00 +5.5511151231257827E-17
+1.2246467991473532E-16 +0.0000000000000000E+00
>
```

6 Constants

6.1 Standard functions

Most of the functions are described in (see Chapter 10 [Vecteurs numeriques], page 55).

6.1.1 factorial

`fac` [Function]

`fac(` <integer> *n*`);`

Return *n*! (factorial function).

```
Example :
> fac(3);
                         6
> n=5;
n =                          5
> fac(n+1);
                       720
>
```

6.2 Input/output on real numbers

These functions perform the sequential writing and reading of a text file containing only real values.

`ecriture` [Procedure]

`ecriture(`<filename>` );`

Open a file in writing mode in the folder specified by `_path`. If the file doesn't exist, it will be automatically created.

```
Example : :
> ecriture("fichier1.dat");
```

`lecture` [Procedure]

`lecture(`<filename>` );`

Open a file in reading mode in the folder specified by `_path`.

```
Example : :
> lecture("fichier1.dat");
```

`print` [Procedure]

`print(`<real>` );`

Write a double-precision floating-point number in the file opened (with the command `ecriture`).

```
Example : :
> ecriture("fichier1.dat");
> print(atan(1)); /* on écrit pi/4 dans fichier1.dat */
```

`read` [Function]

`read;`

Read a floating-point number in the file opened (with the command `lecture`) and return a double-precision floating-point number.

```
Example : :
> ecriture("fichier1.dat");
> print(atan(1));
> close;
> lecture("fichier1.dat");
>  s=read;
s = 0.785398163397448
>  close;
```

`close` [Procedure]

`close;`

Close the file of real numbers if it has been opened (for reading or writing).

```
Example : :
> ecriture("fichier1.dat");
> print(atan(1));
> close;
```

7 Strings

7.1 Declaration and assignment

Character string's declaration is implicit. It will be realized when assignment are performed. To produce double-quote (") in a string, you must double it.

<nom> = <string> ;

```
Example :
/* L'exemple suivant declare les chaines ch et ch2. */
> ch="file";
ch = "file"
> ch2=ch+".txt";
ch2 = "file.txt"
```

7.2 Concatenation

<nom> = <string> + <string> ;

Operator + concatenates two character strings. Length of string isn't limited.

```
Example :
> ch = "file" + "1";
ch = "file1"
> sch = "../" + ch;
sch = "../file1"
> sch1 = sch + "." + str(12);
sch1 = "../file1.12"
```

7.3 Repetition

* [Operator]

<string> * <integer>

<integer> * <string>

Operator * takes an character string and duplicate it as many times as specified by the integer. The integer must be positive or nul. If the integer is 0, then an empty string is returned.

```
Example :
> s=" %g";
s = " %g"
> format=4*s+"\n";
format = " %g %g %g %g\n"
> t=1,10$
> writes(format, t,t**2, t**3, t**4);
```

7.4 Extraction

[] [Operator]

<string> [<integer> *j*]

Return the value of the element specified by the index *j*. The index starts at 1.

<string> [<integer> *binf* : <integer> *bsup*] ;

Return an array containing only the elements located between the lower and uper bounds with the specified step.

If the lower bound is omitted then its value is assumed to be 1. If the upper bound is omitted then its value is assumed to be the length of the string.

Remarks : any combination of omission are permitted.

```
Example :
> s="bonjour";
s = "bonjour"
> s1=s[4];
s1 = "j"
> s2=s[:3];
s2 = "bon"
> s3=s[4:5];
s3 = "jo"
>
```

7.5 Comparaison

`==` [Operator]

<string> == <string>

The Operator == returns true if the strings of characters are equal else it returns false.

```
Example :
> s="monchemin";
s = "monchemin"
> if (s=="monchemin") then { msg "true"; } else { msg "false"; };
true
>
```

`!=` [Operator]

<string> != <string>

The Operator != returns false if the strings of characters are equal else it returns true.

```
Example :
> s="monchemin";
s = "monchemin"
> if (s=="MON") then { msg "true"; } else { msg "false"; };
false
>
```

`<` [Operator]

<string> *s1* < <string> *s2*

The Operator < returns true if *s1* is less than *s2* using the lexicographic order else it returns false.

```
Example :
> s="abc";
s = "abc"
> if (s<"abd") then { msg "true"; } else { msg "false"; };
true
>
```

`<=` [Operator]

<string> *s1* <= <string> *s2*

The Operator <= returns true if *s1* is less than or equal to *s2* using the lexicographic order else it returns false.

```
Example :
> s="abc";
s = "abc"
> if (s<="abe") then { msg "true"; } else { msg "false"; };
true
>
```

`>` [Operator]

<string> *s1* > <string> *s2*

The Operator > returns true if *s1* is greater than *s2* using the lexicographic order else it returns false.

```
Example :
> s="abc";
s = "abc"
> if (s>"abd") then { msg "true"; } else { msg "false"; };
false
>
```

`>=` [Operator]

<string> *s1* >= <string> *s2*

The Operator >= returns true if *s1* is greater than or equal to *s2* using the lexicographic order else it returns false.

```
Example :
> s="abc";
s = "abc"
> if (s>="abe") then { msg "true"; } else { msg "false"; };
false
>
```

7.6 Conversion from an integer, a real or a serie to a string

`str` [Function]

`str`(<integer>);

`str`(<real>);

`str`(<serie>);

`str`(<string> *format*, <real>);

Convert an integer, a real number or an object in a character string. If *format* is specified, then the integer or real number is converted to this format. The format specification is similar to that for the printf function of the C-language.

The conversion specifiers are

– g,G The number is converted as a real (style [-]dddd.dddd or scientific style [-]d.dddE+-dd if exponent is too large).
– e,E The number is converted as a real (scientific style [-]d.dddE+-dd).
– f,F The number is converted as a real (style [-]d.dddE+-dd).
– d,i The number is converted as an integer. If the argument is a real number, only its integer part is converted. If the numerical mode NUMRAT or NUMRATMP is selected, then the rational numbers as written as "numerator/denominator".

The length modifier aren't supported. For example, the format "%lg" will be rejected and produce an error.

If the numerical mode NUMRAT or NUMRATMP is selected, then the rational numbers as written as "numerator/denominator" if no format are specified.

```
Example :
> ch = str(1235);
ch = "1235"
> ch = str(int(2*pi));
ch = "6"
> s = str(2E3);
s = "2000"
> s = str("%.4g",pi);
s = "3.142"
> _modenum = NUMRATMP;
_modenum   =    NUMRATMP
> s = str("%d", 3/2);
s = "3/2"
> s = str("%g", 3/2);
s = "1.5"
> s = str(1+x**2);
s = "   1
 + 1*x**2

"
>
```

7.7 Conversion from a list of integers or reals to a string

`msg` [Function]

`msg(`<string> *textformat*`,` <real> *x*`,` ... `);`

Create a string using the format with the real constants.

The format is the same as the command printf in language C (see Section 7.6 [str], page 33, for the valid conversion specifiers). This format must be a string and could be on several lines.

To create a double-quotes, two double-quotes must be used.

```
Example :
> ch = msg("pi=%g pi=%.8E",pi,pi);
ch = "pi=3.14159 pi=3.14159265E+00"
> _modenum=NUMRATMP;
_modenum   =    NUMRATMP
> s=msg("%d %g", 1/11, 1/11);
s = "1/11 0.0909091"
```

7.8 Conversion from a string to an integer or a real

`coef_num` [Function]

`coef_num(` <string> `);`

It converts a character string to an integer or a real number.

```
Example :
```

```
> d = coef_num("-42");
d =                       -42
> r = coef_num("3.14E0");
r =                      3.14
>
```

7.9 Length of a string

`size` [Function]

`size`(<string>);

it computes the length of the string, that is to compute the number of characters.

```
Example :
> ch = "1235";
ch = "1235"
> size(ch);
 4
```

8 Arrays

TRIP has 4 types of arrays :
- arrays of series and constants.
- array of variables.
- numerical vectors.
- numerical matrices.

Matrix are arrays of series and constants. For the numerical vectors or matrices, see the appropriate chapter.

8.1 Declaration of array of series

`dim` [Procedure]

`dim` <name> [<list of dimension>], ...;

Define one or more array of series with the specified dimensions.

Each dimension is separated with a comma. A dimension is defined with two integers, separated by the character : which specifiy the index of the first element, the index of the last element, and the step for this dimension. If the third integer is missing, the step is 1. The dimensions may stored in an object identifier.

This type of array can contain series, constants, truncatures, strings of characters... but no variables.

```
Example :
> // It defines the array tl of series or numbers
> // with two dimensions
> dim tl [1:22,-2:6];
> // it defines 3 arrays t1, t2 , t3 with different bounds
> bounds1 = 1:4;
bounds1 = bounds [ 1:4 ]
> dim t1[bounds1], t2[5:6], t3[-1:3];
> // it defines an array with a step not equal to 1
> dim t5[1:5:2];
> t5[:]=7;
> afftab(t5);
t5[1] =                     7
t5[3] =                     7
t5[5] =                     7
>
```

8.2 Initialization of an array of series

<name> = [<serie> ,... : <serie> , ...];

<name> = dim[<serie> ,... : <serie> , ...];

Create and intialize an array of series with the specified series and constants.

The symbol : specifies a new line and the symbol , separates the colmuns. It must have the same number of columns on each line.

```
Example :
> // define a 2-dimensional array which contains polynomials
> tab=[1,2+2*x:(1+x)**2,(2+2*x)**2];
```

```
tab [1:2, 1:2 ] number of elements = 4

> stat(tab);
  Array of series
 tab [ 1:2 , 1:2  ]
 list of array's elements :
tab [ 1 , 1 ] =
  constante has name tab and has the value     1
tab [ 1 , 2 ] =
 serie tab (  x )
 number of variables : 1   size of the descriptor : 80 bytes
 number of terms : 2   size : 304 bytes
tab [ 2 , 1 ] =
 serie tab (  x )
 number of variables : 1   size of the descriptor : 80 bytes
 number of terms : 3   size : 352 bytes
tab [ 2 , 2 ] =
 serie tab (  x )
 number of variables : 1   size of the descriptor : 80 bytes
 number of terms : 3   size : 352 bytes
> // define a 2-dimensional array which contains numbers
> tab2=[1,2,3:4,5,6];

tab2 [1:2, 1:3 ] number of elements = 6

>
```

8.3 Declaration of array of variables

`dimvar` [Procedure]

`dimvar` <name> [<list of dimension>], ...;

Define one or more array of variables with the specified dimensions.

Each dimension is separated with a comma. A dimension is defined with two integers which specifiy the index of the first element and the index of the last element for this dimension.

To assign a existing variable to an array of variables, you should the symbol `:=` instead of `=` .

```
Example :
> // define the array of variables t2 with one dimension
> dimvar t2[1:5];
> t2[1] := x;
t2[1] =  x =                         1*x

> // we obtain the derivative of S by x.
> S=1+3*x$
> deriv(S,t2[1]);
                         3
>
> // we define 2 arrays of variables tv1 and tv2
> dimvar tv1[1:2], tv2[1:3];
>
```

8.4 Initialization of an array of variables

<name> = dimvar[<serie> ,... : <serie> , ...];

Create and intialize an array of variables with the specified variables.

The symbol **:** specifies a new line and the symbol **,** separates the colmuns. It must have the same number of columns on each line.

```
Example :
> // define the array of variables t2 with the variables x, y, z and u
> 1+x+y+z$
> t2 = dimvar[x:y:z:u];

t2 [1:4 ] number of elements = 4

>
```

8.5 Generation of an array of variables

`tabvar` [Procedure]

`tabvar`(<array of variables>);

Generate automatically the variables inside the specified array. The name of variables use the name of the array as radical and its indice.

```
Example :
> dimvar t[0:3];
> tabvar(t);
> afftab(t);
t[0] =  t_0 =                         1*t_0

t[1] =  t_1 =                         1*t_1

t[2] =  t_2 =                         1*t_2

t[3] =  t_3 =                         1*t_3

>
```

8.6 Assignment to an array

`=` [Operator]

<name> `=` <array> ;

Assign the array (operation between arrays) to the specified object identifier.

<array> `[<integer> , ...]` `=` <operation> ;

Assign a value to an element of an array. This operation can be a serie, a constant, a numerical vector, a truncature but not an array.

<array> `[` <integer> *binf* `:` <integer> *bsup* `:` <integer> *pas* `,...]=` <operation> ;

<array> `[` <integer> *binf* `:` <integer> *bsup* `,...]=` <operation> ;

Assign a value to a part of an array.

If the operation is a serie, a constant, a numerical vector, a truncature then the value is copied to each element of this part of array. If the operation is an array then each element of this array will be assigned to the corresponding element. In this case, it verifies that the number of elements and dimension is compatible.

If the lower bound is omitted then its value is assumed to be 1. If the upper bound is omitted then its value is assumed to be the size of the array. If the step is omitted then its value is assumed to be 1.

When the number of dimension is unknown or variable, the variadic statement ... could be provided as the last argument in order to specify an undefined number of :,:,:.

Remarks : any combination of omission are permitted.

```
Example :
> _affc=1$
> dim t[1:4,7:25];
> t[1,7]=1+x;
t[1,7] =    1
 + 1*x

> r=t;

r [1:4, 7:25 ] number of elements = 76

> dim t[1:4,7:25];
> dim t2[1:4];
> t2[:]=5;
> t[:,8]=t2;
> t[:,::2]=1+x;
> t[2,...]=3;
>
```

`:=` [Operator]

<array> `[...] :=` <variable> ;

Assign a variable to an array of variables.

```
Example :
> dimvar X[1:4];
> X[1]:=crevar("e",1,1);
X[1] =  e_1_1 =                         1*e_1_1

> deriv(1+2*e_1_1,X[1]);
                               2
>
```

8.7 Size of an array

`size` [Function]

`size`(<array>);

`size`(<array> ,<integer>);

Return the number of elements in the first dimension of the array or in the specified dimension by the integer.

If the specified dimension doesn't exist in the array then the function returns -1.

```
Example :
> dim t[1:4,7:25];
> size(t,2);
 19
```

```
> size(t);
 4
>
```

`num_dim` [Function]

`num_dim`(<array> *t*)

It returns the number of dimensions of the array *t*.

```
Example :
> dim t1[1:5];
> size(t1);
 5
> dim t2[1:22,-2:6];
> size(t2);
 22
>
```

8.8 bounds of an array

`inf` [Function]

`inf`(<array> ,<integer>);

Return the lower bound of the array of the specified dimension by the integer.

If the specified dimension doesn't exist in the array then the function returns -1.

```
Example :
> dim t[1:2,0:3];
> inf(t,2);
 0
> for k=inf(t,2) to sup(t,2) { t[:,k]= k$ };
> afftab(t);
t[1,0] =                     0
t[1,1] =                        1
t[1,2] =                        2
t[1,3] =                        3
t[2,0] =                     0
t[2,1] =                        1
t[2,2] =                        2
t[2,3] =                        3
>
```

`sup` [Function]

`sup`(<array> ,<integer>);

Return the upper bound of the array of the specified dimension by the integer.

If the specified dimension doesn't exist in the array then the function returns -1.

```
Example :
> dim t[1:2,0:3];
> sup(t,2);
 3
> for k=inf(t,1) to sup(t,1) { t[k,:]= k$ };
> afftab(t);
t[1,0] =                        1
t[1,1] =                        1
```

```
t[1,2] =                         1
t[1,3] =                         1
t[2,0] =                         2
t[2,1] =                         2
t[2,2] =                         2
t[2,3] =                         2
>
```

8.9 Display an array

`afftab` [Procedure]

`afftab`(<array>);

Display the content of the arrays of series or variables.

```
Example :
> _affc=1$
> dim t[1:4];
> t[1]=1+x$
> t[2]=2*i$
> t[3]=({(x,y),2})$
> afftab(t);
t[1] =    1
 + 1*x

t[2] =    (0+i*2)
t[3] = ({(y, x), 2})
t[4] =
>
```

8.10 Extract an element

`[]` [Operator]

<array> `[`<integer> ,...`]`;

Return the value of the element specified by the indexes.

<num. vec.> `[` <integer> *binf* `:` <integer> *bsup* `:` <integer> *pas*,... `]`;

<num. vec.> `[` <integer> *binf* `:` <integer> *bsup*, ... `]`;

Return an array containing only the elements located between the lower and uper bounds with the specified step.

If the lower bound is omitted then its value is assumed to be the lower bound of that dimension of this array. If the upper bound is omitted then its value is assumed to be the upper bound of that dimension of this array. If the step is omitted then its value is assumed to be 1.

When the number of dimension is unknown or variable, the variadic statement ... could be provided as the last argument in order to specify an undefined number of :,:,:.

Remarks : any combination of omission are permitted.

```
Example :
> _affc=1$
> dim t[1:4];
> t[1]=1+x$
> s=t[1];
s(x) =
   1
 + 1*x

> t1=t[3:];

t1 [3:4 ] number of elements = 2

> dim t[1:4,5:10];
> v2=t[:,6:];

v2 [1:4, 6:10 ] number of elements = 20

> v3=t[4,...];

v3 [5:10 ] number of elements = 6
```

`select` [Function]

`select ( <condition> , <array> );`

Return an array which only contains the elements of the array where the condition is true.

If the condition is always false, the function returns the consnt number 0. To test if the consant value 0 was returned, the following code could be used : `if (size(result)==0) then { ... } else { ... }`

The condition and the array must have the same number of elements and only one diemnsion size.

```
Example :
> // extract from tm the elmeents such that t1[j]=1
> _modenum=NUMRATMP;
_modenum   =    NUMRATMP
> dim t1[1:5], tm[1:5]$
> for j=1 to 5 { t1[j]=abs(3-j)$ tm[j]=(1+x)**j$ };
> q = select(t1==1, tm);

q [1:2 ] number of elements = 2

> if (size(q)!=0) then { afftab(q); };
q[1] =    1
 + 2*x
 + 1*x**2

q[2] =    1
 + 4*x
 + 6*x**2
 + 4*x**3
 + 1*x**4

>
```

8.11 Operations on array of series

Most of the functions are described in (see Chapter 10 [Vecteurs numeriques], page 55). The following functions could be applied on the arrays of series (as on the series) to avoid the loops for :

- coef_ext see Section 5.5.1 [coef_ext], page 26,
- coef_num see Section 5.6.1 [coef_num], page 27,
- deriv see Section 5.3.1 [deriv], page 26,
- integ see Section 5.3.2 [integ], page 26,

8.11.1 Matrix product

`&*` [Operator]

<array> `&*` <array>

Compute the matrix product of the arrays (with 1 or 2 dimensions).

Compute r=a&*b such that $r[i,j] = \sum_{k=1}^{size(a,1)} a_{i,k} \times b_{k,j}$

If the array has only one dimension then it's considered as a column-vector.

Remark : the number of columns of the first array must be same as the number of lines of the second array.

```
Example :
> _affc=1$ _affdist=0$
> t=[x,y:x-y,x+y]$
> t2=[x+y,x-y:x,y]$
> r=t&*t2;

r [1:2, 1:2 ] number of elements = 4

> afftab(r);
r[1,1] =    2*y*x + x**2
r[1,2] =    y**2 - y*x + x**2
r[2,1] =  - y**2 + y*x + 2*x**2
r[2,2] =    2*y**2 - y*x + x**2
>
```

8.11.2 Matrix determinant

`det` [Function]

`det(` <array> *M* `)`

`det(` <array> *M*`,` <string> *method* `)`

Computes the determinant of a matrix. The array must have two dimensions.

- If *M* is a matrix with numerical entries, the LU algorithm is used in all numerical precision. When computations are performed in double precision, the Lapack Library is used.
- If *M* is a matrix with polynomial entries, un algorithme de Bareiss est utilisé par défaut.

If *method* is specified, it uses that algorithm to compute the determinant if the content of the matrix allows it. In other case, the determinant is computed using the default algorithm.

The available values for *method* are :

- `"minor"` : Determinant expansion by minors using large memory. In pratice, it could be used only for the matrices'size less than 12.

- "bareiss" : Gauss-Bareiss algorithm.
- "interpol" : Interpolation algorithm. It could be only used on the matrices with polynomial entries. Their coefficients must be integer or rational numbers.
- "interpol_modular" : Interpolation algorithm using the modular arithmetic and the chinese remainder. It could be only used on the matrices with polynomial entries. Their coefficients must be integer numbers.

```
Example :
> t=[9,0,7
    :1,2,3
    :4,5,6];

t [1:3, 1:3 ] number of elements = 9

> det(t);
                       -48
>
```

8.11.3 Matrix inversion

**-1 [Function]

<array> **-1

Computes the inverse of a matrix. The array must have two dimensions.

Remarks : When computations are performed in double precision, the Lapack Library is used. The LU algorithm is used in all numerical precision.

```
Example :
> _modenum=NUMRATMP$
> t=[9,0,7
    :1,2,3
    :4,5,6];

t [1:3, 1:3 ] number of elements = 9

> r=t**-1;

r [1:3, 1:3 ] number of elements = 9

> afftab(r);
r[1,1] =    1/16
r[1,2] =  - 35/48
r[1,3] =    7/24
r[2,1] =  - 1/8
r[2,2] =  - 13/24
r[2,3] =    5/12
r[3,1] =    1/16
r[3,2] =    15/16
r[3,3] =  - 3/8
>
```

8.11.4 Eigenvalues

eigenvalues [Function]

eigenvalues(<array>)

Compute the eigenvalues of the 2-dimensional array (square matrix) using a QR algorithm (lapack library).

```
Example :
> t=[9,0,7
    :1,2,3
    :4,5,6];

t [1:3, 1:3 ] number of elements = 9

> l=eigenvalues(t);

l [1:3 ] number of elements = 3

> afftab(l);
l[1] =          13.76915041895731
l[2] =          4.084362034809442
l[3] =        -0.8535124537667539
```

8.11.5 Eigenvectors

`eigenvectors` [Procedure]

`eigenvectors`(<array> *MAT*, <array> *TVECT*, <array> *TVAL*)

`eigenvectors`(<array> *MAT*, <array> *TVECT*)

Compute the eigenvectors of the 2-dimensional array *MAT* (square matrix) using a QR algorithm (lapack library). It stores the eigenvectors in the array *TVECT* and the eigenvalues in the array *TVAL*.

```
Example :
> t=[9,0,7
    :1,2,3
    :4,5,6];

t [1:3, 1:3 ] number of elements = 9

> eigenvectors(t,vectp,valp);
> afftab(vectp);
vectp[1,1] =        -0.8081690381494434
vectp[1,2] =        -0.7505769919446202
vectp[1,3] =        -0.4846638559537327
vectp[2,1] =         -0.209021306608322
vectp[2,2] =         0.3985224414326275
vectp[2,3] =        -0.5474094124384613
vectp[3,1] =         -0.550611386696964
vectp[3,2] =         0.5270806796287867
vectp[3,3] =         0.6822344772186747
> afftab(valp);
valp[1] =          13.76915041895731
valp[2] =          4.084362034809442
valp[3] =        -0.8535124537667539
> //  first vector
> p=vectp[:,1]$
> afftab(p);
```

```
p[1] =        -0.8081690381494434
p[2] =         -0.209021306608322
p[3] =         -0.550611386696964
```

8.11.6 Arithmetic

The arrays must have the same number of elements on each dimension and the number of dimension must be the same.

`+` [Operator]

<array> + <array>

Add term by term two arrays of series.

`+` [Operator]

<array> + <serie>

<serie> + <array>

Add a serie with each element of the array of series.

`*` [Operator]

<array> * <array>

Multiply term by term two arrays of series.

`*` [Operator]

<array> * <serie>

<serie> * <array>

Multiply a serie with each element of the array of series.

`-` [Operator]

<array> - <array>

Substract term by term two arrays of series.

`-` [Operator]

<array> - <serie>

<serie> - <array>

Subtract a serie with each element of the array of series.

`/` [Operator]

<array> / <array>

Divide term by term two arrays of series.

`/` [Operator]

<array> / <serie>

<serie> / <array>

Divide a serie with each element of the array of series.

```
Example :
> _affc=1$
> t2=[x+y,x-y:x,y];

t2 [1:2, 1:2 ] number of elements = 4

> t=[x,y:x-y,x+y];
```

```
t [1:2, 1:2 ] number of elements = 4

> r=t*t2*i-t*x;

r [1:2, 1:2 ] number of elements = 4

> afftab(r);
r[1,1] =    (0+i*1)*x*y
 + (-1+i*1)*x**2

r[1,2] =    (-0-i*1)*y**2
 + (-1+i*1)*x*y

r[2,1] =    (1-i*1)*x*y
 + (-1+i*1)*x**2

r[2,2] =    (0+i*1)*y**2
 + (-1+i*1)*x*y
 - 1*x**2

>
```

8.12 Conversion

For the convertion to numerical vectors, (see Section 10.12 [Conversion], page 94).

9 Structures and OOP

The data structures contain other values indexed by their names. These fields may have a global or private visibility. The macros can be associated to these data structures. These macros may access to all the fields.

9.1 Type declaration

`struct` [Procedure]

`struct` <structtype> { <list of parameters> ; `private` <list of parameters> ; };

`struct` <structtype> { <object identifier> = <value> ; ... `private` <object identifier> = <value> ; `private` ... };

It defines a new type of records whose fields of the first list can be accessed globally and the other list which follows the keyword `private` can only be accessed by an associated macro.

The fields my be initialized to a defaut value.

```
Example :
> // declaration of the type POINT
> struct POINT {
  x, y;
};
>
> // declaration of the type MyData
> struct MyData {
  x = 0;
  y = 0;
  z = -1;
  vnumR v([1:3]);
  dim armap[1:5];
  private name = "NONAME";
  private file = "/tmp/myfile";
};
> MyData a;
> afftab(a);
a@x =                       0
a@y =                       0
a@z =                         -1
a@v =  a  double precision real vector : number of elements =3
a@armap =
a [1:5 ] number of elements = 5

a@name =  "NONAME"
a@file =  "/tmp/myfile"
```

9.2 Object identifier declaration

`struct` [Procedure]

`struct` <structtype> <object identifier> *p1*, <object identifier> *p2*, ... ;

It creates the object identifiers whose the name are given in the list. These objects are data structure of the specified type.

```
Example :
> // declaration of the type POINT
> struct POINT {
  x, y;
};
> // declaration of the object of type POINT
> struct POINT p1, p2, p3;
> p1;
p1 = structure POINT
> afftab(p1);
p1@x = /* UNINITIALIZED */
p1@y = /* UNINITIALIZED */
```

9.3 Access to member attributes

@ [Operator]

<structure> @<name>

It allows to access to the value of a field of the data structure. If this field is private, this field can only be accessed from a macro associated to this structure.

```
Example :
> struct MyData {
  x,y,z;
  name, file;
};
> struct MyData s;
>
> s@x = 3;
s@x =                          3
> s@y = 1;
s@y =                          1
> s@z = s@x + 2*s@y;
s@z =                          5
> s@name = "temp001";
s@name =  "temp001"
> s@file=file_open(s@name,write);
s@file =   file "temp001"  opened in write mode
>
```

9.4 Display

afftab [Procedure]

afftab(<structure> *x*);

It displays the value of all the fields of the data structure *x*.

```
Example :
> struct MyData {
  x,y,z;
  name, file;
};
> struct MyData s;
>
```

```
> s@x = 3$
> s@y = 1$
> s@z = s@x + 2*s@y$
> s@name = "temp001"$
> s@file=file_open(s@name,write)$
> afftab(s);
s@x =                          3
s@y =                          1
s@z =                          5
s@name =  "temp001"
s@file =   file "temp001"  opened in write mode
>
```

9.5 Constructor macros

`macro` [Procedure]

`macro` <structtype> @ <structtype> { <body> };

It defines a constructor of the data structure with no parameter and a body of trip code. This macro is implicitly called when an object is initialized.

The macro allows to access to all the private or public fields of the data structure. A direct access to these fields is allowed and does not require to be prefixed by `@`.

The object *self* is already defined during the execution of the macro. It refers to the value of the data structure which responsible of the call. This object is useful for a global assignment or to call another macro with this value.

```
Example :
> struct MyData {
  t;
  vnumR v([1:3])$
};
> macro  MyData@MyData {
  private _ALL;
  t = log(2)$
  v[:]=4$
};
>
> MyData a;
> afftab(a);
a@t =         0.6931471805599453
a@v =  a  double precision real vector : number of elements =3
>
>
```

9.6 Member macros

9.6.1 Declaration

`macro` [Procedure]

`macro` <structtype> @ <name> [<list of parameters>] { <body> };

`macro` <structtype> @ <name> { <body> };

`private macro` <structure> @ <name> [<list of parameters>] { <body> };

`private macro` <structure> @ <name> { <body> };

It defines a member macro of the data structure with 0 or more parameter and a body of trip code.

The macro allows to access to all the private or public fields of the data structure. A direct access to these fields is allowed and does not require to be prefixed by `@`.

The object *self* is already defined during the execution of the macro. It refers to the value of the data structure which responsible of the call. This object is useful for a global assignment or to call another macro with this value.

If the macro is defined as private, then this macro can be called only other macros associated to the same data structure.

```
Example :
> struct MyData {
  x,y,z;
  private name, file;
};
>
> // Define Init as a macro of MyData
> macro MyData@Init[vx,vy,vz,vname] {
    x = vx$
    y = vy$
    z = vz$
    name = vname$
    %OpenFile$
    afftab(self);
};
>
> // Define OpenFile as a private macro of MyData
> private macro MyData@OpenFile {
    file = file_open(name,write)$
};
>
>
```

9.6.2 Execution

`%` [opérateur]

<structure> `%` <name> [<list of parameters>];

<structure> `%` <name> ;

It calls a member macro of the data structure and defines the object *self* as the value of the data structure during its execution.

```
Example :
> struct MyData {
  x,y,z;
  private name, file;
};
>
> macro MyData@Init[vx,vy,vz,vname] {
    x = vx$
    y = vy$
    z = vz$
    name = vname$
    %OpenFile$
    afftab(self);
};
>
> macro MyData@Clear {
    %CloseFile;
};
>
> private macro MyData@OpenFile {
    file = file_open(name,write)$
};
>
> private macro MyData@CloseFile {
    file_close(file);
};
>
> struct MyData s;
> // execute the macro Init
> s%Init[1,2,3,"temp001"];
s@x =                          1
s@y =                          2
s@z =                          3
s@name =  "temp001"
s@file =   file "temp001"  opened in write mode
>
> // execute the macro Clear
> s%Clear;
>
```

10 Numerical vectors

The numerical values stored in vectors are always real or complex double-precision, quadruple-precision or multiprecision numbers depending on the current numerical mode. A numerical vector is assumed to be a column vector.

In the numerical mode NUMRATMP, the rational numbers may be stored in the numerical vector of rational numbers.

10.1 Declaration

Explicit declaration for numerical vectors are only required before call to `read` , `readbin` (see Section 10.8 [Entree/SortieTabNum], page 63) and `resize` . Explicit declaration for array of numerical vectors are always required.

10.1.1 vnumR

`vnumR` [Procedure]

`vnumR` <name> `,` ... `;`

`vnumR` <name> `[` <dimension of an array> `]` `,` ... `;`

It declares a real vector or an array of real vectors.

The real vector's size is dynamic. After the declaration, the real vectors are empty. To set their sizes, the command `resize` must be used.

The dimension specifies the number of elements in the array of real vectors.

`vnumR` [Procedure]

`vnumR` <name> `([` 1: <integer> *n* `])` `,` ... `;`

`vnumR` <name> `[` <dimension of an array> `]` `([` 1: <integer> *n* `])` `,` ... `;`

It declares a vector of *n* reals or an array of vectors of *n* reals.

After the declaration, the elements of the vector are initialized with the value 0.

```
Example :
> vnumR A, C([1:5]);
> stat(A);
 Numerical vector A contains 0 double precision reals.
> stat(C);
 Numerical vector C contains 5 double precision reals.
  Size of the array in bytes: 40
> bounds=1:2;
bounds = bounds [ 1:2 ]
> vnumR TE[bounds], TO[bounds]([1:6]);
> stat(TE);
  Array of series
 TE [ 1:2  ]
 list of array's elements :
TE [ 1 ] =
 Numerical vector TE contains 0 double precision reals.
TE [ 2 ] =
 Numerical vector TE contains 0 double precision reals.
> stat(TO);
  Array of series
 TO [ 1:2  ]
 list of array's elements :
```

```
TO [ 1 ] =
 Numerical vector TO contains 6 double precision reals.
  Size of the array in bytes: 48
TO [ 2 ] =
 Numerical vector TO contains 6 double precision reals.
  Size of the array in bytes: 48
>
```

10.1.2 vnumC

`vnumC` [Procedure]

`vnumC` <name> `, ... ;`

`vnumC` <name> `[` <dimension of an array> `] , ... ;`

It declares a complex vector or an array of complex vectors.

The complex vector's size is dynamic. After the declaration, the complex vectors are empty. To set their sizes, the command `resize` must be used.

The dimension specifies the number of elements in the array of complex vectors.

`vnumC` [Procedure]

`vnumC` <name> `([` 1: <integer> *n* `]) , ... ;`

`vnumC` <name> `[` <dimension of an array> `] ([` 1: <integer> *n* `]) , ... ;`

It declares a vector of *n* complex numbers or an array of vectors of *n* complex numbers.

After the declaration, the elements of the vector are initialized with the value 0+i*0.

```
Example :
> vnumC A, C([1:5]);
> stat(A);
 Numerical vector A contains 0 double precision complex.
> stat(C);
 Numerical vector C contains 5 double precision complex.
  Size of the array in bytes: 80
> vnumC TE[1:2], TO[1:2]([1:6]);
> stat(TE);
  Array of series
 TE [ 1:2  ]
 list of array's elements :
TE [ 1 ] =
 Numerical vector TE contains 0 double precision complex.
TE [ 2 ] =
 Numerical vector TE contains 0 double precision complex.
> stat(TO);
  Array of series
 TO [ 1:2  ]
 list of array's elements :
TO [ 1 ] =
 Numerical vector TO contains 6 double precision complex.
  Size of the array in bytes: 96
TO [ 2 ] =
 Numerical vector TO contains 6 double precision complex.
  Size of the array in bytes: 96
>
```

10.1.3 vnumQ

`vnumQ` [Procedure]

`vnumQ` <name> , ... ;

`vnumQ` <name> [<dimension of an array>] , ... ;

It declares a vector of rational numbers or an array of vector of rational numbers.

The rational vector's size is dynamic. After the declaration, the rational vectors are empty. To set their sizes, the command `resize` must be used.

The dimension specifies the number of elements in the array of rational vectors.

`vnumQ` [Procedure]

`vnumQ` <name> ([1: <integer> *n*]) , ... ;

`vnumQ` <name> [<dimension of an array>] ([1: <integer> *n*]) , ... ;

It declares a vector of n rational numbers or an array of vectors of n rational numbers.

After the declaration, the elements of the vector are initialized with the value 0.

```
Example :
> _modenum=NUMRATMP;
_modenum   =    NUMRATMP
> vnumQ A, C([1:5]);
> stat(A);
 Numerical vector A contains 0 rational numbers.
> stat(C);
 Numerical vector C contains 5 rational numbers.
  Size of the array in bytes: 160
> vnumQ TE[1:2], TO[1:2]([1:6]);
> stat(TE);
  Array of series
 TE [ 1:2  ]
 list of array's elements :
TE [ 1 ] =
 Numerical vector TE contains 0 rational numbers.
TE [ 2 ] =
 Numerical vector TE contains 0 rational numbers.
> stat(TO);
  Array of series
 TO [ 1:2  ]
 list of array's elements :
TO [ 1 ] =
 Numerical vector TO contains 6 rational numbers.
  Size of the array in bytes: 192
TO [ 2 ] =
 Numerical vector TO contains 6 rational numbers.
  Size of the array in bytes: 192
>
```

10.2 Initialization

, , , [Procedure]

<name> = <real> *binf* , <real> *bsup* , <real> *step* ;

It declares and initializes a real vectors such that all elements are initialized by a loop (from *binf* to *bsup* with the step *bstep*):

for (j=1, valeur=*binf*) to valeur<=*bsup* step (j=j+1, valeur=valeur+*bstep*) <name> [j]=valeur

The step *bstep* is optional; By default, its value is 1.

```
Example :
> t=0,10;
 t  double precision real vector : number of elements =11
> writes(t);
+0.0000000000000000E+00
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
+4.0000000000000000E+00
+5.0000000000000000E+00
+6.0000000000000000E+00
+7.0000000000000000E+00
+8.0000000000000000E+00
+9.0000000000000000E+00
+1.0000000000000000E+01
> tab=-pi, 6, pi/100;
 tab  double precision real vector : number of elements =291
> writes([::30],tab);
-3.1415926535897931E+00
-2.1991148575128552E+00
-1.2566370614359170E+00
-3.1415926535897887E-01
+6.2831853071795907E-01
+1.5707963267948966E+00
+2.5132741228718354E+00
+3.4557519189487733E+00
+4.3982297150257113E+00
+5.3407075111026483E+00
>
```

vnumR[] [Function]

<name> = **vnumR** [<real> or <real vec.> or <array of real vec.> , ... : <real> ,...] ; It declares and initializes a real vector or an array of real vectors with the specified reals or real vectors.

The character : separates the lines and the character , separates the columns. All columns must have the same line counts.

```
Example :
> // declare an array of 3 vectors of 2 reals.
> tab3=vnumR[1,2,3:4,5,6];

tab3 [1:3 ] number of elements = 3

> writes(tab3);
+1.0000000000000000E+00 +2.0000000000000000E+00 +3.0000000000000000E+00
+4.0000000000000000E+00 +5.0000000000000000E+00 +6.0000000000000000E+00
> t=7,8;
 t  double precision real vector : number of elements =2
```

```
> tab4=vnumR[t,tab3];

tab4 [1:4 ] number of elements = 4

> // declare a vector of 5 reals.
> t2=vnumR[1:2:4:8:9];
 t2  double precision real vector : number of elements =5
> writes(t2);
+1.0000000000000000E+00
+2.0000000000000000E+00
+4.0000000000000000E+00
+8.0000000000000000E+00
+9.0000000000000000E+00
>
```

`vnumC[]` [Function]

<name> = `vnumC [` <complex> or <complex vec.> or <array of complex vec.> `,` ... `:` <complex> `,`... `]` `;`

It declares and initializes a complex vector or an array of complex vectors with the specified complexs or complex vectors.

The character `:` separates the lines and the character `,` separates the columns `.` All columns must have the same line counts.

```
Example :
> // declare a vector of 5 complex numbers
> tab3=vnumC[1:1+i:2+2*i:3+3*i];
 tab3  Double precision complex vector : number of elements =4
> writes(tab3);
+1.0000000000000000E+00 +0.0000000000000000E+00
+1.0000000000000000E+00 +1.0000000000000000E+00
+2.0000000000000000E+00 +2.0000000000000000E+00
+3.0000000000000000E+00 +3.0000000000000000E+00
> vnumC ti;
> resize(ti,5,3-5*i);
> tab4=vnumC[tab3 : 5+7*i : ti];
 tab4  Double precision complex vector : number of elements =10
> // declare an array of 2 vectors of 2 complex numbers
> z4=vnumC[5,2+i:4,9+3*i];

z4 [1:2 ] number of elements = 2

> writes("%g %g %g %g\n",z4[1],z4[2]);
5 0 2 1
4 0 9 3
>
```

10.3 Display

`writes` [Procedure]

`writes([` <integer> `:` <integer> `:` <integer> `],` <string> `,` <(array of) num. vec.> `,` ...`);`

`writes(` <string> `,` <(array of) num. vec.> `,` ...`);`

`writes(` <(array of) num. vec.> `,` ...`);`

equivalent to

writes({ [*binf*:{*bsup*}:{*step*}], } {*format*,} <(array of) num. vec.> ,...).

It prints, to the screen, numerical vectors or the array of numerical vectors in columns form.

From the *binf*th element of each numerical vector (if *binf* isn't specified, it starts from the first element).

To the *bsup*th element of each numerical vector (if *bsup* isn't specified, it stops to the last element of the biggest vector).

Every *step* elements (if *step* isn't specified, the step is 1).

format is optional. This format is similar to the C format (cf. printf) and must be between double quotes ("). To display a double quote (") as a character, you must double it :

For example, if the C format is " %g \"titi\" %g", you must write " %g\""titi\"" %g"

A complex vector will be printed on two columns (the first one for the real part and the second one for the imaginary part).

```
Example :
Ecriture de tous les éléments de T et de X.
La première colonne correspondra à T.
La deuxième colonne correspondra à la partie réelle de X.
La troisième colonne correspondra à la partie imaginaire de X.
...
> stat(X);
vecteur numérique X de 6 complexes.
  taille en octets du tableau: 96
> stat(T);
vecteur numérique T de 6 réels.
   taille en octets du tableau: 48
> writes(T,X);
+9.999993149794888E-02  +1.000000000456180E-01  +1.095970178673141E-06
-2.000000944035095E-01  +9.999999960679056E-03  +1.312007532388499E-07
+3.000000832689856E-01  +1.000000314882390E-03  -1.403292661135361E-08
-4.000000970924361E-01  +9.999995669530993E-05  +1.695880029074994E-09
+4.999999805039361E-01  +1.000003117384769E-05  +3.216502329016007E-11
-5.999999830866213E-01  +9.999979187109419E-07  -1.796078221829677E-12
>

Ecriture du 2 au 4 elements de T et de X
avec un format "%.1g\t(%.5g+i*%.5E)\n".
> writes([2:4],"%.1g\t(%.5g+i*%.5E)\n",T,X);
-0.2    (0.01+i*1.31201e-07)
0.3     (0.001+i*-1.40329e-08)
-0.4    (0.0001+i*1.69588e-09)

Ecriture du 1 au 5 elements de T et de X avec un pas de 2 sans format.
> writes([1:5:2],T,X);
+9.999993149794888E-02  +1.000000000456180E-01  +1.095970178673141E-06
+3.000000832689856E-01  +1.000000314882390E-03  -1.403292661135361E-08
+4.999999805039361E-01  +1.000003117384769E-05  +3.216502329016007E-11

Ecriture des elements de T et de X avec un pas de 5 sans format.
> writes([::5],T,X);
```

```
+9.999993149794888E-02  +1.000000000456180E-01  +1.095970178673141E-06
-5.999999830866213E-01  +9.999979187109419E-07  -1.796078221829677E-12
```

10.4 Size

`size` [Function]

`size(` <num. vec.> `)`

Return the number of elements in the numerical vector.

```
Example :
> t=1,10;
 t  double precision real vector : number of elements =10
> size(t);
 10
> p=-pi,pi,pi/400;
 p  double precision real vector : number of elements =800
> size(p);
 800
>
```

10.5 Resize

`resize` [Procedure]

`resize(`<(array of) num. vec.> `,` <integer> `);`

`resize(`<(array of) num. vec.> `,` <integer> `,` <constant> `);`

Change the size of the numerical vector or of the array of numerical vectors.

If a constant isn't specified then all elements will be initialized to 0 else all elements will initialized with this constant.

```
Example :
> vnumR t;
> resize(t,3);
> vnumR t2;
> resize(t2,3,5);
> writes(t,t2);
+0.0000000000000000E+00 +5.0000000000000000E+00
+0.0000000000000000E+00 +5.0000000000000000E+00
+0.0000000000000000E+00 +5.0000000000000000E+00
> vnumC t[1:3];
> resize(t[2],2,1-5*i);
> writes(t[2]);
+1.0000000000000000E+00 -5.0000000000000000E+00
+1.0000000000000000E+00 -5.0000000000000000E+00
>
```

10.6 Bounds

`inf` [Function]

`inf(`<num. vec.> `,`<integer> `);`

Return the lower bound of the numerical vector. The supplied integer must be equal to 1.

If the specified dimension is not 1, then the function returns -1.

```
Example :
> t=3,10;
 t  double precision real vector : number of elements =8
> inf(t,1);
 1
>
```

`sup` [Function]

`sup(<num. vec.> ,<integer> );`

Return the upper bound of the numerical vector. The supplied integer must be equal to 1.

If the specified dimension is not 1, then the function returns -1.

```
Example :
> t=3,10;
 t  double precision real vector : number of elements =8
> sup(t,1);
 8
>
```

10.7 Data retrieval

10.7.1 select

`select` [Function]

`select ( <condition> , <num. vec.> );`

Return a numerical vector which only contains the elements of the numerical vector where the condition is true.

The condition and the numerical vector must have the same number of elements.

```
Example :
> // return the elements which are multiple of 3
> t=1,10;
 t  double precision real vector : number of elements =10
> r=select((t mod 3)==0, t);
 r  double precision real vector : number of elements =3
> writes(r);
+3.0000000000000000E+00
+6.0000000000000000E+00
+9.0000000000000000E+00
>
```

10.7.2 Data retrieval

`[]` [Operator]

`<num. vec.> [ <real vec.> ];`

Return a numerical vector which only contains the elements of the numerical vector located at the indexes stored in the real vector.

Remark : The real vector must be an object identifier and not be a operation result.

```
Example :
> r=vnumR[1:5:7:9];
 r  double precision real vector : number of elements =4
> t=20,30;
```

```
 t  double precision real vector : number of elements =11
> b=t[r];
 b  double precision real vector : number of elements =4
> writes("%g\n",b);
20
24
26
28
>
```

`[::]` [Operator]

<num. vec.> `[` <integer> *binf* `:` <integer> *bsup* `:` <integer> *pas* `];`

<num. vec.> `[` <integer> *binf* `:` <integer> *bsup* `];`

Return a numerical vector which contains only the elements located between the lower bound and the upper bound with the specified step.

If the lower bound isn't specified, then the lower bound is assumed to be 1.
If the upper bound isn't specified, then the upper bound is assumed to be the size of the numerical vector.
If the step isn't specified, then the step is assumed to be 1.

Remark : All missing combinations are allowed.

```
Example :
> t=1,10;
 t  double precision real vector : number of elements =10
> r=t[::2];
 r  double precision real vector : number of elements =5
> v=t[7:9];
 v  double precision real vector : number of elements =3
> y=t[5:10:2];
 y  double precision real vector : number of elements =3
> writes("%g %g\n",v,y);
7 5
8 7
9 9
>
```

10.8 Input/Output

10.8.1 read

`read` [Procedure]

`read(`<filename> `,[` <integer> `:` <integer> `:` <integer> `],`
  <(array of) num. vec.> `,...` `,`
  `(`<(array of) num. vec.> `,` <integer> `),` ...
  `(`<(array of) num. vec.> `,` <integer> `,` <integer> `),` ...`);`

equivalent to

read(fichier.dat, [binf:bsup:step], T, (T1,n1), (T3,n2,n3));

read(fichier.dat, T, (T1,n1), (T3,n2,n3));

Read in a text file the specified columns and store them to numerical vectors.

  starting at the the line binf (if binf isn't specified, then it will start at the first line of the file)

to the line bsup (if bsup isn't specified, then it will read to the end of file)
every bstep lines (if bstep isn't specified, then the step is assumed to 1)

- If the index of the column for a numerical vector of real numbers is specified, then the column for this vector will be the index of the last column +1.
- If the index of the column for the real part of a numerical vector of complex numbers is specified, then the column for this vector will be the index of the last column +1.
- If the index of the column for the imaginary part of a numerical vector of complex numbers isn't specified and the column for the real part is specified, then the imaginary part is initialized to 0.

If the file contains the strings `NAN` or `NANQ`, then it assumes to be the value "Not A Number". If the file contains the strings `INF`, `Inf` ou `Infinity`, then it assumes to be the value "infinite".

When an empty or incomplete line is encountered, the behavior of this function depends on the value of `_read` (see Section 3.25 [_read], page 19).

```
Example :
Lecture dans le fichier tessin.out de la ligne 2 à ligne 100
avec un pas de 3.
Le vecteur T contiendra la première colonne,
la partie réelle de X contiendra la deuxième colonne,
la partie imaginaire de X contiendra la troisième colonne,
TAB[1] contiendra la 4eme colonne,
TAB[2] contiendra la 5eme colonne,
TAB[3] contiendra la 6eme colonne.
> vnumR T;
vnumC X;
vnumR TAB[1:3];
read(tessin.out,[2:100:3],T,X,TAB);
stat(T);
stat(X);
stat(TAB);
T    nb elements reels =0
>
X    nb elements complexes =0
>
TAB [1:3]   nb elements = 3

> > Tableau numerique T de 33 reels.
    taille en octets du tableau: 264
> Tableau numerique X de 33 complexes.
    taille en octets du tableau: 528
>   Tableau de series
 TAB [ 1:3 ]
 liste des elements du tableau :
    TAB [ 1 ] =
Tableau numerique  de 33 reels.
    taille en octets du tableau: 264
    TAB [ 2 ] =
Tableau numerique  de 33 reels.
    taille en octets du tableau: 264
    TAB [ 3 ] =
Tableau numerique  de 33 reels.
```

```
        taille en octets du tableau: 264
    >

    Lecture dans le fichier tessin.out de la ligne 2 a ligne 100.
    Le vecteur T contiendra la premiere colonne,
    la partie reelle de X contiendra la 4eme colonne,
    la partie imaginaire de X sera nulle,
    TAB[3] contiendra la 5eme colonne.
    > read(tessin.out,[2:100],T,(X,4),(TAB[3],5));


    Lecture dans le fichier tessin.out de l'ensemble des lignes.
    Le vecteur T contiendra la 2eme colonne,
    la partie reelle de X contiendra la 3eme colonne,
    la partie imaginaire de X contiendra la 4eme colonne,
    TAB[3] contiendra la 5eme colonne.
    > read(tessin.out,(T,2),(X,3,4),(TAB[3],5));

    Lecture dans le fichier tessin.out a partir de la ligne 1000.
    Le vecteur T contiendra la 2eme colonne,
    la partie reelle de X contiendra la 3eme colonne,
    la partie imaginaire de X contiendra la 4eme colonne,
    TAB[3] contiendra la 5eme colonne.
    > read(tessin.out,[1000:],(T,2),(X,3,4),(TAB[3],5));

    Lecture dans le fichier tessin.out avec un pas de 50 lignes.
    Le vecteur T contiendra la 2eme colonne,
    la partie reelle de X contiendra la 3eme colonne,
    la partie imaginaire de X contiendra la 4eme colonne,
    TAB[3] contiendra la 5eme colonne.
    > read(tessin.out,[::50],(T,2),(X,3,4),TAB[3]);
```

`read` [Procedure]

`read(`<filename> , [<integer> : <integer> : <integer>] , *textformat*,

<(array of) num. vec.> ,... ,

(<(array of) num. vec.> , <integer>) , ...

(<(array of) num. vec.> , <integer> , <integer>) , ...);

equivalent to

fmt="..."; read(fichier.dat, [binf:bsup:step], textformat, T, (T1,n1), (T3,n2,n3));

fmt="..."; read(fichier.dat, textformat, T, (T1,n1), (T3,n2,n3));

This function is similar to the above command but allows to specify a format to read the data. It is able to read columns of numbers or strings of characters. The type of the data is specified by the format.

The allowed formats are :

– %E double-precision floating-point numbers.
– %e double-precision floating-point numbers.
– %g double-precision floating-point numbers.
– %s string of characters

For the format specifier %s, the object must not be declared before ; the returned object is an array of string.

For the format specifiers %g, %e, %E, the objects must be declared before this command as a (array of) numerical vector (vnumR). The returned object will be of the same type.

By default, it assumes that the columns are separated by spaces or tabulions. If a length is specified in the format (e.g., %10g or %10s for a column of 10 characters), this length specifies the number of characters of the column. In this case, there is no distinction between spaces, tabulations and other characters.

When an empty or incomplete line is encountered, the behavior of this function depends on the value of `_read` (see Section 3.25 [_read], page 19).

Limitation : It is not possible to write the following command : `read(file,"....", T1, T2, ...);` . It must be rewritten as : `fmt=" ..."$ read(file,fmt, T1, T2, ...);`.

```
Example :
> // generates a file with 2 columns : asteroid number and name
> f=file_open(data.dat, write)$
> file_writemsg(f, "    1 Ceres  \n    2 Pallas\n    3 Juno  \n");
> file_close(f);
>
>
> // reads two columns like a numerical vector and an array of string
> // using a space separator
> vnumR id;
> fmt="%g %s";
fmt = "%g %s"
> read(data.dat, fmt, id, name);
> writes(id);
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
> afftab(name);
name[1] =  "Ceres"
name[2] =  "Pallas"
name[3] =  "Juno"
>
> // reads two columns like a numerical vector and an array of string
> // using a fixed size for each column
> vnumR id2;
> fmt="%5g %7s";
fmt = "%5g %7s"
> read(data.dat, fmt, id2, name2);
> writes(id2);
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
> afftab(name2);
name2[1] =  " Ceres "
name2[2] =  " Pallas"
name2[3] =  " Juno  "
```

The following example show how to read the astorb data files using the format.

```
Example :
```

```
vnumR number, H, Slope,ColorIndex,v1,v2,v3,v4,v5,v6;
vnumR arc,observations, epochyy,epochymm, epochdd;
vnumR anomaly,perihelion, ascendingnode,Inclination,
      Eccentricity,Semimajoraxis,Date;

fmt="%7s%19s%16s%6s%5g%5s%6s%6s"+6*"%4g"+2*"%g"+"%4g"
    +2*"%2g"+5*"%11g"+"%12g%9s";

read(astorb.dat,[:10000], fmt,  number,name, computer, H,Slope,
     ColorIndex,diameter,Taxonomic,v1,v2,v3,v4,v5,v6,arc,observations,
     epochyy,epochymm, epochdd, anomaly,perihelion, ascendingnode,
     Inclination,Eccentricity,Semimajoraxis,Date);
```

10.8.2 readappend

`readappend` [Procedure]

`readappend(`<filename> `,[` <integer> `:` <integer> `:` <integer> `],`
  `<`(array of) num. vec.`>` `,...` `,`
  `(<`(array of) num. vec.`>` `,` <integer> `),` ...
  `(<`(array of) num. vec.`>` `,` <integer> `,` <integer> `),` ...`);`

equivalent to

readappend(fichier.dat, [binf:bsup:step], T, (T1,n1), (T3,n2,n3));

readappend(fichier.dat, T, (T1,n1), (T3,n2,n3));

This function is similar to the function `read` (see Section 10.8.1 [read], page 63) but it stores read data to the end of the numerical vectors (with an automatic growing step) instead of overwriting their contents.

```
Example :
>  t1=1,5;
t1       Tableau de reels double-precision : nb reels =5
> write(temp001, t1);
> write(temp002, t1**2);
> vnumR w;
> readappend(temp001,w);
> readappend(temp002,w);
> writes(w);
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
+4.0000000000000000E+00
+5.0000000000000000E+00
+1.0000000000000000E+00
+4.0000000000000000E+00
+9.0000000000000000E+00
+1.6000000000000000E+01
+2.5000000000000000E+01
```

`readappend` [Procedure]

`readappend(`<filename> `,[` <integer> `:` <integer> `:` <integer> `],` *textfmt*,
  `<`(array of) num. vec.`>` `,...` `,`
  `(<`(array of) num. vec.`>` `,` <integer> `),` ...

(<(array of) num. vec.> , <integer> , <integer>), ...);

This function is similar to the function `read` (see Section 10.8.1 [read], page 63) using the format *textfmt*. But it stores read data to the end of the numerical vectors (with an automatic growing step) instead of overwriting their contents.

10.8.3 write

`write` [Procedure]

`write(` <filename> , `[` <integer> `:` <integer> `:` <integer> `]` , <string> , <(array of) num. vec.> , ...`);`

`write(` <filename> , <string> , <(array of) num. vec.> , ...`);`

`write(` <filename> , <(array of) num. vec.> , ...`);`

equivalent to

write(fichier.dat, [binf:bsup:step], "format", T, T1, T2).

write(fichier.dat, "format", T, T1, T2).

write(fichier.dat, T, T1, T2).

write(fichier.dat, [binf:bsup:step], T, T1, T2).

Write to a file the numerical vector or the arrays of numerical vectors as columns of numbers.

from the element *binf* of each numerical vector (if *binf* isn't specified, then it's assumed to be the first one).

to the element *bsup* of each numerical vector (if *bsup* isn't specified, then it's assumed to be the last one of the largest vector).

every *step* element (if *step* isn't specified, then it's assumed to be 1).

The format is optional. The format is the format of C language (See printf) and surrounded with double-quotes ("). To obtain a double-quotes, two double-quotes must be used :

For example : if the C format is " %g \"titi\" %E", it must be written as, " %g \""titi\"" %E"

A numerical vector of complex numbers uses two columns (the first one for the real part and the second one for the imaginary part).

```
Example :
Ecriture de tous les elements de T et de X dans le fichier tx.out.
La pemiere colonne correspondra a T.
la deuxieme colonne correspondra a la partie reelle de X.
la troisieme colonne correspondra a la partie imaginaire de X.
> vnumR T;
vnumC X;
...
stat(T);
stat(X);

> Tableau numerique T de 33 reels.
    taille en octets du tableau: 264
> Tableau numerique X de 33 complexes.
    taille en octets du tableau: 528
> write(tx.out,T,X);
>

Ecriture de 10 au 20 éléments de T et de X dans le fichier tx.out avec
```

```
un format "%g\t(%8.4E+i*%e)\n".
La pemiere colonne correspondra à T.
la deuxième colonne correspondra à la partie réelle de X.
la troisieme colonne correspondra à la partie imaginaire de X.
> write(tx.out,[10:20],"%g\t(%8.4E+i*%e)\n",T,X);

Ecriture de 1 au 20 éléments de T et de X avec un pas de 2 dans le
fichier tx.out sans format.
La pemiere colonne correspondra à T.
la deuxième colonne correspondra à la partie réelle de X.
la troisieme colonne correspondra à la partie imaginaire de X.
> write(tx.out,[1:20:2],T,X);

Ecriture des éléments de T et de X avec un pas de 5 dans le fichier
tx.out sans format.
La pemiere colonne correspondra à T.
la deuxième colonne correspondra à la partie réelle de X.
la troisieme colonne correspondra à la partie imaginaire de X.
> write(tx.out,[::5],T,X);
```

10.8.4 readbin

`readbin` [Procedure]

`readbin(`<filename> `,[` <integer> `:` <integer> `:` <integer> `]`, <string> `,` <(array of) real vec.>`, ...);`

`readbin(`<filename> `,` <string> `,` <(array of) real vec.>`, ...);`

equivalent to

readbin(fichier.dat, [binf:bsup:step], format, T, T2, T3);

readbin(fichier.dat, format, T, T2, T3);

Read in a binary file the data with the specified format and store them to numerical vectors.

beginning at the record binf (if binf isn't specified, then it's assumed to be first record).

to the record bsup (if bsup isn't specified, then it's assumed to be the end of file)

every bstep records (if bstep isn't specified, then it's assumed to be 1)

The allowed format to define a record are :

- %2d signed integer on 2 bytes.
- %4d signed integer on 4 bytes.
- %8d signed integer on 8 bytes.
- %d signed integer on (default size of the system for an integer).
- %2u unsigned integer on 2 bytes.
- %4u unsigned integer on 4 bytes.
- %8u unsigned integer on 8 bytes.
- %u unsigned integer (default size of the system for an integer).
- %E double-precision floating-point number.
- %e double-precision floating-point number.
- %g double-precision floating-point number.
- %8g double-precision floating-point number on 8 bytes.
- %hE simple precision floating-point number.

– %he simple precision floating-point number.
– %hg simple precision floating-point number.
– %4g simple precision floating-point number on 4 bytes.

The number of format must be the same as the number of vectors. If the provided object identifier is an array of real vectors, then it must have the same number of format as the number of elements in the array.

The data are previously converted if the global variable `_endian` (see Chapter 3 [_endian], page 9) isn't the same as its default value.

```
Example :
Lecture dans le fichier binaire test1.dat
d'entiers signés sur 4 octets
et stockés dans le vecteur de réels T1.
vnumR T1;
readbin(test1.dat,"%4d",T1);

Lecture dans le fichier binaire test1.dat
des 30 premiers entiers signés sur 4 octets
et stockés dans le vecteur de réels T1.
vnumR T1;
readbin(test1.dat,[:30],"%4d",T1);

Lecture dans le fichier binaire test2.dat
dont chaque enregistrement est composé de 2 réels
et stockés dans le vecteur de réels T1 et T2.
vnumR T1,T2;
readbin(test2.dat,[:30],"%g%g",T1,T2);

Lecture dans le fichier binaire test3.dat
dont chaque enregistrement est composé de 4 réels
et stockés dans le tableau T3.
vnumR T3[1:4];
readbin(test3.dat,"%g%g%g%g",T3);

est équivalent à

readbin(test3.dat,"%g%g%g%g",T3[1],T3[2],T3[3],T3[4]);
```

10.8.5 writebin

`writebin` [Procedure]

`writebin(`<filename> `,[` <integer> `:` <integer> `:` <integer> `],` <string> `,` <(array of) real vec.>`,` ...`);`

`writebin(`<filename> `,` <string> `,` <(array of) real vec.>`,` ...`);`

equivalent to

writebin(fichier.dat, [binf:bsup:step], format, T, T2, T3);

writebin(fichier.dat, format, T, T2, T3);

Write the numerical vectors to a binary file the data with the specified format.

beginning at the index binf (if binf isn't specified, then it's assumed to be first element of the numerical vectors).

to the index bsup (if bsup isn't specified, then it's assumed to be the size of the largest numerical vector)

every bstep elements (if bstep isn't specified, then it's assumed to be 1)

The allowed format to define a record are :

- %2d signed integer on 2 bytes.
- %4d signed integer on 4 bytes.
- %8d signed integer on 8 bytes.
- %d signed integer on (default size of the system for an integer).
- %2u unsigned integer on 2 bytes.
- %4u unsigned integer on 4 bytes.
- %8u unsigned integer on 8 bytes.
- %u unsigned integer (default size of the system for an integer).
- %E double-precision floating-point number.
- %e double-precision floating-point number.
- %g double-precision floating-point number.
- %8g double-precision floating-point number on 8 bytes.
- %hE simple precision floating-point number.
- %he simple precision floating-point number.
- %hg simple precision floating-point number.
- %4g simple precision floating-point number on 4 bytes.

The number of format must be the same as the number of vectors. If the provided object identifier is an array of real vectors, then it must have the same number of format as the number of elements in the array.

If the numerical vectors doesn't have the same size, then the missing data are completed with 0.

The data are previously converted if the global variable `_endian` (see Chapter 3 [_endian], page 9) isn't the same as its default value.

```
Example :
Ecriture du vecteur de réels T1 dans le fichier binaire test1.dat
sosu la forme d'entiers signés sur 4 octets.
vnumR T1;
T1=1,10;
writebin(test1.dat,"%4d",T1);

Ecriture des 30 premiers éléments du vecteur de réels T1
dans le fichier binaire test1.dat sous la forme d'entiers
signés sur 4 octets.
vnumR T1;
T1=1,100;
writebin(test1.dat,[:30],"%4d",T1);

Ecriture des 30 premiers éléments du vecteur de réels T1 et de T2
dans le fichier binaire test2.dat dont chaque enregistrement
est composé de 2 réels.
vnumR T1,T2;
T1=1,100;
```

```
T2=-100,-1;
writebin(test2.dat,[:30],"%g%g",T1,T2);

Ecriture des vecteurs de réels de T3 dans le fichier binaire test3.dat
dont chaque enregistrement est composé de 4 réels.
vnumR T3[1:4];
T3[:]=1,10;
writebin(test3.dat,"%g%g%g%g",T3);

est équivalent à

writebin(test3.dat,"%g%g%g%g",T3[1],T3[2],T3[3],T3[4]);
```

10.9 Input/Output low level

10.9.1 file_open

`<file> file_open` [Function]

`file_open(<filename> , read);`

`file_open(<filename> , write);`

`file_open(<filename> , write, append);`

It opens a file in reading or writing mode depending on the value of the second parameter. It returns an object identifier of type file.

In writing mode, the existing file is kept and the new writing operations are performed at the end of the file if `append` is given, otherwise, the file is overwritten.

```
Example :
> f=file_open("sim2007.dat",read);
f  = fichier "sim2007.dat" ouvert en lecture
> vnumR t;
> file_read(f,t);
> file_close(f);
```

10.9.2 file_close

`file_close` [Procedure]

`file_close(<file> );`

It closes a file previously opened with the function `file_open`.

```
Example :
> f=file_open("sim2007.dat",read);
f  = fichier "sim2007.dat" ouvert en lecture
> vnumR t;
> file_read(f,t);
> file_close(f);
> stat(f);
 fichier f = "sim2007.dat" ferme
         Aucune erreur en lecture/ecriture
```

10.9.3 file_write

`file_write` [Procedure]

`file_write( <file> , [ <integer> : <integer> : <integer> ], <string> , <(array of) num. vec.> , ...);`

`file_write(` <file> `,` <string> `,` <(array of) num. vec.> `,` ...`);`

`file_write(` <file> `,` <(array of) num. vec.> `,` ...`);`

equivalent to

file_write(fichier, [binf:bsup:step], "format", T, T1, T2).

file_write(fichier, "format", T, T1, T2).

file_write(fichier, T, T1, T2).

file_write(fichier, [binf:bsup:step], T, T1, T2).

This function is similar as the function `write` (see Section 10.8.3 [write], page 68) but this function requires an object identifier of type file instead of a filename. The function writes data in the file from the current position.

```
Example :
> f=file_open(sim2007.dat, write);
f  = fichier "sim2007.dat" ouvert en ecriture
> t1=1,10;
t1       Tableau de reels double-precision : nb reels =10
> t2=-t1;
t2       Tableau de reels double-precision : nb reels =10
> file_write(f,t1);
> file_write(f,t2);
> file_close(f);
```

10.9.4 file_read

`file_read` [Procedure]

`file_read(`<file> `,[` <integer> `:` <integer> `:` <integer> `],`
  <(array of) num. vec.> `,`... `,`
  `(`<(array of) num. vec.> `,` <integer> `),` ...
  `(`<(array of) num. vec.> `,` <integer> `,` <integer> `),` ...`);`

equivalent to

file_read(fichier, [binf:bsup:step], T, (T1,n1), (T3,n2,n3));

file_read(fichier, T, (T1,n1), (T3,n2,n3));

This function is similar as the function `read` (see Section 10.8.1 [read], page 63) but this function requires an object identifier of type file instead of a filename. The function reads data from the file from the current position.

```
Example :
> f=file_open(sim2007.dat, read);
f  = fichier "sim2007.dat" ouvert en lecture
> vnumR t;
> file_read(f,[:5],t);
> vnumR t2;
> file_read(f,t2);
> writes(t);
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
+4.0000000000000000E+00
+5.0000000000000000E+00
> writes(t2);
```

```
+6.0000000000000000E+00
+7.0000000000000000E+00
+8.0000000000000000E+00
+9.0000000000000000E+00
+1.0000000000000000E+01
-1.0000000000000000E+00
-2.0000000000000000E+00
-3.0000000000000000E+00
-4.0000000000000000E+00
-5.0000000000000000E+00
-6.0000000000000000E+00
-7.0000000000000000E+00
-8.0000000000000000E+00
-9.0000000000000000E+00
-1.0000000000000000E+01
> file_close(f);
```

10.9.5 file_readappend

`file_readappend` [Procedure]

`file_readappend(`<file> `,[` <integer> `:` <integer> `:` <integer> `],`

<(array of) num. vec.> `,`... `,`

`(`<(array of) num. vec.> `,` <integer> `),` ...

`(`<(array of) num. vec.> `,` <integer> `,` <integer> `),` ...`);`

equivalent to

file_readappend(fichier, [binf:bsup:step], T, (T1,n1), (T3,n2,n3));

file_readappend(fichier, T, (T1,n1), (T3,n2,n3));

This function is similar as the function `readappend` (see Section 10.8.2 [readappend], page 67) but this function requires an object identifier of type file instead of a filename. The function reads data from the file from the current position.

```
Example :
> f=file_open(sim2007.dat, read);
f  = fichier "sim2007.dat" ouvert en lecture
> vnumR t;
>  file_readappend(f,[:5],t);
>  file_readappend(f,[10:15],t);
> writes(t);
+1.0000000000000000E+00
+2.0000000000000000E+00
+3.0000000000000000E+00
+4.0000000000000000E+00
+5.0000000000000000E+00
-5.0000000000000000E+00
-6.0000000000000000E+00
-7.0000000000000000E+00
-8.0000000000000000E+00
-9.0000000000000000E+00
-1.0000000000000000E+01
> file_close(f);
```

10.9.6 file_writemsg

`file_writemsg` [Procedure]

`file_writemsg(`<file> `,` <string> *textformat*`);`

`file_writemsg(`<file> `,` <string> *textformat*, <real> *x*, ... `);`

Write formatted messages to a file with(out) real constants.

The real constants must be formatted. The format is the same as the command printf in language C (see Section 7.6 [str], page 33, for the valid conversion specifiers). This message must a be string or a text between double-quotes. The messages could be on several lines.

To display double-quotes, two double-quotes must be used.

```
Example :
> f=file_open("temp001", write);
f  = fichier "temp001" ouvert en ecriture
> x=4;
x =                        4
> file_writemsg(f," resultat=%g\n", x);
> file_writemsg(f," termine\n");
> file_close(f);
> !"cat temp001";
 resultat=4
 termine
> f=file_open("data1.txt",write);
f  = fichier "data1.txt" ouvert en ecriture
> file_writemsg(f,"%d\n",3/2);
> file_writemsg(f,"%4.2E\n",3/2);
> file_close(f);
> !"cat data1.txt";
3/2
1.50E+00
>
```

10.9.7 file_writebin

`file_writebin` [Procedure]

`file_writebin(` <file> `,` `[` <integer> `:` <integer> `:` <integer> `]`, <string> `,` <(array of) num. vec.> `,` ...`);`

`file_writebin(` <file> `,` <string> `,` <(array of) num. vec.> `,` ...`);`

equivalent to

file_writebin(file, [binf:bsup:step], "format", T, T1, T2).

file_writebin(file, "format", T, T1, T2).

This function is similar as the function `writebin` (see Section 10.8.5 [writebin], page 70) but this function requires an object identifier of type file instead of a filename. The function writes data in the file from the current position.

```
Example :
> f=file_open(sim2007.dat, write);
 f  = file "sim2007.dat"  opened in write mode
> t1=1,10;
 t1  double precision real vector : number of elements =10
> t2=-t1;
 t2  double precision real vector : number of elements =10
```

```
> file_writebin(f,"%g",t1);
> file_writebin(f,"%g",t2);
> file_close(f);
```

10.10 Standard functions

10.10.1 minimum and maximum

`imin` [Function]

`imin(` <real vec.> `)`

Return the index of the minimal value of the numerical vector of real numbers. If more than one elements have the minimal value, then this function returns the index of the first element. If the numerical vector is empty then it returns 0.

```
Example :
> t=-5,5;
> imin(t);
    1
```

`imax` [Function]

`imax(` <real vec.> `)`

Return the index of the maximal value of the numerical vector of real numbers. If more than one elements have the maximal value, then this function returns the index of the first element. If the numerical vector is empty then it returns 0.

```
Example :
> t=-5,5;
> imax(t);
     10
```

`min` [Function]

`min(` <real or real vec.> `,` ...`);`

Return the minimum value of the parameters which could be real constants or real vectors.

```
Example :
>min(1.5,2.5);
3/2
>x=1.5;
>y=2.5;
>min(x,y);
3/2
> t=-10,10;
> p=abs(t);
> min(-5,t,p,20);
    -10
```

`max` [Function]

`max(` <real or real vec.> `,` ...`);`

Return the maximum value of the parameters which could be real constants or real vectors.

```
Example :
>max(1.5,2.5);
5/2
>x=1.5;
```

```
>y=2.5;
>max(x,y);
5/2
> t=-10,10;
> p=abs(t);
> max(p,t,8);
    10
```

`IMIN` [Function]

`IMIN(` <array of real vec.>)

Return a numerical vector of real numbers such that its elements have the index of the column where is located the minimal value term by term of each numerical vector.

The numerical vectors must have the same size.

```
Example :
>   A=vnumR[1,2,3:6,7,1:9,2,5:13,11,16]$
>   writes("%g %g %g\n", A);
1 2 3
6 7 1
9 2 5
13 11 16
>   B=IMIN(A);
 B  double precision real vector : number of elements =4
>   writes("%g\n", B);
1
3
2
2
>
```

`IMAX` [Function]

`IMAX(` <array of real vec.>)

Return a numerical vector of real numbers such that its elements have the index of the column where is located the maximal value term by term of each numerical vector.

The numerical vectors must have the same size.

```
Example :
>   A=vnumR[1,2,3:6,7,1:9,2,5:13,11,16]$
>   writes("%g %g %g\n", A);
1 2 3
6 7 1
9 2 5
13 11 16
>   B=IMAX(A);
 B  double precision real vector : number of elements =4
>   writes("%g\n", B);
3
2
1
3
>
```

`MIN` [Function]

`MIN(` <real vec.> , ...)

Return a numerical vector of real numbers such that its elements have the minimal value term by term of each numerical vector.

The numerical vectors must have the same size.

For an array of numerical vector of real numbers, the operation is performed on each element.

```
Example :
> t=0,pi,pi/6$
> a=MIN(cos(t),sin(t))$
> c=MAX(t,cos(t),sin(t))$
> writes(a,c)$
+0.0000000000000000E+00 +1.0000000000000000E+00
+4.9999999999999994E-01 +8.6602540378443871E-01
+5.0000000000000011E-01 +1.0471975511965976E+00
+6.1232339957367660E-17 +1.5707963267948966E+00
-4.9999999999999978E-01 +2.0943951023931953E+00
-8.6602540378443849E-01 +2.6179938779914940E+00
-1.0000000000000000E+00 +3.1415926535897931E+00
> vnumR cs[1:2]$
> cs[1]=cos(t)$
> cs[2]=sin(t)$
> b=MIN(cs)$
> d=MAX(t,cs)$
> writes(b,d);
+0.0000000000000000E+00 +1.0000000000000000E+00
+4.9999999999999994E-01 +8.6602540378443871E-01
+5.0000000000000011E-01 +1.0471975511965976E+00
+6.1232339957367660E-17 +1.5707963267948966E+00
-4.9999999999999978E-01 +2.0943951023931953E+00
-8.6602540378443849E-01 +2.6179938779914940E+00
-1.0000000000000000E+00 +3.1415926535897931E+00
```

`MAX` [Function]

`MAX(` <real vec.> `, ...)`

Return a numerical vector of real numbers such that its elements have the maximal value term by term of each numerical vector.

The numerical vectors must have the same size.

For an array of numerical vector of real numbers, the operation is performed on each element.

```
Example :
> t=0,pi,pi/6$
> a=MIN(cos(t),sin(t))$
> c=MAX(t,cos(t),sin(t))$
> writes(a,c)$
+0.0000000000000000E+00 +1.0000000000000000E+00
+4.9999999999999994E-01 +8.6602540378443871E-01
+5.0000000000000011E-01 +1.0471975511965976E+00
+6.1232339957367660E-17 +1.5707963267948966E+00
-4.9999999999999978E-01 +2.0943951023931953E+00
-8.6602540378443849E-01 +2.6179938779914940E+00
-1.0000000000000000E+00 +3.1415926535897931E+00
> vnumR cs[1:2]$
> cs[1]=cos(t)$
> cs[2]=sin(t)$
> b=MIN(cs)$
> d=MAX(t,cs)$
> writes(b,d);
+0.0000000000000000E+00 +1.0000000000000000E+00
+4.9999999999999994E-01 +8.6602540378443871E-01
+5.0000000000000011E-01 +1.0471975511965976E+00
+6.1232339957367660E-17 +1.5707963267948966E+00
-4.9999999999999978E-01 +2.0943951023931953E+00
-8.6602540378443849E-01 +2.6179938779914940E+00
-1.0000000000000000E+00 +3.1415926535897931E+00
```

10.10.2 addition and product

`sum` [Function]

`sum(` <num. vec.>)

`sum(` <num. vec.> , "kahan")

`sum(` <num. vec.> , "sort", "kahan")

Return the sum of elements in the numerical vector.

The option "sort" sort the vector using the ascending norm before the summation is done.

If the option "kahan" is set, the function uses the Kahan method to perform the summation (compensated summation).

Reference : Kahan, William (January 1965), "Further remarks on reducing truncation errors", Communications of the ACM 8 (1): 40, `http://dx.doi.org/10.1145%2F363707.363723`

```
Example :
> r=0,100$
> sum(r);
                        5050
> p=-10000,10000$
> p=p*pi/10000$
> sum(p);
      3.885780586188048E-13
> // compensated summation
> sum(p,"sort","kahan");
                        0
```

`prod` [Function]

`prod(` <num. vec.>)

Return the product of elements in the numerical vector.

```
Example :
> p=1,10;
p    Tableau de reels : nb reels =10
> prod(p);
    3628800
> c=p+2*i*p;
c    vecteur de complexes : nb complexes =10
> prod(c);
      (860025600-i*11307340800)
```

`accum` [Function]

`accum(` <num. vec.> `)`

Return the partial sum of elements in the numerical vector.

$$Y = accum(X) : Y[N] = \sum_{i=1}^{N} X[i]$$

```
Example :
> tx=1,10;
tx   Tableau de reels : nb reels =10
> ty=accum(tx);
ty   Tableau de reels : nb reels =10
> writes(accum(tx));
+1.000000000000000E+00
+3.000000000000000E+00
+6.000000000000000E+00
+1.000000000000000E+01
+1.500000000000000E+01
+2.100000000000000E+01
+2.800000000000000E+01
+3.600000000000000E+01
+4.500000000000000E+01
+5.500000000000000E+01
```

10.10.3 sort

`intersectvnum` [Function]

`intersectvnum(`<num. vec.> *A* `,`<num. vec.> *B*`)`

It returns the values common to both numerical vectors *A* and *B*. The values of returned vector are in sorted order using ascending order.

```
Example :
> A = vnumR[7:1:7:5:4:6]$
> B = vnumR[0:7:4:-2:5:3]$
> C = intersectvnum(A,B);
 C  double precision real vector : number of elements =3
> writes(C);
+4.0000000000000000E+00
+5.0000000000000000E+00
+7.0000000000000000E+00
>
```

`unionvnum` [Function]

`unionvnum(`<num. vec.> *A* `,`<num. vec.> *B*`)`

It returns a vector which contains the combined values from *A* and *B* with no repetitions. The values of returned vector are in sorted order using ascending order.

```
Example :
> A = vnumR[7:1:7:5:4:6]$
> B = vnumR[0:7:4:-2:5:3]$
> C = unionvnum(A,B);
 C  double precision real vector : number of elements =8
> writes(C);
```

```
-2.0000000000000000E+00
+0.0000000000000000E+00
+1.0000000000000000E+00
+3.0000000000000000E+00
+4.0000000000000000E+00
+5.0000000000000000E+00
+6.0000000000000000E+00
+7.0000000000000000E+00
>
```

reversevnum [Function]

`reversevnum` (<num. vec.>)

Reverse the order of elements in the numerical vector.

```
Example :
> p=1,10;
p    Tableau de reels : nb reels =10
>  writes(p, reversevnum(p));
+1.000000000000000E+00  +1.000000000000000E+01
+2.000000000000000E+00  +9.000000000000000E+00
+3.000000000000000E+00  +8.000000000000000E+00
+4.000000000000000E+00  +7.000000000000000E+00
+5.000000000000000E+00  +6.000000000000000E+00
+6.000000000000000E+00  +5.000000000000000E+00
+7.000000000000000E+00  +4.000000000000000E+00
+8.000000000000000E+00  +3.000000000000000E+00
+9.000000000000000E+00  +2.000000000000000E+00
+1.000000000000000E+01  +1.000000000000000E+00
```

sort [Procedure]

`sort` (<real vec.>)

Sort in ascending order the elements in the real vector.

`sort` (<real vec.> , <(array of) num. vec.> , ...);

Sort the elements in the (array of) numerical vectors according to the ascending sort of the first real vector.

Remarks : If you call '`sort(TX,TY,TZ);`', the vector *TY* et *TZ* are sorted but *TX* isn't sorted.

```
Example :
> p=vnumR[5:2:-3:7:1:6];
p    Tableau de reels : nb reels =6
> sort(p);
> writes(p);
-3.000000000000000E+00
+1.000000000000000E+00
+2.000000000000000E+00
+5.000000000000000E+00
+6.000000000000000E+00
+7.000000000000000E+00
> T=vnumC[1+i:2+2*i:3+3*i:4+4*i:5+5*i:7+7*i];
T    vecteur de complexes : nb complexes =6
> vnumR TAB[1:2];
> TAB[1]=abs(T);
> TAB[2]=-real(T);
> sort(-p,T,TAB);
```

10.10.4 transposevnum

`transposevnum` [Function]

`transposevnum` (<array of num. vec.>)

Transpose the array of numerical vectors.

The array must have only one dimension and its numerical vectors must have the same size.

```
Example :
> t=1,5;
t        Tableau de reels : nb reels =5
> vnumR TSRC[1:2];
> TSRC[1]=t;
> TSRC[2]=reversevnum(t);
> writes("%g %g\n",TSRC);
1 5
2 4
3 3
4 2
5 1
> TRES=transposevnum(TSRC);

TRES [1:5]      nb elements = 5

> writes("%g %g %g %g %g\n",TRES);
1 2 3 4 5
5 4 3 2 1
```

10.10.5 standard math function

`abs` [Function]

`abs`(<constant or num. vec.>)

Return the absolute value of the constant if the value is a constant.

Return a numerical vector containing the absolute value of each element if the value is a numerical vector.

```
Example :
> abs(-2.3);
2.3
> x = 2.3;
> abs(x);
2.3
> t=-pi,pi,pi/100;
> at=abs(t);
at   Tableau de reels : nb reels =200
> abs(1+i);
    1.4142135623731
```

`arg` [Function]

`arg`(<constant or num. vec.>)

Return the argument (also called phase angle) of the constant if the value is a constant.

Return a numerical vector containing the argument (also called phase angle) of each element if the value is a numerical vector.

```
Example :
> arg(i);
    1.5707963267949
> arg(1+i);
    0.785398163397448
> t=0,pi,pi/6;
t    Tableau de reels : nb reels =7
>  writes(arg(exp(i*t)));
+0.000000000000000E+00
+5.235987755982987E-01
+1.047197551196598E+00
+1.570796326794897E+00
+2.094395102393195E+00
+2.617993877991494E+00
+3.141592653589793E+00
```

real [Function]

`real(` <constant or num. vec.> `)`

Return the real part of the constant if the value is a constant.

Return a numerical vector containing the real part of each element if the value is a numerical vector.

```
Example :
> real(i);
    0
> real(2+3*i);
    2
> t=0,pi,pi/6;
t    Tableau de reels : nb reels =7
> writes(real(exp(i*t)));
+1.000000000000000E+00
+8.660254037844387E-01
+5.000000000000000E-01
-0.000000000000000E+00
-4.999999999999998E-01
-8.660254037844385E-01
-1.000000000000000E+00
> real(4);
    4
```

imag [Function]

`imag(` <constant or num. vec.> `)`

Return the imaginary part of the constant if the value is a constant.

Return a numerical vector containing the imaginary part of each element if the value is a numerical vector.

```
Example :
> imag(i);
    1
```

```
> imag(2+3*i);
    3
> t=0,pi,pi/6;
> writes(real(exp(i*t)));
+1.000000000000000E+00
+8.660254037844387E-01
+5.000000000000000E-01
-0.000000000000000E+00
-4.999999999999998E-01
-8.660254037844385E-01
-1.000000000000000E+00
> imag(4);
    0
```

conj [Function]

conj(<constant or num. vec.>)

Return the conjugate of the constant if the value is a constant.

Return a numerical vector containing the conjugate of each element if the value is a numerical vector.

```
Example :
> conj(1+i);
    (1-i*1)
> conj(3);
    3
> z=vnumC[1+i:3-5*i:i];
z    vecteur de complexes : nb complexes =3
> writes(conj(z));
+1.000000000000000E+00  -1.000000000000000E+00
+3.000000000000000E+00  +5.000000000000000E+00
+0.000000000000000E+00  -1.000000000000000E+00
```

erf [Function]

erf(<real or real vec.>)

Return the error function of the constant if the value is a real.

Return a numerical vector containing the error function of each element if the value is a numerical vector of real numbers.

The error function is defined by :

$$erf(x) = 2/(\sqrt{\pi})\int_0^x \exp^{-t^2} dt$$

```
Example :
> erf(1);
        0.842700792949715
> erf(0.5);
        0.520499877813047
> t=-2,2;
t        Tableau de reels : nb reels =5
> r=erf(t);
r        Tableau de reels : nb reels =5
> writes(t,r);
-2.000000000000000E+00 -9.953222650189527E-01
```

```
-1.0000000000000000E+00 -8.4270079294971478E-01
+0.0000000000000000E+00 +0.0000000000000000E+00
+1.0000000000000000E+00 +8.4270079294971478E-01
+2.0000000000000000E+00 +9.9532226501895271E-01
```

`erfc` [Function]

`erfc(` <real or real vec.> `)`

Return the complementary error function of the constant if the value is a real.

Return a numerical vector containing the complementary error function of each element if the value is a numerical vector of real numbers.

The complementary error function is defined by :

$$erfc(x) = 1 - 2/(\sqrt{\pi}) \int_0^x \exp^{-t^2} dt$$

```
Example :
> erf(1);
           0.842700792949715
> erf(0.5);
           0.520499877813047
> t=-2,2;
t    Tableau de reels : nb reels =5
> r=erf(t);
r        Tableau de reels : nb reels =5
> writes(t,r);
-2.0000000000000000E+00 -9.9532226501895271E-01
-1.0000000000000000E+00 -8.4270079294971478E-01
+0.0000000000000000E+00 +0.0000000000000000E+00
+1.0000000000000000E+00 +8.4270079294971478E-01
+2.0000000000000000E+00 +9.9532226501895271E-01
```

`exp` [Function]

`exp(` <constant or num. vec.> `)`

Return the base-e exponential of the constant if the value is a constant.

Return a numerical vector containing the base-e exponential of each element if the value is a numerical vector.

```
Example :
> exp(1.2);
3.32011692273655
> x = 1.2;
> exp(x);
3.32011692273655
> exp(2*i);
    (-0.416146836547142+i*0.909297426825682)
> t=-2,2;
t    Tableau de reels : nb reels =5
> r=exp(t);
r    Tableau de reels : nb reels =5
> writes(t,r);
-2.000000000000000E+00  +1.353352832366127E-01
-1.000000000000000E+00  +3.678794411714423E-01
+0.000000000000000E+00  +1.000000000000000E+00
```

```
+1.000000000000000E+00  +2.718281828459045E+00
+2.000000000000000E+00  +7.389056098930650E+00
```

fac [Function]

`fac(` <constant or num. vec.> `)`

Return the factorial of a non-negative integer of the constant if the value is a constant.

Return a numerical vector containing the factorial of each element if the value is a numerical vector.

```
Example :
> fac(10);
                   3628800
> t=1,5;
 t  double precision real vector : number of elements =5
> r=fac(t);
 r  double precision real vector : number of elements =5
> writes(t,r);
+1.0000000000000000E+00 +1.0000000000000000E+00
+2.0000000000000000E+00 +2.0000000000000000E+00
+3.0000000000000000E+00 +6.0000000000000000E+00
+4.0000000000000000E+00 +2.4000000000000000E+01
+5.0000000000000000E+00 +1.2000000000000000E+02
>
```

sqrt [Function]

`sqrt(` <constant or num. vec.> `)`

Return the square root of the constant if the value is a constant.

Return a numerical vector containing the square root of each element if the value is a numerical vector.

```
Example :
> x = 4;
> sqrt(x);
2
> sqrt(1+i);
(1.09868411346781+i*0.455089860562227)
> t=0,5;
t    Tableau de reels : nb reels =6
> ts=sqrt(t);
ts   Tableau de reels : nb reels =6
> writes(t,ts);
+0.000000000000000E+00  +0.000000000000000E+00
+1.000000000000000E+00  +1.000000000000000E+00
+2.000000000000000E+00  +1.414213562373095E+00
+3.000000000000000E+00  +1.732050807568877E+00
+4.000000000000000E+00  +2.000000000000000E+00
+5.000000000000000E+00  +2.236067977499790E+00
```

cos [Function]

`cos(` <constant or num. vec.> `)`

Return the cosine of the constant if the value is a constant.

Return a numerical vector containing the cosine of each element if the value is a numerical vector.

```
Example :
> cos(0.12);
 0.992808635853866
> x=0;
x = 0
> cos(x);
>t=-pi,pi,pi/100;
> at=cos(t);
> cos(1+i);
 (0.833730025131149-i*0.988897705762865)
```

sin [Function]

`sin(` <constant or num. vec.> `)`

Return the sine of the constant if the value is a constant.

Return a numerical vector containing the sine of each element if the value is a numerical vector.

```
Example :
> sin(0.12);
0.119712207288912
> x = 0.12;
> sin(x);
0.119712207288912
> t=-pi,pi,pi/100;
> s=sin(t);
> sin(4*i);
(0+i*27.2899171971277)
```

tan [Function]

`tan(` <constant or num. vec.> `)`

Return the tangent of the constant if the value is a constant.

Return a numerical vector containing the tangent of each element if the value is a numerical vector.

```
Example :
> x = 1.2;
> tan(x);
2.57215162212632
> t=-pi,pi,pi/100;
> at=tan(t);
> tan(-1+i);
(-0.271752585319512+i*95227/87854)
```

acos [Function]

`acos(` <real or real vec.> `)`

Return the principal value of the arc cosine of the real number if the value is a real number.

Return a real vector containing the principal value of the arc cosine of each element if the value is a real vector.

```
Example :
> acos(0.12);
1.4505064440011
> x = 0.12;
> acos(x);
1.4505064440011
> t=-pi,pi,pi/100;
> at=acos(t);
```

asin [Function]

`asin(` <real or real vec.> `)`

Return the principal value of the arc sine of the real number if the value is a real number.

Return a real vector containing the principal value of the arc sine of each element if the value is a real vector.

```
Example :
> asin(0.12);
0.120289882394788
> x = 0.12;
> asin(x);
0.120289882394788
> t=-pi,pi,pi/100;
> at=asin(t);
```

atan [Function]

`atan(` <real or real vec.> `)`

Return the principal value of the arc tangent of the real number if the value is a real number.

Return a real vector containing the principal value of the arc tangent of each element if the value is a real vector.

```
Example :
> atan(2);
1.10714871779409
> x = 2;
> atan(x);
1.10714871779409
> t=-pi,pi,pi/100;
> at=atan(t);
```

cosh [Function]

`cosh(` <constant or num. vec.> `)`

Return the hyperbolic cosine of the constant if the value is a constant.

Return a numerical vector containing the hyperbolic cosine of each element if the value is a numerical vector.

```
Example :
> cosh(0.12);
1.0072086414827
> x = 0.12;
> cosh(x);
1.0072086414827
> t=-pi,pi,pi/100;
> at=cosh(t);
> cosh(1+i);
(0.833730025131149+i*0.988897705762865)
```

sinh [Function]

`sinh(` <constant or num. vec.> `)`

Return the hyperbolic sine of the constant if the value is a constant.

Return a numerical vector containing the hyperbolic sine of each element if the value is a numerical vector.

```
Example :
> sinh(1.2);
1.50946135541217
> x = 1.2;
> sinh(x);
1.50946135541217
> t=-pi,pi,pi/100;
> s=sinh(t);
> sinh(-1+3*i);
(1.16344036370325+i*0.217759551622152)
```

tanh [Function]

`tanh(` <constant or num. vec.> `)`

Return the hyperbolic tangent of the constant if the value is a constant.

Return a numerical vector containing the hyperbolic tangent of each element if the value is a numerical vector.

```
Example :
> x = 1.2;
> tanh(x);
0.833654607012155
> tanh(1+i);
(95227/87854+i*0.271752585319512)
> t=0,10;
t    Tableau de reels : nb reels =11
> tanh(t);
```

acosh [Function]

`acosh(` <constant or num. vec.> `)`

Return the inverse of the hyperbolic cosine of the constant if the value is a constant.

Return a numerical vector containing the inverse of the hyperbolic cosine of each element if the value is a numerical vector.

```
Example :
> acosh(1.2);
             0.6223625037147786
> t=1,2,0.1;
t        Tableau de reels double-precision : nb reels =11
> acosh(t);
         Tableau de reels  double-precision : nb reels =11
```

asinh [Function]

`asinh(` <constant or num. vec.> `)`

Return the inverse of the hyperbolic sine of the constant if the value is a constant.

Return a numerical vector containing the inverse of the hyperbolic sine of each element if the value is a numerical vector.

```
Example :
> asinh(1.2);
              1.015973134179692
> t=1,2,0.1;
t        Tableau de reels double-precision : nb reels =11
> asinh(t);
         Tableau de reels  double-precision : nb reels =11
```

asinh [Function]

asinh(<constant or num. vec.>)

Return the inverse of the hyperbolic tangent of the constant if the value is a constant.

Return a numerical vector containing the inverse of the hyperbolic tangent of each element if the value is a numerical vector.

```
Example :
> atanh(0.2);
             0.2027325540540822
> t=0,1,0.1;
t        Tableau de reels double-precision : nb reels =11
> atanh(t);
         Tableau de reels  double-precision : nb reels =11
```

atan2 [Function]

atan2(<real or real vec.> , <real or real vec.>)

Return the principal value of the arc tangent of the first argument divided by the second argument using the signs of both arguments to determine the quadrant of the return value.

```
Example :
> y = 11.5;
> x = 14;
> atan2(y,x);
0.68767125603875
```

mod [Function]

mod(<real or real vec.> , <real or real vec.>)

<real or real vec.> mod <real or real vec.>

Return the modulo of the first and second value. The result has the same sign as the first argument nd magnitude less than the magnitude of the second argument.

```
Example :
> mod(4,2);
0
> x=4$
> y=2$
> mod(x,y);
0
> t=21,30;
> r=1,10;
> f=mod(t,r);
> g=mod(t,3);
```

int [Function]

int(<real or real vec.>)

Return the integer part of the real number if the value is a real.

Return a real vector containing the integer part of each element if the value is a real vector.

```
Example :
>x=1.23;
>int(x);
1
> t=-pi,pi,pi/100;
> at=int(t);
```

`log` [Function]

`log(` <constant or num. vec.> `)`

Return the neperian logarithm of the constant if the value is a constant.

Return a numerical vector containing the neperian logarithm of each element if the value is a numerical vector.

```
Example :
> log(3);
1.09861228866811
> x = 3;
> log(x);
1.09861228866811
> r=1,10;
> at=log(r);
> log(2+i);
(0.80471895621705+i*0.463647609000806)
```

`log10` [Function]

`log10(` <constant or num. vec.> `)`

Return the decimal logarithm (base 10) of the constant if the value is a constant.

Return a numerical vector containing the decimal logarithm of each element if the value is a numerical vector.

```
Example :
> log(3);
1.09861228866811
> x = 3;
> log(x);
1.09861228866811
> r=1,10;
> at=log(r);
> log(2+i);
(0.80471895621705+i*0.463647609000806)
```

`nint` [Function]

`nint(` <real or real vec.> `)`

Return the nearest integer of the real number if the value is a real.

Return a real vector containing the nearest integer of each element if the value is a real vector.

```
Example :
>  x=1.23;
x =                 1.23
> nint(x);
                       1
> nint(1.78);
                       2
```

```
> t=-1,1,0.1;
t    Tableau de reels : nb reels =21
> nint(t);
```

sign [Function]

`sign(` <real or real vec.> `)`

Return the sign of the real number if the value is a real.

Return a real vector containing the sign of each element if the value is a real vector.

The function is defined with the following rules :

sign(x) = +1 if x > 0

sign(x) = 0 if x = 0

sign(x) = -1 if x < 0

```
Example :
> sign(3);
        1
> sign(-3);
        -1
> sign(0);
        0
> t=-3,3;
t        Tableau de reels : nb reels =7
> writes(t,sign(t));
-3.0000000000000000E+00 -1.0000000000000000E+00
-2.0000000000000000E+00 -1.0000000000000000E+00
-1.0000000000000000E+00 -1.0000000000000000E+00
+0.0000000000000000E+00 +0.0000000000000000E+00
+1.0000000000000000E+00 +1.0000000000000000E+00
+2.0000000000000000E+00 +1.0000000000000000E+00
+3.0000000000000000E+00 +1.0000000000000000E+00
```

histogram [Function]

`histogram(`<real vec.> *TY* `,` <real vec.> *TX*`)`

Return the histogram of *TY* with the specified ranges *TX* :

TZ=histogram(TY,TX) : TZ[N] contains the number of elements in TY such that TX[N]<=TY[j]<TX[N+1].

For the last range TX, the relation is : TX[N]<=TY[j]<=TX[N+1]

```
Example :
> TY=vnumR[0:1:4:5:6:9:10:11:-1:-2:-5]$
> TX=-3,9,2$
> writes("%g\n",TX);
-3
-1
1
3
5
7
9
> TZ=histogram(TY,TX)$
>  writes("%g\n",TZ);
1
```

```
2
1
1
2
1
```

10.11 Conditions

`?()::` [Operator]

`?(<condition>)`

Return a numerical vector which contains only the numbers 0 or 1. For each element of the array : if the condition at index j is true then <name> [j]=1 else <name> [j]=0

`?(<condition>):` <constant or num. vec.> *tabvrai* `:` <constant or num. vec.> *tabfaux*

Return a numerical vector which contains only the numbers 0 or 1. For each element of the array : if the condition at index j is true then <name> [j]=*tabvrai*[j] else <name> [j]=*tabfaux*[j]

All vectors must have the same size.

```
Example :
> t=0,5;
 t  double precision real vector : number of elements =6
> r=5-t;
 r  double precision real vector : number of elements =6
> q=?(t<=r);
 q  double precision real vector : number of elements =6
> writes(q);
+1.0000000000000000E+00
+1.0000000000000000E+00
+1.0000000000000000E+00
+0.0000000000000000E+00
+0.0000000000000000E+00
+0.0000000000000000E+00
> l= ?(t<=r):i:5;
 l  Double precision complex vector : number of elements =6
> writes(l);
+0.0000000000000000E+00 +1.0000000000000000E+00
+0.0000000000000000E+00 +1.0000000000000000E+00
+0.0000000000000000E+00 +1.0000000000000000E+00
+5.0000000000000000E+00 +0.0000000000000000E+00
+5.0000000000000000E+00 +0.0000000000000000E+00
+5.0000000000000000E+00 +0.0000000000000000E+00
> m = ?(t>2):r:t;
 m  double precision real vector : number of elements =6
> writes(m);
+0.0000000000000000E+00
+1.0000000000000000E+00
+2.0000000000000000E+00
+2.0000000000000000E+00
+1.0000000000000000E+00
+0.0000000000000000E+00
>
```

10.12 Conversion

For the conversion from or to a numerical matrix, see Section 11.10 [ConversionMat], page 109.

10.12.1 dimtovnumR

`dimtovnumR` [Function]

`dimtovnumR`(<array>)

Return a numerical vector of real numbers from the array of series.

The array of series must contain only real numbers. The array must have only one dimension.

```
Example :
> dim ts[1:3];
> ts[1]=1$
> ts[2]=4$
> ts[3]=6$
> tr=dimtovnumR(ts)$
> writes(tr);
+1.000000000000000E+00
+4.000000000000000E+00
+6.000000000000000E+00
> ltr=log(dimtovnumR(ts));
ltr  Tableau de reels : nb reels =3
```

10.12.2 dimtovnumC

`dimtovnumC` [Function]

`dimtovnumC`(<array>)

Return a numerical vector of complex numbers from the array of series.

The array of series must contain only complex numbers. The array must have only one dimension.

```
Example :
> dim ts[1:3];
> ts[1]=1+i$
> ts[2]=3*i$
> ts[3]=-5+i$
> tc=dimtovnumC(ts)$
> writes(tc);
+1.000000000000000E+00  +1.000000000000000E+00
+0.000000000000000E+00  +3.000000000000000E+00
-5.000000000000000E+00  +1.000000000000000E+00
> ltc=exp(dimtovnumC(ts));
ltc  vecteur de complexes : nb complexes =3
```

10.12.3 vnumtodim

`vnumtodim` [Function]

`vnumtodim`(<(array of) num. vec.>)

Return an array of series (constants) from the array of numerical vectors or from the numerical vector.

```
Example :
> tr=1,4;
```

```
tr   Tableau de reels : nb reels =4
> tsr=vnumtodim(tr);

tsr [1:4]   nb elements = 4

> afftab(tsr);
tsr[1] = 1
tsr[2] = 2
tsr[3] = 3
tsr[4] = 4
> vnumC tc[1:2];
> tc[1]=i*tr$
> tc[2]=tr+i*tr**2$
> tsc=vnumtodim(tc);

tsc [1:4, 1:2]  nb elements = 8

> afftab(tsc);
tsc[1,1] = (0+i*1)
tsc[1,2] = (1+i*1)
tsc[2,1] = (0+i*2)
tsc[2,2] = (2+i*4)
tsc[3,1] = (0+i*3)
tsc[3,2] = (3+i*9)
tsc[4,1] = (0+i*4)
tsc[4,2] = (4+i*16)
```

10.12.4 str

`str` [Function]

`str(` <real vec.> `);`

`str(` <string> *format*, <real vec.> `);`

Convert an vector of real numbers in an array of character strings. If *format* is specified, then each element of the vector is converted to this format.

For the description of the formats, see Section 7.6 [str], page 33.

```
Example :
> t = 8,10;
 t  double precision real vector : number of elements =3
> tabs =  str("%04g", t);

tabs [1:3 ] number of elements = 3

> afftab(tabs);
tabs[1] =  "0008"
tabs[2] =  "0009"
tabs[3] =  "0010"
```

11 Numerical matrices

The numerical values stored in matrices are always real or complex double-precision, quadruple-precision or multiprecision numbers depending on the current numerical mode. A numerical matrix is assumed to be a two-dimensional matrix. The numerical matrix are not resizeable. The first dimension is the lines of the matrix and the second dimension is the columns of the matrix.

11.1 Declaration

Explicit declaration for numerical matrices are only required before call to `read` , `readbin` (see Section 10.8 [Entree/SortieTabNum], page 63) and assignment of a single or several elements. Explicit declaration for array of numerical matrices are always required.

11.1.1 matrixR

`matrixR` [Procedure]

`matrixR` <name> `([` <two dimensions of a matrix> *MxN* `])` `,` ... `;`

`matrixR` <name> `[` <dimension of an array> `]` `([` <two dimensions of a matrix> *MxN* `])` `,` ... `;`

It declares a real matrix of dimension *MxN* or an array of real matrices of dimension *MxN*.

After the declaration, the elements of the matrix are initialized with the value 0.

```
Example :
> matrixR C([1:3, 1:5]);
> stat(C);
 Double precision real matrix C [ 1:3 , 1:5 ].
  Size of the array in bytes: 120
> bounds= 1:3,1:6;
bounds = bounds [ 1:3, 1:6 ]
> matrixR T0[1:2]([bounds]);
> stat(T0);
  Array of series
 T0 [ 1:2  ]
 list of array's elements :
T0 [ 1 ] =
 Double precision real matrix T0 [ 1:3 , 1:6 ].
  Size of the array in bytes: 144
T0 [ 2 ] =
 Double precision real matrix T0 [ 1:3 , 1:6 ].
  Size of the array in bytes: 144
>
```

11.1.2 matrixC

`matrixC` [Procedure]

`matrixC` <name> `([` <two dimensions of a matrix> *MxN* `])` `,` ... `;`

`matrixC` <name> `[` <dimension of an array> `]` `([` <two dimensions of a matrix> *MxN* `])` `,` ... `;`

It declares a complex matrix of dimension *MxN* or an array of complex matrices of dimension *MxN* .

After the declaration, the elements of the matrix are initialized with the value 0+i*0.

```
Example :
> matrixC C([1:3, 1:5]);
> stat(C);
 Double precision complex matrix C [ 1:3 , 1:5 ].
  Size of the array in bytes: 240
> bounds=1:3,1:6;
bounds = bounds [ 1:3, 1:6 ]
> matrixC T0[1:2]([bounds]);
> stat(T0);
  Array of series
 T0 [ 1:2  ]
 list of array's elements :
T0 [ 1 ] =
 Double precision complex matrix T0 [ 1:3 , 1:6 ].
  Size of the array in bytes: 288
T0 [ 2 ] =
 Double precision complex matrix T0 [ 1:3 , 1:6 ].
  Size of the array in bytes: 288
>
```

11.2 Initialization

`matrixR[,,:,,]` [Function]

<name> = `matrixR [` <real> or <real vec.> or <array of real vec.> `,` ... `:` <real> `,`... `]` `;`

It declares and initializes a real matrix with the specified reals or real vectors.

The character `:` separates the lines and the character `,` separates the columns. All columns must have the same line counts.

```
Example :
> // declare a real matrix 2x3
> tab3=matrixR[1,2,3:4,5,6];
tab3  double precision real matrix [ 1:2 , 1:3 ]
> writes(tab3);
+1.0000000000000000E+00 +2.0000000000000000E+00 +3.0000000000000000E+00
+4.0000000000000000E+00 +5.0000000000000000E+00 +6.0000000000000000E+00
>
```

`matrixC[,,:,,]` [Function]

<name> = `matrixC [` <complex> or <complex vec.> `,` ... `:` <complex> `,`... `]` `;`

It declares and initializes a complex matrix with the specified complex numbers or complex vectors.

The character `:` separates the lines and the character `,` separates the columns. All columns must have the same line counts.

```
Example :
> // declare a complex matrix 2x3
> tab3=matrixC[1+2*i,3+4*i,5:2,4*i, -7+2*i];
tab3  double precision complex matrix [ 1:2 , 1:3 ]
> writes(tab3);
+1.0000000000000000E+00 +2.0000000000000000E+00 +3.0000000000000000E+00 +4.00000000000
+5.0000000000000000E+00 +0.0000000000000000E+00
+2.0000000000000000E+00 +0.0000000000000000E+00 +0.0000000000000000E+00 +4.00000000000
-7.0000000000000000E+00 +2.0000000000000000E+00
```

```
>
```

11.3 Display

`writes` [Procedure]

`writes(`[<integer> : <integer> : <integer>] , <string> , <(array of) matrix> , ...`);`

`writes(` <string> , <(array of) matrix> , ...`);`

`writes(` <(array of) matrix> , ...`);`

equivalent to

writes({ [*binf*:{*bsup*}:{*step*}], } {*format*,} <(array of) matrix> ,...).

It prints, to the screen, numerical matrices or the array of numerical matrices in columns form.

From the *binf*th line of each numerical matrix (if *binf* isn't specified, it starts from the first line).

To the *bsup*th line of each numerical matrix (if *bsup* isn't specified, it stops to the last line of the biggest matrix).

Every *step* line (if *step* isn't specified, the step is 1).

format is optional. This format is similar to the C format (cf. printf) and must be between double quotes ("). The format must contain the same number of format specifiers (e.g., %g) as the number of columns.

A complex matrix will be printed on two times more columns (the first for the real part and the second for the imaginary part).

```
Example :
> // display a real matrix 2x3
> mat3=matrixR[1,2,3:4,5,6];
mat3  double precision real matrix [ 1:2 , 1:3 ]
> writes(mat3);
+1.0000000000000000E+00 +2.0000000000000000E+00 +3.0000000000000000E+00
+4.0000000000000000E+00 +5.0000000000000000E+00 +6.0000000000000000E+00
> // display a complex matrix 2x3
> mat4=matrixC[1+2*i,3+4*i,5:2,4*i, -7+2*i];
mat4  double precision complex matrix [ 1:2 , 1:3 ]
> writes(6*"%g "+"\n", mat4);
1 2 3 4 5 0
2 0 0 4 -7 2
>
```

`afftab` [Procedure]

`afftab(` <matrix> `);`

It prints, to the screen, a numerical matrix. Each line of the matrix is surrounded by the characters [].

```
Example :
> _affc=1$
> // display a real matrix 2x3
> mat3=matrixR[1,2,3:4,5,6];
mat3  double precision real matrix [ 1:2 , 1:3 ]
> afftab(mat3);
[   1    2    3]
[   4    5    6]
```

```
> // display a complex matrix 2x3
> mat4=matrixC[1+2*i,3+4*i,5:2,4*i, -7+2*i];
mat4  double precision complex matrix [ 1:2 , 1:3 ]
> afftab(mat4);
[(1+i*2) (3+i*4)    5]
[   2 (0+i*4) (-7+i*2)]
>
```

11.4 Size

`size` [Function]

`size(` <matrix> `,` <integer> *n* `)`

Return the number of lines of the numerical matrix if *n*=1.

Return the number of columns of the numerical matrix if *n*=2.

```
Example :
> mat3=matrixR[1,2,3:4,5,6];
mat3  double precision real matrix [ 1:2 , 1:3 ]
> size(mat3,1);
 2
> size(mat3,2);
 3
>
```

11.5 Data retrieval

`[::,::]` [Operator]

<matrix> `[` <integer> *binfli* `:` <integer> *bsupli* `:` <integer> *pasli* `,`
<integer> *binfcol* `:` <integer> *bsupcol* `:` <integer> *pascol* `];`

Return a numerical matrix which contains only the elements located between the lower bound and the upper bound with the specified step.

If the lower bound isn't specified, then the lower bound is assumed to be 1.
If the upper bound isn't specified, then the upper bound is assumed to be the size of the numerical vector.
If the step isn't specified, then the step is assumed to be 1.

Remark : All missing combinations are allowed.

```
Example :
> M=matrixR[1,2,3:4,5,6:7,8,9];
M  double precision real matrix [ 1:3 , 1:3 ]
> r=M[::2,::2];
r  double precision real matrix [ 1:2 , 1:2 ]
> writes(r);
+1.0000000000000000E+00 +3.0000000000000000E+00
+7.0000000000000000E+00 +9.0000000000000000E+00
> v=M[1:2,2:3];
v  double precision real matrix [ 1:2 , 1:2 ]
> writes(v);
+2.0000000000000000E+00 +3.0000000000000000E+00
+5.0000000000000000E+00 +6.0000000000000000E+00
>
```

11.6 Input/Output

The following functions work on the numerical matrices in the same behavior as on the numerical vectors :

- `write` (see Section 10.8.3 [write], page 68)
- `writebin` (see Section 10.8.5 [writebin], page 70)

11.6.1 sauve_c

`sauve_c` [Procedure]

`sauve_c(`<matrix> `,` <filename> `);`

It saves the matrix to the file using the C language (version C99). The file is created in the directory specified by `_path`.

`sauve_c(`<matrix> `,` <file> `);`

It saves the matrix to the file, previously opened in written mode, using the C language (version C99).

```
Example :
> A=matrixR[9,0,7
           :1,2,3
           :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> sauve_c(A, "prog.c");
>
```

11.6.2 sauve_fortran

`sauve_fortran` [Procedure]

`sauve_fortran(`<matrix> `,` <filename> `);`

It saves the matrix to the file using the Fortran language (version 77 or later). The file source format is compatible with the fixed and free format. The file is created in the directory specified by `_path`.

`sauve_fortran(`<matrix> `,` <file> `);`

It saves the matrix to the file, previously opened in written mode, using the Fortran language (version 77 or later). The file source format is compatible with the fixed and free format.

```
Example :
> A=matrixR[9,0,7
           :1,2,3
           :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> sauve_fortran(A, "prog.f");
>
```

11.6.3 sauve_tex

`sauve_tex` [Procedure]

`sauve_tex(`<matrix> `,` <filename> `);`

It saves the matrix to the file using the TEX form. The file is created in the directory specified by `_path`.

`sauve_tex(`<serie> `,` <file> `);`

It saves the matrix to the file, previously opened in written mode, using the TEX form.

```
Example :
> A=matrixR[9,0,7
           :1,2,3
           :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> sauve_tex(A, "essai.tex");
>
```

11.6.4 sauve_ml

`sauve_ml` [Procedure]

`sauve_ml(`<matrix>` , `<filename>` );`

It saves the matrix to the file using the MathML 2.0 (concept) form. The file is created in the directory specified by `_path`.

`sauve_ml(`<matrix>` , `<file>` );`

It saves the matrix to the file, previously opened in written mode, using the MathML 2.0 (concept) form.

```
Example :
> A=matrixR[9,0,7
           :1,2,3
           :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> sauve_ml(A, "essai.ml");
>
```

11.6.5 write

This function works on the numerical matrices in the same behavior as on the numerical vectors (see Section 10.8.3 [write], page 68).

11.6.6 writebin

This function works on the numerical matrices in the same behavior as on the numerical vectors (see Section 10.8.5 [writebin], page 70).

11.7 Input/Output low level

The following functions work on the numerical matrices in the same behavior as on the numerical vectors :

– `file_write` (see Section 10.9.3 [file_write], page 72)

11.8 Standard math function

The following functions work on the numerical matrices in the same behavior as on the numerical vectors. These operations apply on each element of the matrix.

- `abs`
- `acos`
- `acosh`
- `arg`
- `asin`
- `asinh`
- `atan`

- `atanh`
- `atan2`
- `conj`
- `cos`
- `cosh`
- `erf`
- `erfc`
- `exp`
- `imag`
- `int`
- `log`
- `log10`
- `MAX`
- `MIN`
- `mod`
- `nint`
- `real`
- `sign`
- `sin`
- `sinh`
- `tan`
- `tanh`

The following functions work on the numerical matrices in the same behavior as on the numerical vectors. These operations apply on the matrix.

- `max`
- `min`
- `sum`

11.8.1 Matrix product

`&*` [Operator]

<matrix> `&*` <matrix>

Compute the matrix product of the numerical matrix.

Compute r=a&*b such that $r[i,j] = \sum_{k=1}^{size(a,2)} a_{i,k} \times b_{k,j}$

Remark : the number of columns of the first matrix must be same as the number of lines of the second matrix.

```
Example :
> _affc=1$
> A = matrixR[1,3,5
             :2,4,6];
A  double precision real matrix [ 1:2 , 1:3 ]
> B = matrixR[5,8,11
             :7,6,8
             :4,0,8];
B  double precision real matrix [ 1:3 , 1:3 ]
```

```
> C = A&*B;
C  double precision real matrix [ 1:2 , 1:3 ]
> afftab(C);
[   46    26    75]
[   62    40    102]
>
```

`&*` [Operator]

<matrix> `&*` <num. vec.>

<num. vec.> `&*` <matrix>

Compute the matrix product of the numerical matrix and a numerical vector.

If b is the numcerial vector, i computes r=a&*b such that $r[i] = \sum_{k=1}^{size(a,2)} a_{i,k} \times b_k$

11.8.2 Kronecker product

`kroneckerproduct` [Function]

`kroneckerproduct(`<matrix> A `,`<matrix> B`)`

It returns the Kronecker prouct of two numerical matrices.

```
Example :
> _affc=1$
> A = matrixR[1,2,3:4,5,6]$
> B = matrixR[5,0:10,1:7,3]$
> C = kroneckerproduct(A,B)$
> afftab(C);
[   5    0    10    0    15    0]
[   10    1    20    2    30    3]
[   7    3    14    6    21    9]
[   20    0    25    0    30    0]
[   40    4    50    5    60    6]
[   28    12    35    15    42    18]
>
```

11.8.3 Matrix determinant

`det` [Function]

`det(` <matrix> `)`

Computes the determinant of a square matrix.

Remarks : When computations are performed in double precision, the Lapack Library is used. The LU algorithm is used in all numerical precision.

```
Example :
> t=matrixR[9,0,7
           :1,2,3
           :4,5,6];
t  double precision real matrix [ 1:3 , 1:3 ]
> det(t);
                        -48
>
```

11.8.4 Matrix inversion

`invertmatrix` [Function]

`invertmatrix(` <matrix> `)`

Computes the inverse of a square matrix.

Remarks : When computations are performed in double precision, the Lapack Library is used. The LU algorithm is used in all numerical precision.

```
Example :
> _affc=2$
> A=matrixR[9,0,7
          :1,2,3
          :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> B=invertmatrix(A);
B  double precision real matrix [ 1:3 , 1:3 ]
> afftab(B);
[   0.0625  - 0.72916667    0.29166667]
[ - 0.125  - 0.54166667    0.41666667]
[   0.0625    0.9375  - 0.375]
>
```

11.8.5 Matrix trace

`tracematrix` [Function]

`tracematrix( <matrix> )`

Computes the trace of a square matrix.

```
Example :
> A=matrixR[9,0,7
          :1,2,3
          :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> t=tracematrix(A);
t =                         17
>
```

11.8.6 Transpose

`transposematrix` [Function]

`transposematrix( <matrix> )`

Computes the transpose of a square matrix.

```
Example :
> _affc=1$
> A=matrixR[9,0,7
          :1,2,3];
A  double precision real matrix [ 1:2 , 1:3 ]
> B=transposematrix(A);
B  double precision real matrix [ 1:3 , 1:2 ]
> afftab(B);
[   9    1]
[   0    2]
[   7    3]
>
>
```

11.8.7 Identity matrix

`identitymatrix` [Function]

`identitymatrix(` <operation> *n* `)`

Computes the identity matrix of size *n*. This matrix is a *n*x*n* square matrix with ones on the main diagonal and zeros elsewhere.

```
Example :
> _affc=1$
> I3=identitymatrix(3);
I3  double precision real matrix [ 1:3 , 1:3 ]
> afftab(I3);
[   1    0    0]
[   0    1    0]
[   0    0    1]
>
```

11.8.8 Eigenvalues

`eigenvalues` [Function]

`eigenvalues(`<matrix> `)`

Computes the eigenvalues of the square matrix using a QR algorithm (lapack library).

```
Example :
> A=matrixR[9,0,7
          :1,2,3
          :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> B=eigenvalues(A);
 B  Double precision complex vector : number of elements =3
> writes(B);
+1.3769150418957313E+01 +0.0000000000000000E+00
+4.0843620348094420E+00 +0.0000000000000000E+00
-8.5351245376675389E-01 +0.0000000000000000E+00
```

11.8.9 Eigenvectors

`eigenvectors` [Procedure]

`eigenvectors(`<matrix> *MAT*, <matrix> *TVECT*, <matrix> *TVAL*)

`eigenvectors(`<matrix> *MAT*, <matrix> *TVECT*)

Compute the eigenvectors of the square matrix *MAT* using a QR algorithm (lapack library). It stores the eigenvectors in the matrix *TVECT* and the eigenvalues in the vector *TVAL*.

```
Example :
> _affc=1$
> A=matrixR[9,0,7
          :1,2,3
          :4,5,6];
A  double precision real matrix [ 1:3 , 1:3 ]
> eigenvectors(A,vectp,valp);
> afftab(vectp);
[ - 0.808169  - 0.750577  - 0.484664]
[ - 0.209021    0.398522  - 0.547409]
[ - 0.550611    0.527081    0.682234]
```

```
> writes(valp);
+1.3769150418957313E+01 +0.0000000000000000E+00
+4.0843620348094420E+00 +0.0000000000000000E+00
-8.5351245376675389E-01 +0.0000000000000000E+00
> //  first vector
> p=vectp[:,1]$
> writes(p);
-8.0816903814944341E-01 +0.0000000000000000E+00
-2.0902130660832199E-01 +0.0000000000000000E+00
-5.5061138669696397E-01 +0.0000000000000000E+00
```

11.8.10 Arithmetic

The matrices must have the same number of elements on each dimension.

`+` [Operator]

<matrix> + <matrix>

Add term by term two matrices.

`+` [Operator]

<matrix> + <constant>

<constant> + <matrix>

Add a constant with each element of the matrix.

`+` [Operator]

<matrix> + <num. vec.>

<num. vec.> + <matrix>

Add term by term the numerical vector and the matrix. The matrix must contain only one column.

`*` [Operator]

<matrix> * <matrix>

Multiply term by term two matrices.

`*` [Operator]

<matrix> * <constant>

<constant> * <matrix>

Multiply a constant with each element of the matrix.

`*` [Operator]

<matrix> * <num. vec.>

<num. vec.> * <matrix>

Multiply term by term the numerical vector and the matrix. The matrix must contain only one column.

`-` [Operator]

<matrix> - <matrix>

Substract term by term two matrices.

`-` [Operator]

<matrix> - <constant>

<constant> - <matrix>

Subtract a numerical constant with each element of the matrix.

`-` [Operator]

<matrix> - <num. vec.>

<num. vec.> - <matrix>

Subtract term by term the numerical vector and the matrix. The matrix must contain only one column.

`/` [Operator]

<matrix> / <matrix>

Divide term by term two matrices.

`/` [Operator]

<matrix> / <constant>

<constant> / <matrix>

Divide a numerical constant with each element of the matrix.

`/` [Operator]

<matrix> / <num. vec.>

<num. vec.> / <matrix>

Divide term by term the numerical vector and matrix. The matrix must contain only one column.

11.9 Conditions

`?()::` [Operator]

`?(`<condition>`)`

Return a real matrix which contains only the numbers 0 or 1. For each element of the array : if the condition at index i,j is true then <name> [i,j]=1 else <name> [i,j]=0

`?(`<condition>`):` <constant or matrix> *tabvrai* : <constant or matrix> *tabfaux*

Return a numerical vector which contains only the numbers 0 or 1.
For each element of the condition : if the condition at index i,j is true then <name> [i,j]=*tabvrai*[i,j] else <name> [i,j]=*tabfaux*[j]

All matrix must have the same size.

```
Example :
> mat1=matrixR[1,2,3:4,5,6:7,8,9];
mat1  double precision real matrix [ 1:3 , 1:3 ]
> mat2=matrixR[3,0,6:5,2,7:1,4,11];
mat2  double precision real matrix [ 1:3 , 1:3 ]
> q=?(mat1<=5);
q  double precision real matrix [ 1:3 , 1:3 ]
> writes(3*"%g "+"\n",q);
1 1 1
1 1 0
0 0 0
> m = ?(mat1>2):mat2:-1;
m  double precision real matrix [ 1:3 , 1:3 ]
> writes(3*"%g "+"\n",m);
-1 -1 6
5 2 7
1 4 11
>
```

11.10 Conversion

matrixR [Operator]

`matrixR( <object identifier> )`

Return a real matrix with the object identifier. The object identifier may be an array, a numerical vector ou an array of numerical vectors. The object must contain only real numbers.

```
Example :
> _affc=1$
> // convert an array of numbers to a real matrix
> tab3=[1,2,3
       :4,5,6];

tab3 [1:2, 1:3 ] number of elements = 6

> mat3=matrixR(tab3);
mat3  double precision real matrix [ 1:2 , 1:3 ]
> afftab(mat3);
[   1    2    3]
[   4    5    6]
> // convert an numerical vector to a real matrix
> v2= 1,10;
 v2  double precision real vector : number of elements =10
> mat2=matrixR(v2);
mat2  double precision real matrix [ 1:10 , 1:1 ]
>
```

matrixC [Operator]

`matrixC( <object identifier> )`

Return a complex matrix with the object identifier. The object identifier may be an array, a numerical vector ou an array of numerical vectors. The object must contain only complex numbers.

```
Example :
> _affc=1$
> // convert an array of numbers to a complex matrix
> tab3=[1+2*i,3+4*i,5
       :2,4*i,-7+2*i];

tab3 [1:2, 1:3 ] number of elements = 6

> mat3=matrixC(tab3);
mat3  double precision complex matrix [ 1:2 , 1:3 ]
> afftab(mat3);
[(1+i*2) (3+i*4)    5]
[   2 (0+i*4) (-7+i*2)]
> // convert an numerical vector to a complex matrix
> v2 = 1,10$
> v2 = exp(I*v2);
 v2  Double precision complex vector : number of elements =10
> mat2 = matrixC(v2);
mat2  double precision complex matrix [ 1:10 , 1:1 ]
```

```
>
```

vnumR [Operator]

`vnumR( <matrix> )`

Return a real vector from the matrix. The matrix must have a single column.

```
Example :
> // convert a real matrix to a real vector
> mat1=matrixR[2:3:5];
mat1  double precision real matrix [ 1:3 , 1:1 ]
> v1=vnumR(mat1);
 v1  double precision real vector : number of elements =3
> writes(v1);
+2.0000000000000000E+00
+3.0000000000000000E+00
+5.0000000000000000E+00
>
```

vnumC [Operator]

`vnumC( <matrix> )`

Return a real vector from the matrix. The matrix must have a single column.

```
Example :
> // convert a complex matrix to a complex vector
> mat1=matrixC[1+2*i:4*i:-7+2*i];
mat1  double precision complex matrix [ 1:3 , 1:1 ]
> v1=vnumC(mat1);
 v1  Double precision complex vector : number of elements =3
> writes(v1);
+1.0000000000000000E+00 +2.0000000000000000E+00
+0.0000000000000000E+00 +4.0000000000000000E+00
-7.0000000000000000E+00 +2.0000000000000000E+00
>
```

12 Graphics

TRIP requires the following versions of gnuplot to work :
- on Windows, version 3.7.3 or later.
- on UNIX or MacOS X, version 3.7.0 or later.
- on Android, the application droidplot.

TRIP requires the following versions of grace to work :
- On UNIX, MacOS X or Windows, version 5.1.8 or later.

The commands plot, replot, plotf, plotps, plotps_end and plotreset use grace or gnuplot depending on the global variable *_graph* (see Chapter 3 [_graph], page 9).

12.1 plot

`plot` [Procedure]

`plot(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`);`

`plot(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) string> *options*`);`

Executes gnuplot or grace and plots the contents of the numerical vector *TY* function of *TX*.

The string *options* is directly sent to gnuplot or grace as an argument of the command plot.

If the string *options* contains double-quotes, two double-quotes must be used.

Remarks : Temporary files are created and will be destroyed at the end of the session.

`plot` [Procedure]

`plot(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`);`

`plot(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`,` <(array of) string> *options*`);`

Executes gnuplot or grace and plots in 3D the contents of the numerical vector *TZ* function of *TY* and *TX*.

`plot` [Procedure]

`plot(`<string> *cmd*`,` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`);`

`plot(`<string> *cmd*`,` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) string> *options*`);`

It executes gnuplot or grace if necessary. It sends to gnuplot or grace the command *cmd* and plots the contents of the numerical vector *TY* function of *TX*.

`plot` [Procedure]

`plot(`<string> *cmd*`,` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`);`

`plot(`<string> *cmd*`,` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`,` <(array of) string> *options*`);`

It executes gnuplot or grace if necessary. It sends to gnuplot or grace the command *cmd* and plots in 3D the contents of the numerical vector *TZ* function of *TY* and *TX*.

`plot` [Procedure]

`plot( (`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`), ... );`

`plot( (`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) string> *options*`), ...);`

Executes gnuplot or grace and overlays all draws of each couplet with plotting the contents of the numerical vector *TY* function of *TX*.

`plot` [Procedure]

`plot(` `(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`)`, `...);`

`plot(` `(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`,` <(array of) string> *options*`)`, `...);`

Executes gnuplot or grace and overlays all draws of each triplet with plotting in 3D the contents of the numerical vector *TZ* function of *TY* and *TX*. session.

`plot` [Procedure]

`plot(`<string> *cmd*`,` `(` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`)`, `...);`

`plot(`<string> *cmd*`,` `(` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) string> *options*`)`, `...);`

It executes gnuplot or grace if necessary. It sends to gnuplot or grace the command *cmd* and plots the contents of each couplet of the numerical vector *TY* function of *TX*.

`plot` [Procedure]

`plot(`<string> *cmd*`,` `(` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`)`, `...);`

`plot(`<string> *cmd*`,` `(` <(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`,` <(array of) real vec.>*TZ*`,` <(array of) string> *options*`)`, `...);`

It executes gnuplot or grace if necessary. It sends to gnuplot or grace the command *cmd* and plots in 3D the contents of each triplet of the vectors *TZ* function of *TY* and *TX*.

```
Example :
Tracer cos(x) pour x=-pi à pi avec un pas de pi/100.
> x=-pi,pi,pi/100;
x    Tableau de reels : nb reels =200
> y=cos(x);
y    Tableau de reels : nb reels =200
> plot(x,y);
> plot(x,y,cos(x));
> plot(x,y,"notitle w points pt 5");
> plot(x,y,"title ""x,cos(x)"" ");
> t=1,10;
t        Tableau de reels : nb reels =10
>  plot("set xrange[2:5]",t,log(t));


> t=0,pi,pi/100;
> vnumR ta[1:3];
> ta[1]=cos(t);
> ta[2]=sin(t);
> ta[3]=cosh(t);
> dim nom[1:3];
> nom[1]="title 'cos' w l";
> nom[2]="title 'sin' w l";
> nom[3]="title 'cosh' w l";
> plot((t,ta,nom));
```

12.2 replot

`replot` [Procedure]

`replot(`<(array of) real vec.>*TX*`,` <(array of) real vec.>*TY*`);`

`replot(`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) string> *options*`);`

`replot(`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`);`

`replot(`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`, `<(array of) string> *options*`);`

`replot(`<string> *cmd*`, `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`);`

`replot(`<string> *cmd*`, `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) string> *options*`);`

`replot(`<string> *cmd*`, `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`);`

`replot(`<string> *cmd*`, `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`, `<(array of) string> *options*`);`

`replot( (`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`), ... );`

`replot( (`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) string> *options*`), ...);`

`replot( (`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`), ...);`

`replot( (`<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`, `<(array of) string> *options*`), ...);`

`replot(`<string> *cmd*`, ( `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`), ...);`

`replot(`<string> *cmd*`, ( `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) string> *options*`), ...);`

`replot(`<string> *cmd*`, ( `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`), ...);`

`replot(`<string> *cmd*`, ( `<(array of) real vec.>*TX*`, `<(array of) real vec.>*TY*`, `<(array of) real vec.>*TZ*`, `<(array of) string> *options*`), ...);`

This command is very similar to the command `plot` but it overlays the draws on the previous draws.

It uses the same arguments as the command `plot` (see Section 12.1 [plot], page 111).

`replot` [Procedure]

`replot;`

Executes gnuplot or grace and send a command to redraw all graphics.

```
Example :
Tracer cos(x) et sin(x) pour x=-pi à pi avec un pas de pi/100
dans la même fenêtre.
> x=-pi,pi,pi/100;
x    Tableau de reels : nb reels =200
> y=cos(x);
y    Tableau de reels : nb reels =200
> y1=sin(x);
y1   Tableau de reels : nb reels =200
> plot(x,y);
> replot(x,y1);
> plot(x,y,y1);
> replot(x,y,sin(2.*x));
> replot(x,sin(x),"notitle");
> replot(x,y,"w points pt 5");
```

12.3 plotf

`plotf` [Procedure]

`plotf(`<filename> *filename*`,` <integer> *ncolTX*`,` <integer> *ncolTY*`);`

It executes gnuplot or grace and plots the contents of the column *ncolTY* function of *ncolTX* in the file *filename*.

`plotf` [Procedure]

`plotf(`<filename> *filename*`,` <integer> *ncolTX*`,` <integer> *ncolTY*`,` <integer> *ncolTZ*`);`

It executes gnuplot or grace and plots the contents of the column *ncolTZ* function of *ncolTY* and *ncolTX* in the file *filename*.

gnuplot will ignore lines beginning by the character `#`.

```
Example :
Afficher la troisieme colonne en fonction de la première colonne du
fichier tab.out
> plotf(tab.out,1,3);
Afficher la troisieme colonne en fonction de la première et deuxième
colonne du fichier tab.out
> plotf(tab.out,1,2,3);
```

12.4 plotps

`plotps` [Procedure]

`plotps` <filename> `;`

It executes gnuplot or grace and set the terminal of gnuplot or grace in postscript.

All graphics will be stored in the specified postscript file. This file will be located in the folder specified by `_path`.

To close the postscript file, the command `plotps_end` must be executed.

```
Example :
Tracer cos(x) pour x=-pi à pi avec un pas de pi/100
et le stocker dans le fichier res.ps.

> x=-pi,pi,pi/100;
x    nb elements réels =200
> y=cos(x);
y    nb elements réels =200
> plotps res.ps;
> plot(x,y);
> plotps_end;
```

12.5 plotps_end

`plotps_end` [Procedure]

`plotps_end;`

Close the file created by `plotps`.

For gnuplot, it sets the terminal to its default value.

Remarks :

- on Unix, MacOS X or cygwin, the default terminal is x11.
- on Windows, the default terminal is windows.

```
Example :
Tracer cos(x) pour x=-pi à pi avec un pas de pi/100
et le stocker dans le fichier res.ps.

> x=-pi,pi,pi/100;
x    nb elements réels =200
> y=cos(x);
y    nb elements réels =200
> plotps res.ps;
> plot(x,y);
> plotps_end;
```

12.6 plotreset

`plotreset` [Procedure]

`plotreset;`

Send a command to reinitialize gnuplot or grace.

For gnuplot, it sends the command `reset`.

For grace, it sends the command `new` followed with `redraw`.

```
Example :
> _graph=grace;
                _graph      = grace
> t=0,10;
t        Tableau de reels : nb reels =11
> plot(t,t);
> plotreset;
```

12.7 gnuplot

`gnuplot` [Procedure]

`gnuplot;`

<gnuplot command>

<gnuplot command>`@`<string> `@`<gnuplot command>

`%`<trip command> \

<trip command>

`end;`

An alternative syntax to `gnuplot; ... end;` is `gnuplot; ... gnuplot_end;` .

TRIP takes the gnuplot commands and sends them to gnuplot (line after line). Gnuplot is executed if it wasn't started. The prompt becomes '`gnuplot>`' when the user could enter the gnuplot commands.

When the first character is %, then the end of this line is one or more commands trip. This trip command could follow on several lines : the last character must be `\` to indicates that the command follows on the next line.

Strings declared in trip could be send to gnuplot by surrounding with the character @.

```
Example :
> gnuplot;
gnuplot> plot 'tftf' using 1:3
gnuplot> set xrange[1:10]
gnuplot> replot
gnuplot> end$
> >
> gnuplot;
gnuplot> set terminal macintosh singlewin
Terminal type set to 'macintosh'
Options are 'nogx singlewin novertical'

gnuplot> %plot(t,log(t),"notitle");

gnuplot> %replot(t,exp(t),"notitle \
w points pt 5");

gnuplot> set terminal macintosh multiwin
Terminal type set to 'macintosh'
Options are 'nogx multiwin novertical'

> ch="title 'sinus'";
ch = "title 'sinus'"
> gnuplot;
gnuplot> plot sin(x) @ch@
gnuplot> end;
```

12.8 grace

`grace` [Procedure]

`grace;`

<commande grace>

<commande grace>`@`<string> `@`<commande grace>

`%`<commande trip> `\`

<commande trip>

`end;`

TRIP takes the grace commands and sends them to grace (line after line). Grace is executed if it wasn't started. The prompt becomes '`grace>`' when the user could enter the grace commands.

WHen the first character is %, then the end of this line is one or more commands trip. This trip command could follow on several lines : the last character must be `\` to indicates that the command follows on the next line.

Strings declared in trip could be send to grace by surrounding with the character @.

```
Example :
> grace;
grace> grace> with g0
grace> read block "/USER/toto"
grace> block xy "1:2"
grace> read block "/USER/toto"
grace> block xy "1:3"
grace> redraw
grace> title "2courbes"
grace> %t=1,10;
```

```
t        Tableau de reels : nb reels =10
grace> %msg "deux lignes\
fin deligne";
deux lignesfin deligne
grace> end;
> >
```

13 Communications

TRIP have different communication tools :
- Communication with other computer algebra systems.
- Execution of functions located in external dynamic libraries.

TRIP communicates with other computer algebra systems on the same computer. These computer algebra systems must be able to import and export computations using the MathML 2.0 protocol.

A global common session is avaible for each computer algebra system. Several different session for each computer algebra system may be defined.

13.1 Maple

TRIP communicates with Maple[1] on all operating systems which could run maple.

The required version of maple must be equal or greater than 11.

13.1.1 maple_put

`maple_put` [Procedure]

`maple_put(` <object identifier> *id*`);`

Send the object identifier to the global maple session. This starts maple if it isn't running.

```
Example :
> _affdist=1$
> s=1+x;
s(x) =    1 +  x
> maple_put(s);
> maple;
> s;
                                        1 + x

> end;
>
```

`maple_put` [Procedure]

`maple_put(` <Maple client> *session* `,` <object identifier> *id*`);`

Send the object identifier to the maple session specified by *session*, which must be started previously.

```
Example :
> _affdist=1$
> fm=maple;
> z:=1;
                                        z := 1

> end;
> s=1+x;
s(x) =    1 +  x
> maple_put(fm,s);
> maple(fm);
> z+s;
                                        2 + x

> end;
>
```

[1] Maple is a registered trademark of Waterloo Maple Inc.

13.1.2 maple_get

`maple_get` [Procedure]

`maple_get(` <object identifier> *id*`);`

Get the value of an object identifier from the global Maple session.

```
Example :
> _affdist=1$
> maple;
> s:=1+y;
                                   s := 1 + y
> end;
> maple_get(s);
> s;
s(y) =    1 +  y
```

`maple_get` [Procedure]

`maple_get(` <Maple client> *session* `,` <object identifier> *id*`);`

Get the value of an object identifier from the specified Maple session.

```
Example :
> _affdist=1$
> fm=maple;
> s:=1+y;
                                   s := 1 + y
> end;
> maple_get(fm, s);
> s;
s(y) =    1 +  y
```

13.1.3 maple

`maple` [Procedure]

`maple;`

<commande maple>

%<commande maple> \

<commande maple>

%<commande trip> \

<commande trip>

`end;`

TRIP accepts maple commands and send them to the global Maple session (line after line). maple will be started if the global session is not already started. The prompt becomes '`maple>`' when you could enter maple commands.

This maple command could continue on several lines : To continue the command on a new line, the last character of the line must be a \ .

If the first character is a %, then the end of the line will be considered as a TRIP command and not as a maple command. This TRIP command could continue on several lines : To continue the command on a new line, the last character of the line must be a \ .

The function `maple_get` retrieves objects (series, vectors, ...) from maple. The command `maple_put` sends objects to maple.

```
Example :
> maple;
> s:=gcd((x+1)*(x-1),(x-1));
                                  s := x - 1
> %maple_get(s);
> %s;
s(x) =
 -                     1
 +                     1*x
> end;
>
```

`maple` [Function]

`<Maple client> = maple;`

`<commande maple>`

`end;`

This command is similar to the previous one but it starts a new session every time. It returns an object which specifies the session for the commands `maple_put` and `maple_get`.

An existing session could be continued by the command `maple(<Maple client> ); ... end;` .

The session may be closed by the command `delete( <Maple client> );`.

`maple` [Function]

`maple( <Maple client> );`

`<commande maple>`

`end;`

This command continues the existing session.

```
Example :
> fm=maple;
> f:=gcd((x+1)*(x-1),(x-1));
                                  f := x - 1
> g:=f+2;
                                  g := x + 1
> end;
> maple_get(fm,g);
> delete(fm);
```

13.2 Mathematica

TRIP communicates with Mathematica[2] on all operating systems which could run mathematica.

The required version of mathematica must be equal or greater than 9.

13.2.1 mathematica_put

`mathematica_put` [Procedure]

`mathematica_put( <object identifier> *id*);`

Send the object identifier to the global mathematica session. This starts mathematica if it isn't running.

[2] Mathematica is a registered trademark of Wolfram Research Inc.

```
Example :
> _affdist=1$
> s=1+x;
s(x) =    1 +  x
> mathematica_put(s);
> mathematica;
> s
1 + x
> end;
>
```

`mathematica_put` [Procedure]

`mathematica_put(` <Mathematica client> *session* `,` <object identifier> *id*`);`

Send the object identifier to the mathematica session specified by *session*, which must be started previously.

```
Example :
> _affdist=1$
> fm=mathematica;
> z:=1;
> end;
> s=1+x;
s(x) =    1 +  x
> mathematica_put(fm,s);
> mathematica(fm);
> z+s;
> end;
>
```

13.2.2 mathematica_get

`mathematica_get` [Procedure]

`mathematica_get(` <object identifier> *id*`);`

Get the value of an object identifier from the global mathematica session.

```
Example :
> _affdist=1$
> mathematica;
> s:=1+y;
> end;
> mathematica_get(s);
> s;
s(y) =    1 +  y
>
```

`mathematica_get` [Procedure]

`mathematica_get(` <Mathematica client> *session* `,` <object identifier> *id*`);`

Get the value of an object identifier from the specified Mathematica session.

```
Example :
> _affdist=1$
> fm=mathematica;
> s:=1+y;
> end;
> mathematica_get(fm, s);
> s;
s(y) =    1 +  y
>
```

13.2.3 mathematica

`mathematica` [Procedure]

`mathematica;`

<commande mathematica>

%<commande mathematica> \

<commande mathematica>

%<commande trip> \

<commande trip>

`end;`

TRIP accepts mathematica commands and send them to the globale session of Mathematica (line after line). mathematica will be started if the global session not already started. The prompt becomes '`mathematica>`' when you could enter mathematica commands.

This mathematica command could continue on several lines : To continue the command on a new line, the last character of the line must be a \ .

If the first character is a %, then the end of the line will be considered as a TRIP command and not as a mathematica command. This TRIP command could continue on several lines : To continue the command on a new line, the last character of the line must be a \ .

The function `mathematica_get` retrieves objects (series, vectors, ...) from mathematica. The command `mathematica_put` sends objects to mathematica.

```
Example :
> mathematica;
> s=PolynomialGCD[(x+1)*(x-1),(x-1)]
-1 + x
> %mathematica_get(s);
> %s;
s(x) =
 -                        1
 +                        1*x
> end;
>
```

`mathematica` [Function]

<Mathematica client> = `mathematica;`

<commande mathematica>

`end;`

This command is similar to the previous one but it starts a new session every time. It returns an object which specifies the session for the commands `mathematica_put` and `mathematica_get`.

An existing session could be continued by the command `mathematica(`<Mathematica client> `); ... end;` .

The session may be closed by the command `delete(`<Mathematica client> `);`.

`mathematica` [Function]

`mathematica(`<Mathematica client>` );`

<commande mathematica>

`end;`

This command continues the existing session.

```
Example :
> fm=mathematica;
> f=PolynomialGCD[(x+1)*(x-1),(x-1)]
-1 + x
> g=f+2
1 + x
> end;
> mathematica_get(fm,g);
> delete(fm);
>
```

13.3 Communications with other computer algebra systems

TRIP communicates with other computer algebra systems on the same or remote computer. These computer algebra systems must be able to compatible with the "Symbolic Computation Software Composability Protocol" (SCSCP) version 1.3 (`http://www.symcomp.org/`). It supports the OpenMath symbols from the scscp1 content dictionary (`http://www.win.tue.nl/SCIEnce/cds/scscp1.html`) and from the scscp2 content dictionary (`http://www.win.tue.nl/SCIEnce/cds/scscp2.html`). The list of OpenMath content dictionary is given in the appendice (see Appendix A [Supported OpenMath Content Dictionaries], page 211).

Multiple connections could be opened at the same time. TRIP could run in client or server mode.

Before using any functions from this module, you must execute the following command in the TRIP session. This command defines many subroutines which contain the definition of the Openmath symbols.

```
include libscscpserver.t;
```

13.3.1 SCSCP server

In order to start the SCSCP server of TRIP, you should execute the following command in a TRIP session :

```
include libscscpserver.t;
port = 26133;
%scscp_runserver[port];
```

The value of the variable of port may be changed. The default is 26133 for SCSCP servers.

The global variables of TRIP, such as `_modenum`, could be modified before starting the SCSCP server.

The file `libscscpserver.t` exports some symbols from the standard OpenMath content dictionaries. New symbols could be exported to add new functionalities to the SCSCP server. The function `scscp_map_macroassymbolcd`, defined in the file `libscscpserver.t`, must be used.

`scscp_map_macroassymbolcd` [Procedure]

`scscp_map_macroassymbolcd(`<string> *macroname*, <string> *cdname*, <string> *symbolname* `);`

Associate the symbol *symbolname* of the content dictionary *cdname* with the macro *macroname*. When this symbol is found by theSCSCP server or the SCSCP client in a exchanged message, then this macro is executed.

```
Example :
include libscscpserver.t;
_modenum=NUMRATMP;
macro myscscp_evalmul[P1, P2, value]
{
  t=value,value;
  q=evalnum(P1*P2,REAL, (x,t));
  return q[1];
};
scscp_map_macroassymbolcd("myscscp_evalmul",
                           "SCSCP_transient_1","scscp_muleval");
%scscp_runserver[26133];
```

13.3.1.1 scscp_disable_cd

see Section 13.3.2.7 [scscp_disable_cd], page 129.

13.3.1.2 Content dictionary scscp_transient_1

The SCSCP server of TRIP allow the following operations through the content dictionary `scscp_transient_1` :

- serversupportbinary. It returns 1 if the SCSCP server supports the binary encoding for the OpenMath objects.
- serverdisablecd(*CDname*). It specifies that the SCSCP server must not use the content dictionary *CDname* for the OpenMath objects inside the communications with this client.

```
Example :
> with(SCSCP):
> with(Client):
> cookie := StorePersistent("localhost:26133", x*y+1);
       cookie := "TempOID1@localhost:26133"

> Retrieve("localhost:26133", cookie);
Error, (in SCSCP:-Client:-ExtractAnswer) unsupported_CD, <OMS cd = 'polyr' name = 'ter
> CallService("localhost", "scscp_transient_1",
              "serverdisablecd", ["polyr"]):
> CallService("localhost", "scscp_transient_1",
              "serverdisablecd", ["polyu"]):
>  Retrieve("localhost:26133", cookie);
             1.0000000000000000 + 1.0000000000000000 x y
```

13.3.2 SCSCP client

TRIP communicates with other computer algebra systems providing a SCSCP server.

13.3.2.1 scscp_connect

`<scscp client> scscp_connect` [Function]

`scscp_connect(`<string> *computername*, <integer> *port* `);`

Connect to the computer algebra system on the remote computer using the specified port. This function returns an scscp client object which manages that connection.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                   26133
> sc=scscp_connect("localhost", port);
sc  = client SCSCP connecte au serveur SCSCP localhost
> scscp_close(sc);
> stat(sc);
client SCSCP sc : deconnecte
```

13.3.2.2 scscp_close

`scscp_close` [Procedure]

`scscp_close(`<scscp client> *sc* `);`

Close the connection to the computer algebra system specified by *sc*.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                   26133
> sc=scscp_connect("localhost", port);
sc  = client SCSCP connecte au serveur SCSCP localhost
> scscp_close(sc);
> stat(sc);
client SCSCP sc : deconnecte
```

13.3.2.3 scscp_put

`<remote object>  scscp_put` [Function]

`scscp_put(` <scscp client> *sc*, <operation> *x* `);`

`scscp_put(` <scscp client> *sc*, <operation> *x*, <string> *storeoption* `);`

Send the object identifier *x* to the computer algebra system using the scscp client *sc*, previously opened with `scscp_connect`. This function returns a remote object.

If *storeoption* isn't specified, then its value is "persistent".

The string *storeoption* must be one of the following values :

- "persistent" : the remote object will be kept by the server after the connction is closed.
- "session" : the remote object will be destroyed by the server at the end of the connection *sc*.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                   26133
> sc=scscp_connect("localhost", port);
```

```
sc  = client SCSCP connecte au serveur SCSCP localhost
> r=1+x;
r(x) =
                         1
 +                       1*x

> remoter=scscp_put(sc, r);
remoter  = objet distant "TempOID1@localhost:26133"
> scscp_close(sc);
```

13.3.2.4 scscp_get

`scscp_get` [Function]

`scscp_get(` <remote object> *remoteobjectid* `);`

Get the value of the object identifier from the remote computer algebra system using the object *remoteobjectid*, previously returned by `scscp_put`.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                   26133
> sc=scscp_connect("localhost", port);
sc  = client SCSCP connecte au serveur SCSCP localhost
> s=(1+x+y)**2;
s(x,y) =
                         1
 +                       2*y
 +                       1*y**2
 +                       2*x
 +                       2*x*y
 +                       1*x**2

> remoteS = scscp_put(sc, s);
remoteS  = objet distant "TempOID1@localhost:26133"
> q=scscp_get(remoteS);
q(x,y) =
                         1
 +                       2*y
 +                       1*y**2
 +                       2*x
 +                       2*x*y
 +                       1*x**2

> scscp_close(sc);
```

13.3.2.5 scscp_delete

`scscp_delete` [Procedure]

`scscp_delete(` <remote object> *remoteobjectid* `);`

Destroy the value of the object identifier from the remote computer algebra system using the object *remoteobjectid*, previously returned by `scscp_put`.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                    26133
> sc=scscp_connect("localhost", port);
sc  = client SCSCP connecte au serveur SCSCP localhost
> s=(1+x+y)**2;
s(x,y) =
                          1
 +                        2*y
 +                        1*y**2
 +                        2*x
 +                        2*x*y
 +                        1*x**2

> remoteS = scscp_put(sc, s);
remoteS  = objet distant "TempOID1@localhost:26133"
> delete(remoteS);
> scscp_close(sc);
```

13.3.2.6 scscp_execute

`scscp_execute` [Procedure]

`scscp_execute(` <scscp client> *sc*, <string> *returnoption* , <string> *CDname* , <string> *remotecommand* , <operation> , ... `);`

It executes the command *remotecommand* of the OpenMath CD *CDname* on the remote computer algebra system specified by *sc*. The parameters are specified just after the name of the command.

The string *returnoption* must be one of the following values :

- "cookie" : a remote object is returned and the result of the execution will be kept on the SCSCP server.
- "object" : the result of the execution is returned as an OpenMath object and this object is evaluated.
- "nothing" : no result or remote object are returned.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                    26133
> sc=scscp_connect("localhost", port);
sc  = client SCSCP connecte au serveur SCSCP localhost
> q = scscp_execute(sc,"object", "SCSCP_transcient_1", "SCSCP_MUL", 1+7*x,2+3*x);
q(x) =
                          2
 +                       17*x
 +                       21*x**2

> qr = scscp_execute(sc,"cookie", "SCSCP_transcient_1", "SCSCP_MUL", 1+7*x,2+3*x);
```

```
qr  = objet distant "TempOID6@localhost:26133"
> scscp_close(sc);
```

13.3.2.7 scscp_disable_cd

`scscp_disable_cd` [Procedure]

`scscp_disable_cd(` <string> *CDname* `, );`

It disable the content dictionary *CDname*. This dictionary will not be used for the OpenMath output and for the communication with the remote computer algebra system.

```
Example :
> include libscscpserver.t;
Loading the SCSCP client/server library...
Registering the OpenMath CD...
> port=26133;
port =                  26133
> sc=scscp_connect("localhost", port);
sc  = client SCSCP connecte au serveur SCSCP localhost
> scscp_disable_cd("polyr");
> scscp_disable_cd("polyu");
> s=(1+x+y)**2$
> q=scscp_put(sc, s);
q  = objet distant "TempOID1@localhost:26133"
> scscp_close(sc);
```

13.4 Dynamic library

TRIP can load a dynamic library and execute functions of this library. The dynamic libraries have the extensions .so, .dll, .dylib depending on the operating system. The function prototypes is required in order to perform the conversion of the parameters.

13.4.1 extern_function

`extern_function` [Procedure]

`extern_function(` <filename> *filelib*`,` <string> *declfunc*`);`

`extern_function(` <filename> *filelib*`,` <string> *declfunc*`,` <string> *declinout*`);`

It loads the dynamic library *filelib*. It adds the specific extension (.dll, .so, .dylib) of the operating system. It checks the availability of the function in this library. The function could be called as any other standard function of TRIP.

It checks the validity of the prototype of the function specified by *declfunc*. The prototype must be written in C language. The implementation function could be written in another language (fortran, ...).

If the dynamic library depends on other libraries , then it requires to load the other libraries with the command `extern_lib`.

By default, all arguments are input only. To specify that arguments are input-output, it must be specified in the string *declinout*. This string must contain the same number of elements as the number of parameter of the function. Each element are separated with a comma. An element could have the following values :

– in : the parameter is an input only.

– out : the parameter is an output only.

– inout : the parameter is an input and output.

For output arguments, an object identifier must be given to the function to get the value.

Remarks : the function could not have structure or derived type.

```
Example :
> extern_function("libm", "double j0(double);");
> r = j0(0.5);
r =      0.9384698072408129
```

```
Example :
/* Appel de sa propre librairie libtest1 contenant test1.c */
> !"cat test1.c";
#include <math.h>

double mylog1p(double x)
{
 return log1p(x);
}

void mylog1parray(double tabdst[], double tabsrc[], int n)
{
 int i;
 for(i=0; i<n; i++) tabdst[i]=log1p(tabsrc[i]);
}
>
> extern_function(libtest1,"double mylog1p(double x);");
> mylog1p(10);
        2.39789527279837
> log(11);
        2.39789527279837
> extern_function(libtest1,
    "void mylog1parray(double tabdst[], double tabsrc[], int n);",
    "inout,in,in");
> t=1,10;
t        Tableau de reels : nb reels =10
> vnumR res; resize(res,10);
> mylog1parray(res,t,10);
> writes(t,res);
+1.0000000000000000E+00 +6.9314718055994529E-01
+2.0000000000000000E+00 +1.0986122886681098E+00
+3.0000000000000000E+00 +1.3862943611198906E+00
+4.0000000000000000E+00 +1.6094379124341003E+00
+5.0000000000000000E+00 +1.7917594692280550E+00
+6.0000000000000000E+00 +1.9459101490553132E+00
+7.0000000000000000E+00 +2.0794415416798357E+00
+8.0000000000000000E+00 +2.1972245773362196E+00
+9.0000000000000000E+00 +2.3025850929940459E+00
+1.0000000000000000E+01 +2.3978952727983707E+00
```

13.4.2 extern_lib

`extern_lib` [Procedure]

`extern_lib(` <filename> *filelib*`);`

Load the dynamic library *filelib*. It adds the specific extension (.dll, .so, .dylib) of the operating system. This function is used to load librairies required by other librairies.

```
Example :
> extern_lib("libm");
> extern_function("libm", "double j0(double);");
> r = j0(0.5);
r =      0.9384698072408129
```

13.4.3 extern_type

`extern_type` [Procedure]

`extern_lib(` <string> *type*`);`

Declares an external type *type*. The external functions can use or return pointer to objects of this type. *type* could be a C data structure whose fields may be read or modified.

```
Example :
> extern_type("t_calcephbin");
> extern_function("libcalceph",
                "t_calcephbin* calceph_open(const char *filename);");
> eph = calceph_open("inpop06c_m100_p100_littleendian.dat");
```

13.4.4 extern_display

`extern_display` [Procedure]

`extern_display;`

Display the list of external types or functions previously loaded with the command `extern_function` or `extern_type`.

```
Example :
> extern_function("libm", "double j0(double);");
> extern_function("libm", "double j1(double);");
> extern_display;
Liste des fonctions externes :
double j1(double);
double j0(double);
```

14 Macros

14.1 Declaration

`macro` [Procedure]

`macro` <name> [<list of parameters>] { <body> };

`macro` <name> { <body> };

`MACRO` <name> [<list of parameters>] { <body> };

`MACRO` <name> { <body> };

`private macro` <name> [<list of parameters>] { <body> };

`private macro` <name> { <body> };

Define a macro with 0 or more parameter and a body of trip code.

The list of parameters are separated by a comma. The parameter must a name. There is no limitation with the number of parameters. The macros support recursivity but a limit exists (often 70 times).

The last parameter could be `...` to specify that this macro could receive one or more optional argument when it's called. To access to each optional argument, the keyword `macro_optargs[]` is used. To acces to the optional argument of index j, you just write `macro_optargs[j]`. To know the number of provided optional argument, the function `size(macro_optargs)` must be used.

The values corresponding to the `...` could be given as an argument to the execution of a macro using `macro_optargs`.

The execution of the macro could be immediately stopped using the `stop` command (see Section 15.1.5 [stop], page 141).

The macro may define local object identifiers, See Section 2.3.1 [private], page 7.

The visibility of a macro is restricted to the source file which contains it if the declaration is prefixed by the keyword `private`. Only other macros of the same file can call it.

The following example shows the definition of a macro *a* which assigns to *r* the sum of *x* and *y*. *x* and *y* are transmitted by arguments. The macro *b* displays the contents of the current directory.

```
Example :
> macro a [x,y]
{r = x+y;};
> macro b { ! "ls";};
> macro macvar[x,y,...]
{
 s=x+y;
 l = size(macro_optargs);
 msg("number of optional argument %g\n", l);
 for j=1 to l { msg("The optional argument %g :", j); macro_optargs[j]; };
};
> %macvar[P1,P2];
s(P1,P2) =
                        1*P2
 +                      1*P1

l =                      0
```

```
number of optional argument 0
> %macvar[P1,P2,"arg1", 3,5, 7];
s(P1,P2) =
                         1*P2
 +                       1*P1

l =                       4
number of optional argument 4
The optional argument 1 :macro_optargs[1] = "arg1"
The optional argument 2 :macro_optargs[2] =                      3
The optional argument 3 :macro_optargs[3] =                      5
The optional argument 4 :macro_optargs[4] =                      7

> macro macopt[x,y,...]
{
   %macvar[x, macro_optargs];
};
```

14.2 Execution

% [opérateur]

% <name> [<list of parameters>];

% <name> ;

Execute a macro. The list of parameters is a list of parameters separated by a comma.

The parameters can be object identifier, series (result of a computation), strings of characters, arrays or numerical vectors.

The parameter can be transmitted by value or reference.

Parameters given by value can only be a result of a computation, strings of characters, arrays or numerical vectors.

Parameters given by reference can be any object identifier or an element of array. To give a parameter by reference, it must be surrounded with additional brackets []. In this case, the object identifier can be modified or created during the execution of the macro.

The execution of the macro returns a value if the command `return` has been called in the body of the macro. The execution of the macro continues after the command `return`.

The execution of the macro could be immediately stopped using the `stop` command (see Section 15.1.5 [stop], page 141).

Execution of the macro a and b:

```
Example :
> %a[1,5+3*6];
/*ou si S = x + y*/
> %a[S,5];
>%b;
>%c[[t],[u[5]],1+x];
Usage des chaines de caractères:
> macro nomfichier[name,j] {msg name + str(j);};
> %nomfichier["file_",2];
file_2
> ch="file_1.2.";
```

```
ch = "file_1.2."
> %nomfichier[ch,3];
file_1.2.3
>
```

Remarks : The parameters (x,y,...) keep their values. To give an expression, it mustn't be surrounded with brackets [].

```
Example :
 - Si x vaut z et que l'on applique la macro a, une fois la macro
   exécutée, x vaudra z. Alors que si x n'est pas défini, sa valeur
   sera celle qu'il avait à la fin de l'exécution de la macro.
   Example :
            Si on veut ajouter à 'S' la variable 'x':
            > %a[S,[x]];
            et on aura r(x,y) = 1*y + 2*x
- Déclaration d'une matrice. L'identificateur R1 n'existe pas.
   /*creation d'une matrice */
   >macro matrix_create[_nom, _nbligne, _nbcol]
   {
    dim _nom[0:_nbligne, 0: _nbcol];
    for iligne=1 to _nbligne {
      for icol=1 to _nbcol {_nom[iligne,icol]=0$};};
    /*met le nb de ligne en 0,0 et met le nb de col en 0,1 */
    _nom[0, 0]=_nbligne$
    _nom[0, 1]=_nbcol$
   };
   >%matrix_create[[R1],n,3];
  Le tableau sera alors crée et initialise.
  En sortie de la macro, _nom, _nbligne, _nbcol n'existeront plus.
```

return [Procedure]

`return` (<operation>);

`return` <operation> ;

Return the result of an operation when the macro is executed.

The execution of the macro continues after the command `return`. Only the `stop` command stops the execution of the macro.

`return` (<operation> , <operation> , ...);

Return using an one-dimensional array the list of operations as the result of the execution of the macro. The first index of the array is 1.

```
Example :
> _affdist=1$
> macro func1 { s=1+y$ return s; };
> b=%func1;
b(y) =    1 +  y
> macro func2 { v=1,10$ ch="file1"$ return (v, ch); };
> (a, s)=%func2;
> a;
 a  double precision real vector : number of elements =10
> s;
s = "file1"
> t=%func2;
```

```
t [1:2 ] number of elements = 2

> afftab(t);
t[1] =  t  double precision real vector : number of elements =10
t[2] =  "file1"
>
```

14.3 List of macros

`@` [Procedure]

`@;`

Display the list of the defined macros.

```
Example :
> @;
Voici le nom des macros que je connais:
a [x ,y ]
b
```

14.4 Display the body

`affmac` [Procedure]

`affmac` <macro>;

Display the body (trip code) of the macro.

```
Example :
> affmac b;
mon nom est : b
!"ls";
```

14.5 Deletion

14.5.1 effmac

`effmac` [Procedure]

`effmac` <macro>;

Delete a macro.

```
Example :
> macro a[x] { return (x*2);};
> @;
Voici le nom des macros que je connais :
a [ x]
> effmac a;
> @;
je ne connais aucune macro
```

14.5.2 effmacros

`effmacros` [Procedure]

`effmacros;`

Remove all defined macros.

```
Example :
> macro a[x] { return (x*2);};
> macro b {!"ls";};
> @;
Voici le nom des macros que je connais :
a [ x]
b []
> effmacros;
> @;
je ne connais aucune macro
```

14.6 How to redefine a macro ?

We write a new macro using the command `macro`.

```
Example :
> macro a [n] {n;};
> @;
Voici le nom des macros que je connais:
a [n ]
> affmac a;
mon nom est : a
n;
```

14.7 How to save a macro ?

To save the macros on disk, the best solution is to use a text editor such as vi, emacs or nedit. The name of the file which contains trip code should be ended with .t .

15 Loops and conditions

15.1 Loops

15.1.1 while

`while` [Procedure]

`while (<condition> ) do { <body> };`

Execute the body loop while the condition is true.

The `while` loop is more yielding than `for` loop because condition could be sophisticated. A `while` loop could be immediately stopped using the `stop` command. For the condition description, See Section 15.2 [condition], page 141.

The loop may define local object identifiers, See Section 2.3.1 [private], page 7. They will destroyed at the end of each iteration.

```
Example :
> // It displays the numbers from 1 to n:
> p=1$
> while(p<=5) do { p; p=p+1$ };
p =                         1
p =                         2
p =                         3
p =                         4
p =                         5
>
```

15.1.2 for

`for` [Procedure]

`for <nom> = <real> to <real> { <body> };`

`for <nom> = <real> to <real> step <real> { <body> };`

Execute the loop "for ... until ..." (similar to the for loop in pascal, C or fortran languages).

The argument after `step` is the loop step. A `for` loop could be immediately stopped using the `stop` command.

The loop may define local object identifiers, See Section 2.3.1 [private], page 7. They will destroyed at the end of each iteration.

Remarks :

- The loop counter could not be i because i is the mathematical notation for complex numbers.
- The value of the loop counter could not be modified in the body loop.

```
Example :
> for p = 1 to 5 step 2  {p; };
p =                         1
p =                         3
p =                         5
> for p = 5 to -1 step -2 {p; };
p =                         5
p =                         3
p =                         1
```

```
p =                            -1
```

The loop for may be parallelized using the terminology OpenMP. TRIP does not parallelize the loop if global objets are written or if functions which modified the global state of TRIP are called. In these cases, a warning is printed.

If the keyword `distribute` is used, the parallelization is performed on all computing nodes if the application tripcluster runs this code.

`for` [Procedure]

`/*!trip omp parallel for */`
`for` <nom> = <real> `to` <real> `{` <body> `};`

`for` [Procedure]

`/*!trip omp parallel for distribute */`
`for` <nom> = <real> `to` <real> `{` <body> `};`

```
Example :
s=(1+x+y+z+t)**(20)$
dim f[1:8];
n=size(f)$
// parallel execution
time_s;
/*!trip omp parallel for */
for p = 1 to n { f[p]=s/p$ };
time_t;
utilisateur 00.119s  - reel 00.023s  - systeme 00.019s  - (594.73% CPU)
// sequential execution
time_s;
for p = 1 to n { f[p]=s/p$ };
time_t;
utilisateur 00.118s  - reel 00.116s  - systeme 00.008s  - (109.27% CPU)
```

15.1.3 sum

`sum` [Function]

`sum` <nom> = <real> `to` <real> `{` <body> `};`

`sum` <nom> = <real> `to` <real> `step` <real> `{` <body> `};`

Execute the loop "sum ... until ...". It replaces the for loop statement "s=0$ for j=1 to n { s=s+...$}". The value returned by the statement `return` is used for the summation.

The argument after `step` is the loop step. A `sum` loop could be immediately stopped using the `stop` command.

The loop may define local object identifiers, See Section 2.3.1 [private], page 7. They will destroyed at the end of each iteration.

Remarks :

- The loop counter could not be i because i is the mathematical notation for complex numbers.
- The value of the loop counter could not be modified in the body loop.

```
Example :
> s=sum j=1 to 5  { return j; };
s =                        15
> s=sum j=1 to 10 step 2  {
 a=3*j$
 if (mod(j,2)==0) then { return a; } else { return -a; };
};
s =                     -75
```

15.1.4 prod

`sum` [Function]

`prod` <nom> `=` <real> `to` <real> `{` <body> `};`

`prod` <nom> `=` <real> `to` <real> `step` <real> `{` <body> `};`

Execute the loop "prod ... until ...". It replaces the for loop statement "s=1$ for j=1 to n { s=s*...$}". The value returned by the statement `return` is used for the product.

The argument after `step` is the loop step. A `prod` loop could be immediately stopped using the `stop` command.

The loop may define local object identifiers, See Section 2.3.1 [private], page 7. They will destroyed at the end of each iteration.

Remarks :

- The loop counter could not be i because i is the mathematical notation for complex numbers.
- The value of the loop counter could not be modified in the body loop.

```
Example :
> p = prod j = 1 to 10 { return j$ };
p =                    3628800
```

15.1.5 stop

`stop` [Procedure]

`stop;`

Immediately stops the execution of a `for` or `while` loop or of a macro.

Remarks : if stop is used outside a `for` or `while` loop or macro statement, a warning message is displayed but the execution continues.

```
Example :
for p = 1 to 5 { if (p>3) then {stop;} fi; p;};
     1
     2
```

15.2 condition

15.2.1 if

`if` [Procedure]

`if (`<condition>`) then {` <body> `};`

`if (`<condition>`) then {` <body> `} else {` <body> `};`

`if (`<condition>`) then {` <body> `} fi;` (obsolete statement)

Execute the first body (after the then statement) if the condition is true. The second body is executed if the condition is false.

```
Example :
> if (n == 2) then {msg "VRAI";} else {msg"FAUX";};
FAUX
> n=2;
```

```
2
> if (( n >= 0) && (n < 3)) then {msg "VRAI";} else
             {msg "FAUX";};
VRAI
```

Remarks : All object identifiers should be intialized before the execution of the condition. Because otherwise, uninitialized object identifiers will be created as variables and the condition will return false.

15.2.2 Comparison operators

`!=` [Operator]

<operation> `!=` <operation>

This test returns true if the two operations are not equal.

Remarks : the test could be performed on polynomials.

<real vec.> `!=` <real vec.>

This test could be used with the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62). It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

```
Example :
> if (n != 2) then {} else {};
> // Condition between two vectors
> t = 0,10$
> r = 10,0,-1$
> q = ?(t!=r);
 q  double precision real vector : number of elements =11
> q = ?(t!=5);
 q  double precision real vector : number of elements =11
>
```

`==` [Operator]

<operation> == <operation>

This test returns true if the two operations are equal.

If the two operands are the values `NaN` (not a number) , then the test returns false.

Remarks : the test could be performed on polynomials.

<real vec.> == <real vec.>

This test could be used with the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62). It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

```
Example :
> if (n == 2) then {} else {};
> // Condition between two vectors
> t = 0,10$
> r = 10,0,-1$
> q = ?(t==r);
 q  double precision real vector : number of elements =11
> q = ?(t==5);
 q  double precision real vector : number of elements =11
>
```

< [Operator]

<real> < <real>

This test returns true if the first real number is less than the second number.

<real vec.> < <real vec.>

This test could be used with the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62). It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

```
Example :
> n=3$
> if (n < 2) then {} else {};
>
> // Condition between two vectors
> t = 0,10$
> r = 10,0,-1$
> q = ?(t<r);
 q  double precision real vector : number of elements =11
> q = ?(t<5);
 q  double precision real vector : number of elements =11
>
```

> [Operator]

<real> > <real>

This test returns true if the first real number is greater than the second number.

<real vec.> > <real vec.>

This test could be used with the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62). It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

```
Example :
> n=3$
> if (n > 2) then {} else {};
>
> // Condition between two vectors
> t = 0,10$
> r = 10,0,-1$
> q = ?(t>r);
 q  double precision real vector : number of elements =11
> q = ?(t>5);
 q  double precision real vector : number of elements =11
>
```

<= [Operator]

<real> <= <real>

This test returns true if the first real number is less or equal than the second number.

<real vec.> <= <real vec.>

This test could be used with the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62). It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

```
Example :
> n=3$
> if (n <= 2) then {} else {};
>
> // Condition between two vectors
> t = 0,10$
> r = 10,0,-1$
> q = ?(t<=r);
 q  double precision real vector : number of elements =11
> q = ?(t<=5);
 q  double precision real vector : number of elements =11
>
```

`>=` [Operator]

<real> `>=` <real>

This test returns true if the first real number is greater or equal than the second number.

<real vec.> `>=` <real vec.>

This test could be used with the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62). It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

```
Example :
> n=3$
> if (n >= 2) then {} else {};
>
> // Condition between two vectors
> t = 0,10$
> r = 10,0,-1$
> q = ?(t>=r);
 q  double precision real vector : number of elements =11
> q = ?(t>=5);
 q  double precision real vector : number of elements =11
>
```

`&&` [Operator]

<condition> `&&` <condition>

This test returns true if the two conditions are true.

This test could be apply on numerical vectors when the operator `?::` (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62) are used. It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

The test in parenthesis are required if the tests include several `&&` and `||` .

```
Example :
> if ((x==2)&&(y==3)) then {} else {};
>
> // Example using two vectors
> t=0,10;
 t  double precision real vector : number of elements =11
> r=10,0,-1;
 r  double precision real vector : number of elements =11
```

```
> q=?((t>r) && (t!=5));
 q  double precision real vector : number of elements =11
>
```

`||` [Operator]

<condition> `||` <condition>

This test returns true if at least one condition is true.

This test could be apply on numerical vectors when the operator (see Section 10.11 [Conditions], page 93) or the command `select` (see Section 10.7 [Extraction], page 62) are used. It returns a numerical vector which contains only 0 or 1. It compares each element of the two vectors. The numerical vector of real numbers must have the same size.

The test in parenthesis are required if the tests include several `&&` and `||` .

```
Example :
> if ((x==2)||(y==3)) then {} else {};
>
> // Example using two vectors
> t=0,10;
 t  double precision real vector : number of elements =11
> r=10,0,-1;
 r  double precision real vector : number of elements =11
> q=?((t>r) || (t!=5));
 q  double precision real vector : number of elements =11
>
```

15.2.3 switch

`switch` [Procedure]

`switch (` <operation> *expr* `)`

`{`

`case` <operation> `: {` <body> `};`

`case` <operation> `, ..., ` <operation> `: {` <body> `};`

`else {` <body> `}`

`};`

The instruction `switch` control complex conditional on the same value and branching operations.

expr can be a string, a serie or a constant. The value of *expr* are tested successively with each value of the `case` statements with the operator ==. When *expr* is equal to one of the values of a `case` statement, the body statement after this case is executed. The statement `else` is executed if no `case` constant-expression is equal to the value of *expr*. The statement `else {` <body> `}` is optional and has no semi-column after its body.

```
Example :
n=0;
z=0;
j=5;

switch( j )
{
    case -1  : { n=n+1; };
    case 0,1 : { z=z+1; };
```

```
        case "mystring" : { msg "j is a string"; };
        else { msg "j is invalid"; }
    };
```

16 Library

16.1 Lapack

16.1.1 AX=B Solution

Real case for general matrices, (see [lapack_dgesv], page 147)

Real case for band matrices, (see [lapack_dgbsv], page 148)

Real case for tridiagonal matrices, (see [lapack_dgtsv], page 148)

Real case for symmetric matrices, (see [lapack_dsysv], page 149)

Real case for symmetric definite positive matrices (see [lapack_dposv], page 149)

Real case for symmetric definite positive band matrices (see [lapack_dpbsv], page 150)

Real case for symmetric definite positive tridiagonal matrices (see [lapack_dptsv], page 150)

Complex case for general matrices, (see [lapack_zgesv], page 151)

Complex case for band matrices, (see [lapack_zgbsv], page 151)

Complex case for tridiagonal matrices, (see [lapack_zgtsv], page 152)

Complex case for symmetric matrices, (see [lapack_zsysv], page 153)

Complex case for hermitian matrices, (see [lapack_zhesv], page 153)

Complex case for hermitian definite positive matrices (see [lapack_zposv], page 154)

Complex case for hermitian definite positive band matrices (see [lapack_zpbsv], page 154)

Complex case for hermitian definite positive tridiagonal matrices (see [lapack_zptsv], page 155)

lapack_dgesv

`lapack_dgesv` [Function]

`lapack_dgesv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations *A*X=*B*, where *A* is an N-by-N matrix and X and *B* are N-by-NRHS matrices. It uses the LU decomposition.

More information available at `http://www.netlib.org/lapack/double/dgesv.f`

```
Example :
> // resolves a real system of linear equations AX=B
> // where A is a square matrix
> _affc=1$
> A = matrixR[
22.22, -11.11:
-35.07, 78.01]$
> B = matrixR[
-88.88:
382.18]$
> S = lapack_dgesv(A, B)$
> afftab(S);
[ - 2]
[   4]
>
```

lapack_dgbsv

`lapack_dgbsv` [Function]

`lapack_dgbsv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations *AX*=*B*, where *A* is a band matrix of order N, and X and *B* are N-by-NRHS matrices. It uses the LU decomposition.

More information available at `http://www.netlib.org/lapack/double/dgbsv.f`

```
Example :
> // resolves a real system of linear equations AX=B
> // where A is a square matrix
> _affc=1$
> A = matrixR[
-22.22, 0:
15.4, -4.1];
A  double precision real matrix [ 1:2 , 1:2 ]
> B = matrixR[
-88.88:
69.8];
B  double precision real matrix [ 1:2 , 1:1 ]
> S = lapack_dgbsv(A, B);
S  double precision real matrix [ 1:2 , 1:1 ]
> afftab(S);
[   4]
[ - 2]
>
```

lapack_dgtsv

`lapack_dgtsv` [Function]

`lapack_dgtsv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations *AX*=*B*, where *A* is an N-by-N tridiagonal matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/double/dgtsv.f`

```
Example :
> // example of dgtsv routine
> _affc=1$
> A = matrixR[
3.0, 2.1, 0, 0, 0:
3.4, 2.3, -1.0, 0, 0:
0, 3.6, -5.0, 1.9, 0:
0, 0, 7.0, -0.9, 8.0:
0, 0, 0, -6.0, 7.1]$
> B = matrixR[
2.7:
-0.5:
2.6:
0.6:
2.7]$
> S = lapack_dgtsv(A, B);
S  double precision real matrix [ 1:5 , 1:1 ]
> afftab(S);
```

```
[ - 4]
[   7]
[   3]
[ - 4]
[ - 3]
>
```

lapack_dsysv

`lapack_dsysv` [Function]

`lapack_dsysv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations *A*X=*B*, where A is an N-by-N symmetric matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/double/dsysv.f`

```
Example :
> // resolves a real system of linear equations AX=B
> // where A is a symmetric matrix
> _affc=1$
> A = matrixR[
-1.81, 2 :
2, 1.15]$
> B = matrixR[
0.19:
3.15]$
> S=lapack_dsysv(A,B)$
> afftab(S);
[   1]
[   1]
>
```

lapack_dposv

`lapack_dposv` [Function]

`lapack_dposv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations *A*X=*B*, where A is an N-by-N symmetric positive definite matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/double/dposv.f`

```
Example :
> // resolves a real system of linear equations AX=B
> // where A is a symmetric positive difinite matrix
> _affc=1$
> A = [
4.16, -3.12:
-3.12, 5.03]$
> B = matrixR[
10.40:
-13.18]$
> S=lapack_dposv(A,B)$
> afftab(S);
[   1]
[ - 2]
>
```

lapack_dpbsv

`lapack_dpbsv` [Function]

`lapack_dpbsv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations $AX{=}B$, where A is an N-by-N symmetric positive definite band matrix and X and B are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/double/dpbsv.f`

```
Example :
> // example of dpbsv routine
> _affc=1$
> A = matrixR[
5.49, 2.68, 0, 0:
2.68, 5.63, -2.39, 0:
0, -2.39, 2.60, -2.22:
0, 0, -2.22, 5.17]$
> B = matrixR[
22.09:
9.31:
-5.24:
11.83]$
> S = lapack_dpbsv(A, B);
AB  double precision real matrix [ 1:2 , 1:4 ]
S  double precision real matrix [ 1:4 , 1:1 ]
> afftab(S);
[   5]
[ - 2]
[ - 3]
[   1]
>
```

lapack_dptsv

`lapack_dptsv` [Function]

`lapack_dptsv(`<real matrix> *A* `,`<real matrix> *B*`)`

It computes the solution to a real system of linear equations $AX{=}B$, where A is an N-by-N symmetric positive definite tridiagonal matrix and X and B are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/double/dptsv.f`

```
Example :
> // example of dptsv routine
> _affc=1$
> A = matrixR[
4.0, -2.0, 0, 0, 0:
-2.0, 10.0, -6.0, 0, 0:
0, -6.0, 29.0, 15.0, 0:
0, 0, 15.0, 25.0, 8.0:
0, 0, 0, 8.0, 5.0]$
> B = matrixR[
6.0:
9.0:
2.0:
14.0:
```

```
7.0]$
> S = lapack_dptsv(A, B);
S  double precision real matrix [ 1:5 , 1:1 ]
> afftab(S);
[   2.5]
[   2]
[   1]
[ - 1]
[   3]
>
```

lapack_zgesv

lapack_zgesv [Function]

lapack_zgesv(<complex matrix> *A* ,<complex matrix> *B*)

It computes the solution to a complex system of linear equations *AX*=*B*, where *A* is an N-by-N matrix and X and *B* are N-by-NRHS matrices. It uses the LU decomposition.

A is a square matrix. *B* est une matrice.

On entry, *A* is a complex matrix of order N and *B* has N rows. More information available at `http://www.netlib.org/lapack/complex16/zgesv.f`

```
Example :
> // resolves a complex system of linear equations AX=B
> // where A is a square matrix
> _affc=1$
> A = matrixC[
1*i, 0:
0, 2*i]$
> B = matrixC[
7.37*i:
14.74]$
> S=lapack_zgesv(A,B)$
> afftab(S);
[   7.37]
[(0-i*7.37)]
>
```

lapack_zgbsv

lapack_zgbsv [Function]

lapack_zgbsv(<complex matrix> *A* ,<complex matrix> *B*)

It computes the solution to a complex system of linear equations *AX*=*B*, where *A* is a band matrix of order N with KL subdiagonals and KU superdiagonals, and X and *B* are N-by-NRHS matrices. It uses the LU decomposition.

A is a band matrix. *B* est une matrice.

If *A* is diagonal, the call is equivalent to "lapack_zgesv(*A*,*B*);"

More information available `http://www.netlib.org/lapack/complex16/zgbsv.f`

```
Example :
> // resolves a complex system of linear equations AX=B
> // where A is a square matrix
> _affc=1$
> A = matrixC[
-1.65+2.26*i, -2.05-0.85*i, 0.97-2.84*i, 0:
6.30*i, -1.48-1.75*i, -3.99+4.01*i, 0.59-0.48*i:
0, -0.77+2.83*i, -1.06+1.94*i, 3.33-1.04*i:
0, 0, 4.48-1.09*i, -0.46-1.72*i]$
> B = matrixC[
-1.06+21.50*i:
-22.72-53.90*i:
28.24-38.60*i:
-34.56+16.73*i]$
> S = lapack_zgbsv(A, B);
S  double precision complex matrix [ 1:4 , 1:1 ]
> afftab(S);
[(-3+i*2)]
[(1-i*7)]
[(-5+i*4)]
[(6-i*8)]
>
```

lapack_zgtsv

`lapack_zgtsv` [Function]

`lapack_zgtsv(`<complex matrix> *A* `,`<complex matrix> *B*`)`

It computes the solution to a complex system of linear equations *AX*=*B*, where *A* is an N-by-N tridiagonal matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/complex16/zgtsv.f`

```
Example :
> // example of zgtsv routine
> _affc=1$
> A = matrixC[
-1.3+1.3*i, 2-1*i, 0, 0, 0:
1-2*i, -1.3+1.3*i, 2+1*i, 0, 0:
0, 1+i, -1.3+3.3*i, -1+i, 0:
0, 0, 2-3*i, -0.3+4.3*i, 1-i:
0, 0, 0, 1+i, -3.3+1.3*i]$
> B = matrixC[
2.4-5*i:
3.4+18.2*i:
-14.7+9.7*i:
31.9-7.7*i:
-1.0+1.6*i]$
> S = lapack_zgtsv(A, B);
S  double precision complex matrix [ 1:5 , 1:1 ]
> afftab(S);
[(1+i*1)]
[(3-i*1)]
[(4+i*5)]
[(-1-i*2)]
[(1-i*1)]
>
```

lapack_zsysv

`lapack_zsysv` [Function]

`lapack_zsysv(`<complex matrix> *A* `,`<complex matrix> *B*`)`

It computes the solution to a complex system of linear equations *AX=B*, where A is an N-by-N symmetric matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/complex16/zsysv.f`

```
Example :
> // example of zsysv routine
> _affc=1$
> A = matrixC[
-0.56+0.12*i, -1.54-2.86*i, 5.32-1.59*i, 3.80+0.92*i:
-1.54-2.86*i, -2.83-0.03*i, -3.52+0.58*i, -7.86-2.96*i:
5.32-1.59*i, -3.52+0.58*i, 8.86+1.81*i, 5.14-0.64*i:
3.80+0.92*i, -7.86-2.96*i, 5.14-0.64*i, -0.39-0.71*i]$
> B = matrixC[
-6.43+19.24*i:
-0.49-1.47*i:
-48.18+66*i:
-55.64+41.22*i]$
> S = lapack_zsysv(A, B);
S  double precision complex matrix [ 1:4 , 1:1 ]
> afftab(S);
[(-4+i*3)]
[(3-i*2)]
[(-2+i*5)]
[(1-i*1)]
>
```

lapack_zhesv

`lapack_zhesv` [Function]

`lapack_zhesv(`<complex matrix> *A* `,`<complex matrix> *B*`)`

It computes the solution to a complex system of linear equations *AX=B*, where A is an N-by-N hermitian matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/complex16/zhesv.f`

```
Example :
> // example of zhesv routine
> _affc=1$
> A = matrixC[
-1.84, 0.11-0.11*i, -1.78-1.18*i, 3.91-1.50*i:
0.11+0.11*i, -4.63, -1.84+0.03*i, 2.21+0.21*i:
-1.78+1.18*i, -1.84-0.03*i, -8.87, 1.58-0.90*i:
3.91+1.50*i, 2.21-0.21*i, 1.58+0.90*i, -1.36]$
> B = matrixC[
2.98-10.18*i:
-9.58+3.88*i:
-0.77-16.05*i:
7.79+5.48*i]$
> S = lapack_zhesv(A, B);
S  double precision complex matrix [ 1:4 , 1:1 ]
```

```
> afftab(S);
[(2+i*1)]
[(3-i*2)]
[(-1+i*2)]
[(1-i*1)]
>
```

lapack_zposv

`lapack_zposv` [Function]

`lapack_zposv(`<complex matrix> *A* `,`<complex matrix> *B*`)`

It computes the solution to a complex system of linear equations *AX*=*B*, where A is an N-by-N hermitian positive definite matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/complex16/zposv.f`

```
Example :
> // example of zposv routine
> _affc=1$
> A = matrixC[
3.23, 1.51-1.92*i, 1.90+0.84*i, 0.42+2.50*i:
1.51+1.92*i, 3.58, -0.23+1.11*i, -1.18+1.37*i:
1.90-0.84*i, -0.23-1.11*i, 4.09, 2.33-0.14*i:
0.42-2.50*i, -1.18-1.37*i, 2.33+0.14*i, 4.29]$
> B = matrixC[
3.93-6.14*i:
6.17+9.42*i:
-7.17-21.83*i:
1.99-14.38*i]$
> S = lapack_zposv(A, B);
S  double precision complex matrix [ 1:4 , 1:1 ]
> afftab(S);
[(1-i*1)]
[(-4.34355E-15+i*3)]
[(-4-i*5)]
[(2+i*1)]
>
```

lapack_zpbsv

`lapack_zpbsv` [Function]

`lapack_zpbsv(`<complex matrix> *A* `,`<complex matrix> *B*`)`

It computes the solution to a complex system of linear equations *AX*=*B*, where A is an N-by-N hermitian positive definite band matrix and X and *B* are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/complex16/zpbsv.f`

```
Example :
> // example of zpbsv routine
> _affc=1$
> A = matrixC[
16, 16-16*i, 0, 0:
16+16*i, 41, 18+9*i, 0:
0, 18-9*i, 46, 1+4*i:
0, 0, 1-4*i, 21]$
```

```
> B = matrixC[
64+16*i:
93+62*i:
78-80*i:
14-27*i]$
> S = lapack_zpbsv(A, B);
AB  double precision complex matrix [ 1:2 , 1:4 ]
S  double precision complex matrix [ 1:4 , 1:1 ]
> afftab(S);
[(2+i*1)]
[(1+i*1)]
[(1-i*2)]
[(1-i*1)]
>
```

lapack_zptsv

`lapack_zptsv` [Function]

`lapack_zptsv(`<complex matrix> *A* ,<complex matrix> *B*`)`

It computes the solution to a complex system of linear equations $AX{=}B$, where A is an N-by-N hermitian positive definite tridiagonal matrix and X and B are N-by-NRHS matrices.

More information available at `http://www.netlib.org/lapack/complex16/zptsv.f`

```
Example :
> // example of zptsv routine
> _affc=1$
> A = matrixC[
9.39, 1.08-1.73*i, 0, 0:
1.08+1.73*i, 1.69, -0.04+0.29*i, 0:
0, -0.04-0.29*i, 2.65, -0.33+2.24*i:
0, 0, -0.33-2.24*i, 2.17]$
> B = matrixC[
-12.42+68.42*i:
-9.93+0.88*i:
-27.30-0.01*i:
5.31+23.63*i]$
> S = lapack_zptsv(A, B);
S  double precision complex matrix [ 1:4 , 1:1 ]
> afftab(S);
[(-1+i*8)]
[(2-i*3)]
[(-4-i*5)]
[(7+i*6)]
>
```

16.1.2 Least Squares

Real case for general matrices, (see [lapack_dgels], page 156)

Real case for general matrices, using the singular value decomposition (see [lapack_dgelss], page 158)

Real case for the linear equality-constrained least squares (LSE) problems, (see [lapack_dgglse], page 159)

Real case for Gauss-Markov linear model (GLM) problems, (see [lapack_dggglm], page 160)

Complex case for general matrices, (see [lapack_zgels], page 161)

Complex case for general matrices, using the singular value decomposition (see [lapack_zgelss], page 162)

Complex case for the linear equality-constrained least squares (LSE) problems, (see [lapack_zgglse], page 163)

Complex case for Gauss-Markov linear model (GLM) problems, (see [lapack_zggglm], page 163)

lapack_dgels

`lapack_dgels` [Function]

`lapack_dgels(`<real matrix> *A* ,<real matrix> *B* ,<string> *TRANS*`)`

It solves overdetermined or underdetermined real linear systems involving an M-by-N matrix *A*, or its transpose, using a QR or LQ factorization of *A*. It is assumed that *A* has full rank.

Four options are provided:

- case 1: if *TRANS* = 'N' and M >= N: it solves the least squares problem: minimize ||*B* - *A**X||
- case 2: if *TRANS* = 'N' and M < N : it finds the minimum norm solution of an underdetermined system *A**X=*B*
- case 3: if *TRANS* = 'T' and M >= N: it finds the minimum norm solution of an undetermined system *A***T*X=*B*
- case 4: if *TRANS* = 'T' and M < N : it solves the least squares problem: minimize ||*B* - *A***T*X||

Output formats:

- case 1: Rows 1 to n of the output matrix contain the least squares solution vectors. Rows N+1 to M contain the residual squares.
- case 2: Rows 1 to N of the output matrix contain the minimum norm solution vectors.
- case 3: Rows 1 to M of the output matrix contain the minimum norm solution vectors.
- case 4: Rows 1 to M of the output matrix contain the least squares solution vectors. Rows M+1 to N contain the residual squares.

More information available at `http://www.netlib.org/lapack/double/dgels.f`

```
Example :
> // real least squares problem: minimize ||B-A*X||
> _affc=1$
> A = matrixR[
-3, 1:
1, 1:
4-7, 1:
5, 1]$
> B = matrixR[
70:
21:
110:
-35]$
> S = lapack_dgels(A, B, "N")$
> afftab(S);
[ - 15.7727]
[   41.5]
[   28.5243]
[ - 4.13405]
>
> // the residual squares sum can be obtained computing:
> // sqrt(S[3]**2+S[4]**2)
> resid=sqrt(S[3,1]**2+S[4,1]**2);
resid =    28.8223
>
> // minimum norm solution to the underdetermined system AX=B
> A = matrixR[
1, 2, 3, 4:
5, 6, 7, 8:
9, 10, 11, 12]$
> B = matrixR[
30:
70:
110]$
> S = lapack_dgels(A, B, "N")$
> afftab(S);
[   1]
[   2]
[   3]
[   4]
>
> // beware of the row number of the solution
> size(B);
 3
> size(S);
 4
>
> // minimum norm solution of an undetermined system A**T*X=B,
> // where A**T is the transpose of A
> A = matrixR[
1, 5, 9:
2, 6, 10:
3, 7, 11:
4, 8, 12]$
> B = matrixR[
30:
70:
110]$
> S = lapack_dgels(A, B, "T")$
> afftab(S);
[   1]
[   2]
[   3]
[   4]
>
> // same here, beware of the row number of the solution
> size(B);
 3
> size(S);
```

```
 4
>
> // real least squares problem: minimize ||B-A**T*X||
> A = matrixR[
-3, 1, -7, 5:
1, 1, 1, 1]$
> B = matrixR[
70:
21:
110:
-35]$
> S = lapack_dgels(A, B, "T")$
> afftab(S);
[ - 12.1]
[   29.4]
[ - 6.80331]
[ - 4.23261]
>
> // the residual squares sum can be obtained computing:
> // sqrt(S[3]**2+S[4]**2)
> resid=sqrt(S[3,1]**2+S[4,1]**2);
resid =    8.01249
```

lapack_dgelss

`lapack_dgelss` [Function]

`lapack_dgelss(`<real matrix> *A* ,<real matrix> *B* ,<integer> *RCOND*)

It computes the minimum norm solution to a real linear least squares problem: Minimize || |B-A*X| || using the singular value decomposition (SVD) of *A*. *A* is an M-by-Nmatrix which may be rank-deficient.

B has M rows. *RCOND* is the input precision. Assign -1 to RCOND enables machine precision computation.

Rows 1 to N of the output matrix contain the least squares solution vectors. If *A* has rank N and M>=N, then rows N+1 to M contain the residual squares.

More information available at `http://www.netlib.org/lapack/double/dgelss.f`

```
Example :
> // real least squares problem: minimize ||B-A*X||
> _affc=1$
>
> A = [
-0.09, 0.14, -0.46, 0.68, 1.29:
-1.56, 0.20, 0.29, 1.09, 0.51:
-1.48, -0.43, 0.89, -0.71, -0.96:
-1.09, 0.84, 0.77, 2.11, -1.27:
0.08, 0.55, -1.13, 0.14, 1.74:
-1.59, -0.72, 1.06, 1.24, 0.34]$
> B = matrixR[
7.4:
4.2:
-8.3:
1.8:
```

```
8.6:
2.1]$
>
> // we use a 1.E-2 precision
> S = lapack_dgelss(A, B, 1E-2)$
> afftab(S);
[   0.634385]
[   0.969928]
[ - 1.44025]
[   3.36777]
[   3.39917]
[ - 0.00347521]
>
> // using machine precision
> S = lapack_dgelss(A, B, -1)$
> afftab(S);
[ - 0.799745]
[ - 3.28796]
[ - 7.47498]
[   4.93927]
[   0.767833]
[ - 0.00347521]
>
>
```

lapack_dgglse

`lapack_dgglse` [Function]

`lapack_dgglse(`<real matrix> *A* ,<real matrix> *B* ,<real matrix> *C* ,<real matrix> *D*`)`

`lapack_dgglse(`<real matrix> *A* ,<real matrix> *B* ,<real matrix> *C* ,<real matrix> *D* ,<object identifier> *T* ,<object identifier> *R* ,<object identifier> *S*`)`

It solves the linear equality-constrained least squares problem:

minimize `||` *C* - *A**X `||` subject to *B**X = *D*

where *A* is an M-by-N matrix, *B* is a P-by-N matrix, *C* is a given M-vector, and *D* is a given P-vector.

It is assumed that P <= N <= M+P, and

rank(B) = P and rank((A)) = N. ((B))

It uses a generalized QR factorization of matrices (B,A) where B = (0 R)*Q and A = Z*T*Q.

In the complete version of lapack_dgglse, you can get T, R and the residual squares vector S. S is M-N+P long.

More information available at `http://www.netlib.org/lapack/double/dgglse.f`

```
Example :
> // dgglse routine example
> _affc=1$
> A = matrixR[
1, 2:
3, 4:
5, 6]$
> B = matrixR[
2.50, 0.00:
```

```
0.00, 2.50]$
> C = matrixR[
-1:
-2:
-3]$
> D = matrixR[
3.00:
4.00]$
> S = lapack_dgglse(A,B,C,D)$
> afftab(S);
[   1.2]
[   1.6]
```

lapack_dggglm

`lapack_dggglm` [Function]

`lapack_dggglm(`<real matrix> *A* `,`<real matrix> *B* `,`<real matrix> *D* `,`<object identifier> *X* `,`<object identifier> *Y*`)` `lapack_dggglm(`<real matrix> *A* `,`<real matrix> *B* `,`<real matrix> *D* `,`<object identifier> *X* `,`<object identifier> *Y* `,`<object identifier> *T* `,`<object identifier> *R*`)`

It solves a general Gauss-Markov linear model (GLM) problem:

minimize `||` *Y* `||`_2 subject to *D* = *A***X* + *B***Y* *X* where *A* is an N-by-M matrix, *B* is an N-by-P matrix, and *D* is a given N-vector. It is assumed that M <= N <= M+P, rank(*A*) = M and rank(*AB*) = N.

It uses a generalized QR factorization of matrices (*B*,*A*) where *B* = Q*T*Z and *A* = Q*(R). (0)

In particular, if matrix *B* is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

minimize `||` inv(*B*)*(*D*-*A***X*) `||`_2 *X* where inv(*B*) denotes the inverse of *B*.

In the complete version of lapack_dggglm, you can get T and R.

More information available at `http://www.netlib.org/lapack/double/dggglm.f`

```
Example :
> // dggglm routine example
> _affc=1$
> A = matrixR[
7687, 13450.888:
9999, 15499.08:
7594, 12450.12]$
> B = matrixR[
2.50, 11.77:
7, 7:
12.00, 2.50]$
> D = matrixR[
20777:
35713:
21777]$
> lapack_dggglm(A, B, D, X, Y)$
> afftab(X);
[   11.8512]
[ - 5.31511]
> afftab(Y);
```

```
[ - 200.172]
[   141.898]
```

lapack_zgels

`lapack_zgels` [Function]

`lapack_zgels(`<complex matrix> *A* `,`<complex matrix> *B* `,`<string> *TRANS*`)`

It solves overdetermined or underdetermined complex linear systems involving an M-by-N matrix *A*, or its conjuguate-transpose, using a QR or LQ factorization of *A*. It is assumed that *A* has full rank.

Four options are provided:

- case 1: if *TRANS* = 'N' and M >= N: it solves the least squares problem: minimize `||`*B* - *A**X`||`
- case 2: if *TRANS* = 'N' and M < N : it finds the minimum norm solution of an underdetermined system *A**X=*B*
- case 3: if *TRANS* = 'T' and M >= N: it finds the minimum norm solution of an undetermined system *A***H*X=*B*
- case 4: if *TRANS* = 'T' and M < N : it solves the least squares problem: minimize `||`*B* - *A***H*X`||`

Formats de sortie:

- case 1: Rows 1 to n of the output matrix contain the least squares solution vectors. Rows N+1 to M contain the residual squares.
- case 2: Rows 1 to N of the output matrix contain the minimum norm solution vectors.
- case 3: Rows 1 to M of the output matrix contain the minimum norm solution vectors.
- case 4: Rows 1 to M of the output matrix contain the least squares solution vectors. Rows M+1 to N contain the residual squares.

More information available at `http://www.netlib.org/lapack/complex16/zgels.f`

```
Example :
> // resolves a complex least square problem: minimize ||B-A*X||
> _affc=1$
> A = matrixC[
-0.57, -1.28, -0.39, 0.25:
-1.93, 1.08, -0.31, -2.14:
2.30, 0.24, 0.40, -0.35:
-1.93, 0.64, -0.66, 0.08:
0.15, 0.30, 0.15, -2.13:
-0.02, 1.03, -1.43, 0.50]$
> B = matrixC[
-2.67:
-0.55:
3.34:
-0.77:
0.48:
4.10+i]$
> S = lapack_zgels(A, B, "N")$
> afftab(S);
[(1.53387+i*0.158054)]
[(1.87075+i*0.0726528)]
[(-1.52407-i*0.621165)]
```

```
[(0.039183-i*0.0141075)]
[(-0.0085366-i*0.0685483)]
[(0.0204427+i*0.205957)]
>
```

lapack_zgelss

`lapack_zgelss` [Function]

`lapack_zgelss(`<complex matrix> *A* `,`<complex matrix> *B* `,`<real> *RCOND*`)`

It computes the minimum norm solution to a complex linear least squares problem: Minimize `||` |B-A*X| `||` using the singular value decomposition (SVD) of *A*. *A* is an M-by-Nmatrix which may be rank-deficient.

B has M rows. *RCOND* is the input precision. Assign -1 to RCOND enables machine precision computation.

Rows 1 to N of the output matrix contain the least squares solution vectors. If *A* has rank N and M>=N, then rows N+1 to M contain the residual squares.

More information available at `http://www.netlib.org/lapack/complex16/zgelss.f`

```
Example :
> // real least squares problem: minimize ||B-A*X||
> _affc=1$
>
> A = [
-0.09, 0.14, -0.46, 0.68, 1.29:
-1.56, 0.20, 0.29, 1.09, 0.51:
-1.48, -0.43, 0.89, -0.71, -0.96:
-1.09, 0.84, 0.77, 2.11, -1.27:
0.08, 0.55, -1.13, 0.14, 1.74:
-1.59, -0.72, 1.06, 1.24, 0.34]$
> B = matrixC[
7.4:
4.2*i:
-8.3*i:
1.8*i:
8.6:
2.1*i]$
>
> // we use a 1.E-2 precision
> S = lapack_zgelss(A, B, 1E-2)$
> afftab(S);
[(-1.88522+i*2.51961)]
[(2.23426-i*1.26433)]
[(-2.22465+i*0.784398)]
[(-0.257364+i*3.62513)]
[(2.36045+i*1.03872)]
[(3.35419-i*3.35767)]
>
> // using machine precision
> S = lapack_zgelss(A, B, -1)$
> afftab(S);
[(307.51-i*308.309)]
[(920.819-i*924.107)]
```

```
[(1299.69-i*1307.17)]
[(-339.29+i*344.229)]
[(570.037-i*569.27)]
[(3.35419-i*3.35767)]
>
> // wrong result. Beware of the RCOND value
>
```

lapack_zgglse

`lapack_zgglse` [Function]

`lapack_zgglse(`<complex matrix> *A* `,`<complex matrix> *B* `,`<complex matrix> *C* `,`<complex matrix> *D*`)` `lapack_zgglse(`<complex matrix> *A* `,`<complex matrix> *B* `,`<complex matrix> *C* `,`<complex matrix> *D* `,`<object identifier> *T* `,`<object identifier> *R* `,`<object identifier> *S*`)`

It solves the linear equality-constrained least squares problem:

minimize || *C* - *A**X || subject to *B**X = *D*

where *A* is an M-by-N matrix, *B* is a P-by-N matrix, *C* is a given M-vector, and *D* is a given P-vector.

It is assumed that P <= N <= M+P, and

rank(B) = P and rank((A)) = N. ((B))

It uses a generalized QR factorization of matrices (B,A) where B = (0 R)*Q and A = Z*T*Q.

In the complete version of lapack_zgglse, you can get T, R and the residual squares vector S. S is M-N+P long.

More information available at `http://www.netlib.org/lapack/complex16/zgglse.f`

```
Example :
> // zgglse routine example
> _affc=1$
> A = matrixC[
1, 2:
3, 4:
5, 6]$
> B = matrixC[
2.50*i, 0.00:
0.00, 2.50]$
> C = matrixC[
-1:
-2*i:
-3]$
> D = matrixC[
3.00:
4.00*i]$
> S = lapack_zgglse(A,B,C,D)$
> afftab(S);
[(0-i*1.2)]
[(0+i*1.6)]
```

lapack_zggglm

`lapack_zggglm` [Function]

`lapack_zggglm(`<complex matrix> *A* `,`<complex matrix> *B* `,`<complex matrix> *D* `,`<object identifier> *X* `,`<object identifier> *Y*`)` `lapack_zggglm(`<complex matrix> *A* `,`<complex ma-

trix> *B* ,<complex matrix> *D* ,<object identifier> *X* ,<object identifier> *Y* ,<object identifier> *T* ,<object identifier> *R*)

It solves a general Gauss-Markov linear model (GLM) problem:

minimize || *Y* ||_2 subject to *D* = *A***X* + *B***Y* *X* where *A* is an N-by-M matrix, *B* is an N-by-P matrix, and *D* is a given N-vector. It is assumed that M <= N <= M+P, rank(*A*) = M and rank(*AB*) = N.

It uses a generalized QR factorization of matrices (*B*,*A*) where *B* = Q*T*Z and *A* = Q*(R). (0)

In particular, if matrix *B* is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

minimize || inv(*B*)*(*D*-*A***X*) ||_2 *X* where inv(*B*) denotes the inverse of *B*.

In the complete version of lapack_zggglm, you can get T and R.

More information available at `http://www.netlib.org/lapack/complex16/zggglm.f`

```
Example :
> // zggglm routine example
> _affc=1$
> A = matrixC[
7687, 13450.888:
9999, 15499.08:
7594, 12450.12]$
> B = matrixC[
2.50, 11.77*i:
7*i, 7:
12.00, 2.50]$
> D = matrixC[
20777*i:
35713*i:
21777*i]$
> lapack_zggglm(A, B, D, X, Y)$
> afftab(X);
[(-1.49403+i*10.5244)]
[(0.851288-i*4.50903)]
> afftab(Y);
[(45.469-i*168.668)]
[(80.5683+i*6.76462)]
>
```

16.1.3 Factorizations

Real general case for QL factorization, (see [lapack_dgeqrf], page 164)

Real general case for QL factorization, (see [lapack_dgeqlf], page 165)

Real case for Cholesky factorization, (see [lapack_dpotrf], page 166)

Complex general case for QL factorization, (see [lapack_zgeqrf], page 166)

Complex case for Cholesky factorization, (see [lapack_zpotrf], page 167)

lapack_dgeqrf

`lapack_dgeqrf` [Function]

`lapack_dgeqrf(`<real matrix> *A* ,<object identifier> *Q* ,<object identifier> *R*)

It computes a QR factorization of a real M-by-N matrix *A*:

A = Q*R.

More information available at `http://www.netlib.org/lapack/double/dgeqrf.f`

```
Example :
> // computes a QR factorization of a matrix A
> _affc=1$
> A = matrixR[
2, 3, -1, 0, 20:
-6, -5, 0, 2, -33:
2, -5, 6, -6, -43:
4, 6, 2, -3, 49]$
> lapack_dgeqrf(A, Q, R)$
> afftab(Q);
[ - 0.258199  - 0.182574    0.208237  - 0.925547]
[   0.774597    0  - 0.535468  - 0.336563]
[ - 0.258199    0.912871  - 0.267734  - 0.168281]
[ - 0.516398  - 0.365148  - 0.773453    0.0420703]
> afftab(R);
[ - 7.74597  - 6.45497  - 2.32379    4.64758  - 44.9266]
[   0  - 7.30297    4.9295  - 4.38178  - 60.7972]
[   0    0  - 3.36155    2.85583  - 4.55148]
[   0    0    0    0.210352    1.89316]
```

lapack_dgeqlf

`lapack_dgeqlf` [Function]

`lapack_dgeqlf(`<real matrix> *A* `,`<object identifier> *Q* `,`<object identifier> *L*`)`

It computes a QL factorization of a real M-by-N matrix *A*:

A = Q*L.

We assume M >= N.

More information available at `http://www.netlib.org/lapack/double/dgeqlf.f`

```
Example :
> // computes a QL factorization of a matrix A
> _affc=1$
> A = matrixR[
2, 3, -1, 0:
-6, -5, 0, 2:
2, -5, 6, -6:
4, 6, 2, -3:
20, -33, -43, 49]$
> lapack_dgeqlf(A, Q, L)$
> afftab(Q);
[   0.258503  - 0.0987267  - 0.446322    0]
[ - 0.0598575    0.395467    0.782976  - 0.0404061]
[ - 0.292516  - 0.875179    0.329003    0.121218]
[ - 0.914354    0.236816  - 0.28182    0.0606092]
[ - 0.089356  - 0.108807  - 0.00892644  - 0.989949]
> afftab(L);
[ - 5.15342    0    0    0]
[ - 5.54949    7.11392    0    0]
[ - 6.2383  - 8.29521    2.24054    0]
[ - 19.0717    32.6279    43.4164  - 49.4975]
```

lapack_dpotrf

`lapack_dpotrf` [Function]

`lapack_dpotrf(`<real matrix> *A* `)`

It computes the Cholesky decomposition (L) of a real symmetric positive matrix *A*:

A = L*L**T

More information available at `http://www.netlib.org/lapack/double/dpotrf.f`

```
Example :
> _affc=1$
>
> A=matrixR[4.16, -3.12 , 0.56,  -0.10:
 -3.12 ,  5.03 , -0.83 ,  1.18:
  0.56 , -0.83,   0.76 ,  0.34 :
 -0.10 ,  1.18 ,  0.34 ,  1.18 ];
A  double precision real matrix [ 1:4 , 1:4 ]
>
> L=lapack_dpotrf(A);
L  double precision real matrix [ 1:4 , 1:4 ]
> afftab(L);
[   2.03961    0    0    0]
[ - 1.52971    1.64012    0    0]
[   0.274563  - 0.249981    0.788749    0]
[ - 0.049029    0.67373    0.661658    0.534689]
```

lapack_zgeqrf

`lapack_zgeqrf` [Function]

`lapack_zgeqrf(`<complex matrix> *A* `,`<object identifier> *Q* `,`<object identifier> *R*`)`

It computes a QR factorization of a complex M-by-N matrix *A*:

A = Q*R.

More information available at `http://www.netlib.org/lapack/complex16/zgeqrf.f`

```
Example :
> // computes a QR factorization of a matrix A
> _affc=1$
> A = matrixC[
1+i, -5-i, 3+5*i:
0, 8*i, 27:
0, 0, 2+3*i]$
> lapack_zgeqrf(A, Q, R)$
> afftab(Q);
[(-0.707107-i*0.707107)    0    0]
[   0 (0-i*1)    0]
[   0    0 (-0.5547-i*0.83205)]
> afftab(R);
[ - 1.41421 (4.24264-i*2.82843) (-5.65685-i*1.41421)]
[   0  - 8 (0+i*27)]
[   0    0  - 3.60555]
>
```

lapack_zgeqlf

`lapack_zgeqlf` [Function]

`lapack_zgeqlf(`<complex matrix> *A* `,`<object identifier> *Q* `,`<object identifier> *L*)

It computes a QL factorization of a complex M-by-N matrix *A*:

A = Q*L.

We assume M >= N.

More information available at `http://www.netlib.org/lapack/complex16/zgeqlf.f`

```
Example :
> // computes a QL factorization of a matrix A
> _affc=1$
> A = matrixC[
1+i, 0, 0:
-5-i, 8*i, 0:
3+5*i, 27, 2+3*i]$
> lapack_zgeqlf(A, Q, L)$
> afftab(Q);
[(-0.707107-i*0.707107)    0    0]
[   0 (0-i*1)    0]
[   0    0 (-0.5547-i*0.83205)]
> afftab(L);
[ - 1.41421    0    0]
[(1-i*5)  - 8    0]
[(-5.82435-i*0.27735) (-14.9769+i*22.4654)  - 3.60555]
```

lapack_zpotrf

`lapack_zpotrf` [Function]

`lapack_zpotrf(`<complex matrix> *A* `)`

It computes the Cholesky decomposition (L) of a Hermitian, positive-definite matrix *A*:

A = L*L**H

where *L***H is the conjugate-transpose of *L*

More information available at `http://www.netlib.org/lapack/double/zpotrf.f`

```
Example :
>   A=matrixC[ 3.23+I* 0.00 , 1.51-I* 1.92 ,  1.90-I*-0.84 , 0.42-I*-2.50  :
  1.51+I* 1.92  ,  3.58+I* 0.00 ,  -0.23-I*-1.11 ,   -1.18-I*-1.37 :
  1.90+I*-0.84 ,  -0.23+I*-1.11  ,  4.09-I* 0.00 ,   2.33-I* 0.14  :
  0.42+I*-2.50 ,  -1.18+I*-1.37 ,   2.33+I* 0.14 ,   4.29+I* 0.00  ];
A  double precision complex matrix [ 1:4 , 1:4 ]
>
>   L=lapack_zpotrf(A);
L  double precision complex matrix [ 1:4 , 1:4 ]
>   afftab(L);
[         1.797220075561143                          0                           0
0]
[(     0.8401864749527325+i*      1.068316577423342)          1.316353439509685
0                           0]
[(      1.057188279741849-i*      0.467388502622712) (    -0.4701749470106329+i*     0
1.560392977137124                          0]
[(      0.233694251311356-i*      1.391037210186643) (    0.08335250923944196+i*    0.(
(     0.9359617337923402+i*     0.9899692192815739)         0.6603332973655888]
```

16.1.4 Singular Value Decomposition

Real general case for SVD, (see [lapack_dgesvd], page 168)

Real general case for SVD using Divide and Conquer algorithm, (see [lapack_dgesdd], page 168)

Complex general case for SVD, (see [lapack_zgesvd], page 169)

Complex general case for SVD using Divide and Conquer algorithm, (see [lapack_zgesdd], page 169)

lapack_dgesvd

`lapack_dgesvd` [Function]

`lapack_dgesvd(`<real matrix> *A* `,`<object identifier> *U* `,`<object identifier> *SIGMA* `,`<object identifier> *VT*`)`

It computes the singular value decomposition (SVD) of a real M-by-N matrix A : *A* = *U***SIGMA***VT* where *U* is the left singular vectors matrix, *VT* is the transpose matrix of the right singular vectors one, and the diagonal elements of *SIGMA* are the singular values of *A*.

More information available at `http://www.netlib.org/lapack/double/dgesvd.f`

```
Example :
> // DGESVD computes the singular values decomposition of A
> _affc=1$
> A = matrixR[
1, 2:
3, 4]$
> lapack_dgesvd(A, U, SIGMA, VT)$
> afftab(U);
[ - 0.404554  - 0.914514]
[ - 0.914514    0.404554]
> afftab(SIGMA);
[   5.46499    0]
[   0    0.365966]
> afftab(VT);
[ - 0.576048  - 0.817416]
[   0.817416  - 0.576048]
>
```

lapack_dgesdd

`lapack_dgesdd` [Function]

`lapack_dgesdd(`<real matrix> *A* `,`<object identifier> *U* `,`<object identifier> *SIGMA* `,`<object identifier> *VT*`)`

It computes the singular value decomposition (SVD) of a real M-by-N matrix A : *A* = *U***SIGMA***VT* where *U* is the left singular vectors matrix, *VT* is the transpose matrix of the right singular vectors one, and the diagonal elements of *SIGMA* are the singular values of *A*.

It uses a divide and conquer algorithm.

More information available at `http://www.netlib.org/lapack/double/dgesdd.f`

```
Example :
> // DGESDD computes the singular values decomposition of A
> _affc=1$
> A = matrixR[
1, 2:
3, 4]$
> lapack_dgesdd(A, U, SIGMA, VT)$
> afftab(U);
[ - 0.404554  - 0.914514]
[ - 0.914514    0.404554]
> afftab(SIGMA);
[   5.46499    0]
[   0    0.365966]
> afftab(VT);
[ - 0.576048  - 0.817416]
[   0.817416  - 0.576048]
>
```

lapack_zgesvd

`lapack_zgesvd` [Function]

`lapack_zgesvd(`<complex matrix> *A* ,<object identifier> *U* ,<object identifier> *SIGMA* ,<object identifier> *VT*`)`

It computes the singular value decomposition (SVD) of a complex M-by-N matrix A : *A* = *U***SIGMA***VT* where *U* is the left singular vectors matrix, *VT* is the transconjugate matrix of the right singular vectors one, and the diagonal elements of *SIGMA* are the singular values of *A*.

More information available at `http://www.netlib.org/lapack/complex16/zgesvd.f`

```
Example :
> // ZGESVD computes the singular values decomposition of A
> _affc=1$
> A = matrixC[
1+i, -2-i:
3, 2*i]$
> lapack_zgesvd(A, U, SIGMA, VT)$
> afftab(U);
[(-0.158181-i*0.528862) (0.497893-i*0.668869)]
[(-0.824631+i*0.12356) (0.391736+i*0.388921)]
> afftab(SIGMA);
[   4.2049    0]
[   0    1.52278]
> afftab(VT);
[ - 0.751728 (0.259779-i*0.606152)]
[   0.659474 (0.29612-i*0.690947)]
>
```

lapack_zgesdd

`lapack_zgesdd` [Function]

`lapack_zgesdd(`<complex matrix> *A* ,<object identifier> *U* ,<object identifier> *SIGMA* ,<object identifier> *VT*`)`

It computes the singular value decomposition (SVD) of a complex M-by-N matrix A : *A* = *U***SIGMA***VT* where *U* is the left singular vectors matrix, *VT* is the transconjugate matrix of the right singular vectors one, and the diagonal elements of *SIGMA* are the singular values of *A*.

It uses a divide and conquer algorithm.

More information available at `http://www.netlib.org/lapack/complex16/zgesdd.f`

16.1.5 Eigenvalues and Eigenvectors

Real symmetric case, (see [lapack_dsyev], page 170)

Real symmetric case for band matrices, (see [lapack_dsbev], page 170)

Real symmetric case for tridiagonal matrices, (see [lapack_dstev], page 171)

Real non-symmetric case, (see [lapack_dgeev], page 172)

Generalized Eigenproblem, real symmetric case, (see [lapack_dsygv], page 172)

Generalized Eigenproblem, real non-symmetric case, (see [lapack_dggev], page 173)

Generalized Eigenproblem and Schur matrices, (see [lapack_dgges], page 174)

Complex hermitian case, (see [lapack_zheev], page 174)

Complex hermitian case for band matrices, (see [lapack_zhbev], page 175)

Complex non-hermitian case, (see [lapack_zgeev], page 175)

Generalized Eigenproblem, complex hermitian case, (see [lapack_zhegv], page 176)

lapack_dsyev

`lapack_dsyev` [Function]

`lapack_dsyev(`<real matrix> *A*`,`<object identifier> *VALUES* `,`<object identifier> *VECTORS*`)`

It computes all the eigenvalues and eigenvectors of a real symmetricmatrix *A*. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

More information available at `http://www.netlib.org/lapack/double/dsyev.f`

```
Example :
> // dsyev routine test
> _affc=1$
> A = matrixR[
451.27, 0:
0, 512.75]$
> lapack_dsyev(A, values, vectors);
vectors  double precision real matrix [ 1:2 , 1:2 ]
> writes(values);
+4.5126999999999998E+02
+5.1275000000000000E+02
> afftab(vectors);
[   1    0]
[   0    1]
>
```

lapack_dsbev

`lapack_dsbev` [Function]

`lapack_dsbev(`<real matrix> *A*`,`<object identifier> *VALUES* `,`<object identifier> *VECTORS*`)`

It computes all the eigenvalues and eigenvectors of a real symmetric band matrix *A*. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

More information available at `http://www.netlib.org/lapack/double/dsbev.f`

```
Example :
> // example of dsbev routine
> _affc=1$
> A = matrixR[
1, 2, 3, 0, 0:
2, 2, 3, 4, 0:
3, 3, 3, 4, 5:
0, 4, 4, 4, 5:
0, 0, 5, 5, 5]$
> lapack_dsbev(A, Values, Vectors);
> writes(Values);
-3.2473787952520272E+00
-2.6633015451836046E+00
+1.7511163179896112E+00
+4.1598880678262953E+00
+1.4999675954619729E+01
> afftab(Vectors);
[   0.0393846    0.623795    0.563458  - 0.516534    0.15823]
[   0.572127  - 0.257506  - 0.389612  - 0.595543    0.316058]
[ - 0.437179  - 0.590046    0.400815  - 0.147035    0.527682]
[ - 0.44235    0.430844  - 0.558098    0.0469667    0.552286]
[   0.533217    0.103873    0.242057    0.595564    0.540002]
>
```

lapack_dstev

`lapack_dstev` [Function]

`lapack_dstev(`<real matrix> *A*,<object identifier> *VALUES* ,<object identifier> *VECTORS*)

It computes all the eigenvalues and eigenvectors of a real symmetric tridiagonal matrix *A*. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

More information available at `http://www.netlib.org/lapack/double/dstev.f`

```
Example :
> // example of dstev routine
> _affc=1$
> A = matrixR[
1, 1, 0, 0:
1, 4, 2, 0:
0, 2, 9, 3:
0, 0, 3, 16]$
> lapack_dstev(A, Values, Vectors);
> writes(Values);
+6.4756286546948838E-01
+3.5470024748920901E+00
+8.6577669890059994E+00
+1.7147667670632423E+01
> afftab(Vectors);
[   0.939572    0.338755  - 0.04937    0.00337683]
[ - 0.33114    0.86281  - 0.378064    0.0545279]
[   0.0852772  - 0.364803  - 0.855782    0.356769]
[ - 0.0166639    0.0878831    0.349668    0.932594]
>
```

lapack_dgeev

`lapack_dgeev` [Function]

`lapack_dgeev(`<real matrix> *A* `,`<object identifier> *VALUES* `,`<object identifier> *LVECTOR* `,`<object identifier> *RVECTOR*`)`

It computes all the eigenvalues and eigenvectors of a real matrix *A*. It can be non-symmetric. Eigenvalues are put in *VALUES*, left eigenvectors in *LVECTOR* and right eigenvectors in *RVECTOR*.

It solves

*A***RVECTOR*=*VALUES***RVECTOR* and

*LVECTOR***T**A*=*VALUES***LVECTOR***T

where *LVECTOR***T is the conjugate transpose of *LVECTOR*

More information available at `http://www.netlib.org/lapack/double/dgeev.f`

```
Example :
> // dgeev routine example
> _affc=1$
> A = matrixR[
10, 6:
4, 12]$
> lapack_dgeev(A, Values, LVector, RVector)$
> writes(Values);
+6.0000000000000000E+00
+1.6000000000000000E+01
> afftab(LVector);
[ - 0.707107  - 0.5547]
[   0.707107  - 0.83205]
> afftab(RVector);
[ - 0.83205  - 0.707107]
[   0.5547  - 0.707107]
>
```

lapack_dsygv

`lapack_dsygv` [Function]

`lapack_dsygv(`<integer> *ITYPE* `,`<real matrix> *A* `,`<real matrix> *B* `,`<object identifier> *VALUES* `,`<object identifier> *VECTORS*`)`

It computes all the eigenvalues and eigenvectors of a real generalized symmetric-definite eigenproblem. Here *A* and *B* are assumed to be symmetric and *B* is also positive definite. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

3 problems are provided:

*A**X=(lambda)**B**X, if *ITYPE* = 1

*A***B**X=(lambda)*X, if *ITYPE* = 2

*B***A**X=(lambda)*X, if *ITYPE* = 3

More information available at `http://www.netlib.org/lapack/double/dsygv.f`

```
Example :
> // example with B*A*X = (lambda)*X
> _affc=1$
> B = matrixR[
10, 0:
```

```
0, 10]$
> A = matrixR[
451.27, 0:
0, 512.75]$
> lapack_dsygv(3, A, B, Values, Vectors);
Vectors  double precision real matrix [ 1:2 , 1:2 ]
> writes(Values);
+4.5127000000000007E+03
+5.1275000000000009E+03
> afftab(Vectors);
[   3.16228    0]
[   0    3.16228]
```

lapack_dggev

`lapack_dggev` [Function]

`lapack_dggev(`<real matrix> *A* ,<real matrix> *B* ,<object identifier> *VALUES* ,<object identifier> *LVECTOR* ,<object identifier> *RVECTOR*)

It computes all the eigenvalues and eigenvectors of a real generalized eigenproblem. Here *A* and *B* can be non-symmetric. Eigenvalues are put in *VALUES*, left eigenvectors in *LVECTOR* and right eigenvectors in *RVECTOR*.

It solves:

*A***RVECTOR*=*VALUES***B***RVECTOR*

et *LVECTOR***H**A*=*VALUES***LVECTOR***H**B*

where *LVECTOR***H is the conjugate-transpose of *LVECTOR*

More information available at `http://www.netlib.org/lapack/double/dggev.f`

```
Example :
> // dggev routine example
> _affc=1$
> A = matrixR[
10, 6:
4, 12]$
> B = matrixR[
8, 10:
0.3, 12]$
> lapack_dggev(A, B, Values, LVector, RVector)$
> writes(Values);
+9.3655913978494620E-01 +3.9384647034270903E-01
+9.3655913978494620E-01 -3.9384647034270914E-01
> afftab(LVector);
[(0.629444-i*0.320052) (0.629444+i*0.320052)]
[(-0.704921-i*0.295079) (-0.704921+i*0.295079)]
> afftab(RVector);
[(0.7792-i*0.2208) (0.7792+i*0.2208)]
[(-0.283744-i*0.56193) (-0.283744+i*0.56193)]
>
```

lapack_dgges

`lapack_dgges` [Function]

`lapack_dgges(`<real matrix> *A* `,`<real matrix> *B* `,`<object identifier> *VALUES* `,`<object identifier> *S* `,`<object identifier> *T* `,`<object identifier> *VSL* `,`<object identifier> *VSR*`)`

It computes for a pair of N-by-N real nonsymmetric matrices (*A*,*B*), the generalized eigenvalues, the generalized real Schur form and the left and right matrices of Schur vectors. Eigenvalues are put in *VALUES*, the real Schur form of *A* in *S*, the real Schur form of *B* in *T*, the left matrix of Schur vectors in *VSL*, and the right matrix of Schur vectors in *VSR*.

It gives the generalized Schur factorization:

(*A*,*B*) = ((*VSL*)**S**(*VSR*)**T, (*VSL*)**T**(*VSR*)**T)

where *VSR***T is the transpose of *VSR*

More information available at `http://www.netlib.org/lapack/double/dgges.f`

```
Example :
> // dgges routine example
> _affc=1$
> A = matrixR[
10, 6:
4, 12]$
> B = matrixR[
8, 10:
0.3, 12]$
> lapack_dgges(A, B, Values, S, T, VSL, VSR)$
> writes(Values);
+9.3655913978494620E-01 +3.9384647034270903E-01
+9.3655913978494620E-01 -3.9384647034270914E-01
> afftab(S);
[   15.3566    4.77834]
[ - 3.02259    5.31087]
> afftab(T);
[   16.6388    0]
[   0    5.58935]
> afftab(VSL);
[   0.735209    0.677841]
[   0.677841  - 0.735209]
> afftab(VSR);
[   0.365713    0.930728]
[   0.930728  - 0.365713]
>
```

lapack_zheev

`lapack_zheev` [Function]

`lapack_zheev(`<complex matrix> *A*`,`<object identifier> *VALUES* `,`<object identifier> *VECTORS*`)`

It computes all the eigenvalues and eigenvectors of a complex hermitian matrix *A*. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

More information available at `http://www.netlib.org/lapack/complex16/zheev.f`

```
Example :
> // zheev routine test
> _affc=1$
> A = matrixC[
8+8*i, 8:
8, 8+8*i]$
> lapack_zheev(A, Values, Vectors);
Vectors  double precision complex matrix [ 1:2 , 1:2 ]
> writes(Values);
+0.0000000000000000E+00
+1.6000000000000000E+01
> afftab(Vectors);
[ - 0.707107    0.707107]
[   0.707107    0.707107]
```

lapack_zhbev

`lapack_zhbev` [Function]

`lapack_zhbev(`<complex matrix> *A*,<object identifier> *VALUES* ,<object identifier> *VECTORS*)

It computes all the eigenvalues and eigenvectors of a complex hermitian band matrix *A*. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

More information available at `http://www.netlib.org/lapack/complex16/zhbev.f`

```
Example :
> // zhbev routine test
> _affc=1$
> A = matrixC[
8+8*i, 15+i:
15-i, 8+8*i]$
> lapack_zheev(A, Values, Vectors);
Vectors  double precision complex matrix [ 1:2 , 1:2 ]
> writes(Values);
-7.0332963783729099E+00
+2.3033296378372910E+01
> afftab(Vectors);
[(0.705541+i*0.047036) (0.705541+i*0.047036)]
[ - 0.707107    0.707107]
>
```

lapack_zgeev

`lapack_zgeev` [Function]

`lapack_zgeev(`<complex matrix> *A* ,<object identifier> *VALUES* ,<object identifier> *LVECTOR* ,<object identifier> *RVECTOR*)

It computes all the eigenvalues and eigenvectors of a complex matrix *A*. Eigenvalues are put in *VALUES*, left eigenvectors in *LVECTOR* and right eigenvectors in *RVECTOR*.

It solves

*A***RVECTOR*=*VALUES***RVECTOR* and

*LVECTOR***T**A*=*VALUES***LVECTOR***T

where *LVECTOR***T is the transpose of *LVECTOR*

More information available at `http://www.netlib.org/lapack/complex16/zgeev.f`

```
Example :
> //Fr exemple de la routine zgeev
> // zgeev routine test
> _affc=1$
> A = matrixC[
8, 0:
0, 8]$
> lapack_zgeev(A, Values, LV, RV);
> writes(Values);
+8.0000000000000000E+00 +0.0000000000000000E+00
+8.0000000000000000E+00 +0.0000000000000000E+00
> afftab(LV);
[   1    0]
[   0    1]
> afftab(RV);
[   1    0]
[   0    1]
>
```

lapack_zhegv

`lapack_zhegv` [Function]

`lapack_zhegv(`<integer> *ITYPE* `,`<complex matrix> *A* `,`<complex matrix> *B* `,`<object identifier> *VALUES* `,`<object identifier> *VECTORS*`)`

It computes all the eigenvalues and eigenvectors of a complex generalized positive definite eigenproblem. Here *A* and *B* are assumed to be hermitian and *B* is also positive definite. Eigenvalues are put in *VALUES*, eigenvectors in *VECTORS*.

3 problems are provided:

*A**X=(lambda)**B**X, if *ITYPE* = 1

*A***B**X=(lambda)*X, if *ITYPE* = 2

*B***A**X=(lambda)*X, if *ITYPE* = 3

More information available at `http://www.netlib.org/lapack/complex16/zhegv.f`

```
Example :
> // zhegv routine test
> _affc=1$
> B = matrixC[
10,i:
-1*i, 10]$
> A = matrixC[
451.27, 0:
0, 512.75]$
> lapack_zhegv(3, A, B, Values, Vectors);
> afftab(Vectors);
[(0+i*2.69134) (0+i*1.66033)]
[ - 1.38287    2.84388]
> writes(Values);
+4.2492379742004214E+03
+5.3909620257995803E+03
```

17 Numerical computations

17.1 Frequency analysis

17.1.1 naf

`naf` [Procedure]

`naf(`*KTABS*`,` *fichiersource*`,` *fichierresultat*`,` *XH*`,` *T0*`,` *NTERM*`,` *CX*`,` *CY*`);`

with :

- *KTABS* : <integer> : number of data minus 1 to read in the source file.
- *fichiersource* : <string> : file's name (with its extension) containing the data to be processed.
- *fichierresultat* : <string> : file's name (with or without its extension) which will contain the frequencies and amplitudes
- *XH* : <real> : sample step.
- *T0* : <real> : initial time.
- *NTERM* : <integer> : number of frequency to find.
- *CX* : <integer> : index of the column of the real part of the data in the source file (this number must greater or equal to 1).
- *CY* : <integer> : index of the column of the imaginary part of the data in the source file (this number must greater or equal to 1).

It performs the frequency analysis of the data in the source file and stores the found frequencies to the file *fichierresultat*.sol. It uses KTABS+1 values.

KTABS is rounded to the nearest least value such that KTABS = 6n with n positive integer.

The used parameters are stored to the file *fichierresultat*.par. The intermediate results are stored to the file *fichierresultat*.prt if `_naf_iprt`>=0. It uses the global variables `_naf_nulin`, `_naf_iprt`, `_naf_isec`, `_naf_iw`, `_naf_dtour`, `_naf_icplx`.

The file *fichierresultat*.sol contains:

- on the first line : the value of UNIANG.
- on the second line : the number of found frequencies
- on the next lines : the index of the frequency, the frequency, the norm of the amplitude, the real part of the amplitude and the imaginary part of the amplitude.

```
Example :
Le fichier "tessin.out" contient des données
sur 32996 lignes avec une ligne d'entete.
tels que la colonne 2 contient la partie réelle des données.
la colonne 3 contient la partie imaginaire des données.
la première ligne du fichier est à ignorer.
Le temps initial est 0 et le pas est de 0.01.
Nous recherchons 10 frequences.
Le fichier resultat à pour nom "tesnaf".

> _naf_nulin=1;
> naf(32996,tessin.out, tesnaf, 0.01, 0, 10, 2, 3);

Le premier argument de naf n'est pas multiple de 6.
Le premier argument de naf devient 32994.
```

```
Les frequences trouvees sont sauves dans le fichier :tesnaf.sol
Les paramètres employees sont sauves dans le fichier :tesnaf.par

Le fichier tesnaf.par contient :
NOMFPAR = tesnaf.par
NOMFOUT =
NOMFDAT = tessin.out
NOMFSOL = tesnaf.sol
DTOUR   = 6. 83185307179586E+00
T0      = 0.000000000000000E+00
XH      = 1.000000000000000E-02
KTABS   = 32994
DNEPS   = 1.000000000000000E+100
NTERM   = 10
ICPLX   = 1
IW      = 1
ISEC    = 1
NULIN   = 1
ICX     = 2
ICY     = 3
IPRNAF  = -1
```

17.1.2 naftab

`naftab` [Procedure]

`naftab(`*TX*, *TY*, *A*, *F*, *KTABS*, *XH*, *T0*, *NTERM*);

`naftab(`*TX*, *TY*, *A*, *F*, *KTABS*, *XH*, *T0*, *NTERM*, *TRESX*, *TRESY*);

`naftab(`*TX*, *TY*, *A*, *F*, *KTABS*, *XH*, *T0*, *NTERM*, *TRESX*, *TRESY*, `(`*FMIN1*, *FMAX1*, *NTERM1*`)`,...`);`

with :

- *KTABS* : <integer> : number of processed data in TX and TY.
- *TX* : <real vec.> : real part of the processed data.
- *TY* : <real vec.> : imaginary part of the processed data.
- *A* : <complex vec.> : numerical vector of the found amplitudes.
- *F* : <real vec.> : numerical vector of the found frequencies.
- *XH* : <real> : sample step.
- *T0* : <real> : initial time.
- *NTERM* : <integer> : number of frequency to find.
- *TRESX* : <real vec.> : real part of the residues.
- *TRESY* : <real vec.> : imaginary part of the residues.
- *FMIN1* : <real> : lowest frequency of the first window
- *FMAX1* : <real> : upper frequency of the first window
- *NTERM1*: <integer> : number of frequency to find in the first window.

It performs the frequency analysis of the dat in the vectors (TX+i*TY) and store the found frequencies to F and the complex amplitudes to A.

KTABS is rounded to the nearest least value such that KTABS = 6n+1 with n positive integer.

It uses the global variables`_naf_isec`, `_naf_iw`, `_naf_dtour`, `_naf_icplx` .

It finds an approximation of TX+iTY in the form of

$$\sum_{l=0}^{size(F)} A[l]\exp(\imath F[l] \times t)$$

sum from l=0 to size(F) { A[l]*exp(i*F[l]*t) }

In this case, A is a numerical vector of complex numbers (<complex vec.>).

The triplets (*FMINx*, *FMAXx*, *NTERMx*) define the windows. Only the frequencies in the window are searched if a window is specified.

```
Example :
Le fichier "tessin.out" contient des données
sur 32998 lignes avec une ligne d'entete.
tels que la colonne 2 contient la partie réelle des données.
la colonne 3 contient la partie imaginaire des données.
Le temps initial est 0 et le pas est de 0.01.
Nous recherchons 10 frequences.

> vnumR X,Y,F;
vnumC A;
read(tessin2.out, [1:32000],(X,2),(Y,3));
naftab(X,Y,A,F,32000,0.01, 0, 10);
B=abs(A);
Le premier argument de naf n'est pas multiple de 6.
Le premier argument de naf devient 31998.
CORRECTION DE  IFR = 1 AMPLITUDE  = 8.72192e-07
B     Tableau de reels : nb reels =6
> writes(F,B,A);
+9.999993149794888E-02  +1.000000E-01  +1.000E-01  +1.095970178673141E-06
-2.000000944035095E-01  +9.999999E-03  +9.999E-03  +1.312007532388499E-07
+3.000000832689856E-01  +1.000000E-03  +1.000E-03  -1.403292661135361E-08
-4.000000970924361E-01  +9.999995E-05  +9.999E-05  +1.695880029074994E-09
+4.999999805039361E-01  +1.000003E-05  +1.000E-05  +3.216502329016007E-11
-5.999999830866213E-01  +9.999979E-07  +9.999E-07  -1.796078221829677E-12
>
```

17.1.3 freqa

`freqa` [Procedure]

`freqa(`*TFREQ*, *TFREQREF*, *TABCOMBI*, *TINDIC*, *ORDREMAX*, *EPSILON*, *TFREQRESIDU*`)`

with :

- *TFREQ* : <real vec.> : frequencies
- *TFREQREF* : <real vec.> : fundamental frequencies
- *TABCOMBI* : <object identifier> : array with found combinations
- *TINDIC* : <object identifier> : numerical vector of 0 or 1
- *ORDREMAX* : <positive integer> : maximal order of the combinations
- *EPSILON* : <real> : accuracy of the search
- *TFREQRESIDU* : <object identifier> : numerical vector of the residual frequencies

For each frequency in the vector *TFREQ*, it finds the integer combination of the fundamental frequencies stored in *TFREQREF* with the accuracy *EPSILON* and the maximal order *ORDREMAX*. For each frequency in the vector *TFREQ*, the searching will be stop as soon as a valid integer combination is found. The searching is performed by increasing the total order.

If a combination is found, then

- *TINDIC*[...] = 1,
- *TABCOMBI*[][...] contains the combination
- *TFREQRESIDU*[...] contains the residual frequency

If a combination isn't found, then alors

- *TINDIC*[...] =0,
- *TABCOMBI*[][...] =0,
- *TFREQRESIDU*[...] =0

On exit, *TABCOMBI* is an array of `size(`*TFREQREF*`)` numerical vectors of `size(`*TFREQ*`)` integers,

TINDIC is a numerical vector of `size(`*TFREQ*`)` integers (0/1),

*TFREQRESIDU*is a numerical vector of `size(`*TFREQ*`)` reals.

On exit, all elements of the array *TABCOMBI* verify

1<=sum(abs(*TABCOMBI*[i][]))<=*ORDREMAX*.

```
Example :
> freqfond=1,5;
> freqfond[1]=3.1354;
> freqfond[2]=5.452;
> freqfond[3]=7.888;
> freqfond[4]=11.111;
> freqfond[5]=19.777;

> freq=1,5;
> freq=freq*freqfond[1]+freq*freqfond[2]+freq*freqfond[3]+
>  freq*freqfond[4]+freq*freqfond[5];

> freqa(freq,freqfond,tabcomb, tabind,20,1E-6, freqres);

> %writesfreqa[[freq],[freqfond],[tabcomb],[tabind] , [freqres]];

> freqfond[1] = 15677/5000
> freqfond[2] = 1363/250
> freqfond[3] = 986/125
> freqfond[4] = 11111/1000
> freqfond[5] = 19777/1000
  Frequence               Frequence residuelle      combinaison
 --------------------------------------------------------------------
4.736340000000000E+01 +0.000000000000000E+00  1   1   1   1   1
9.472680000000000E+01 +0.000000000000000E+00  2   2   2   2   2
1.420902000000000E+02 +0.000000000000000E+00  3   3   3   3   3
1.894536000000000E+02 +0.000000000000000E+00  4   4   4   4   4
2.368170000000000E+02
```

17.1.4 freqareson

`freqareson` [Procedure]

`freqareson(`*FREQ*, *TFREQREF*, *TABCOMBI*, *ORDREMAX*, *EPSILON*, *TFREQRESIDU*`)`

with :

- *FREQ* : <real> : frequency
- *TFREQREF* : <real vec.> : fundamental frequencies
- *TABCOMBI* : <object identifier> : array with found combinations
- *ORDREMAX* : <positive integer> : maximal order of the combinations
- *EPSILON* : <real> : accuracy of the search
- *TFREQRESIDU* : <object identifier> : numerical vector of the residual frequencies résiduelles

For the frequency *FREQ*, it finds all integer combinations of the fundamental frequencies *TFREQREF* with the accuracyn *EPSILON* and the maximal order *ORDREMAX*.

On exit, for each found combination,

- *TABCOMBI*[][...] contains the combination
- *TFREQRESIDU*[...] contains the residual frequency

On exit, *TABCOMBI* is an array of `size(`*TFREQREF*`)` numerical vectors of integers.

On exit, all elements of the array *TABCOMBI* verify
sum(abs(*TABCOMBI*[i][]))<=*ORDREMAX*.

```
Example :
>vnumR freqfond;
>resize(freqfond,5);
>freqfond[1]=3.1354;
>freqfond[2]=5.452;
>freqfond[3]=2*freqfond[3];
>freqfond[4]=11.111;
>freqfond[5]=19.777;

>freq=2*freqfond[1]+freqfond[2]-freqfond[3]+freqfond[4]-freqfond[5];
>freqareson(freq,freqfond,tabcomb, 8,1E-6, freqres);
>writes("%+-g\t%+-g\t%+-g\t%+-g\t%+-g\n",tabcomb);
+2      -1      +0      +1      -1
+2      +1      -1      +1      -1
+2      -3      +1      +1      -1
```

17.1.5 sertrig

`sertrig` [Function]

`sertrig(`<complex vec.> *TAMP*, <real vec.> *TFREQ*, <variable> *VAR*`)`

`sertrig(`<complex vec.> *TAMP*, <real vec.> *TFREQ*, <variable> *VAR*, <real> *FACT* `)`

with :

- *TFREQ* : vector of frequencies
- *TAMP* : vector of amplitudes
- *VAR* : variable
- *FACT* : real

The vector *TAMP* et *TFREQ* must have the same size.

It computes the sum :

$$\sum_{j=1}^{size(TAMP)} TAMP[j] \times \exp^{(\imath TFREQ[j] \times VAR)}$$

or

$$\sum_{j=1}^{size(TAMP)} TAMP[j] \times \exp^{(\imath 2\pi TFREQ[j] \times VAR/FACT)}$$

The part exp(i*...)is represented as an angular variable where the argument is the frequency and the phase is null. The amplitude is the coefficient of this variable. All angular variables have the prefix _Ex. There will be the same number of angular variables as the frequencies' one. The serie will contain the same number of terms as the size of the numerical vectors.

```
Example :
Le F contient les frequences et le tableau A contient les amplitudes.
> writes(F,A);
+9.999993149794888E-02  +1.000000000456180E-01  +1.095970178673141E-06
-2.000000944035095E-01  +9.999999960679056E-03  +1.312007532388499E-07
+3.000000832689856E-01  +1.000000314882390E-03  -1.403292661135361E-08
-4.000000970924361E-01  +9.999995669530993E-05  +1.695880029074994E-09
+4.999999805039361E-01  +1.000003117384769E-05  +3.216502329016007E-11
-5.999999830866213E-01  +9.999979187109419E-07  -1.796078221829677E-12
> s=sertrig(A,F,t);
s(_EXt1,_EXt2,_EXt3,_EXt4,_EXt5,_EXt6) =
   (9.99997918710942E-07-i*1.79607822182968E-12)*_EXt6
 + (1.00000311738477E-05+i*3.21650232901601E-11)*_EXt5
 + (9.99999566953099E-05+i*1.69588002907499E-09)*_EXt4
 + (0.00100000031488239-i*1.40329266113536E-08)*_EXt3
 + (0.00999999996067906+i*1.3120075323885E-07)*_EXt2
 + (0.100000000045618+i*1.09597017867314E-06)*_EXt1
```

17.2 Fourier Transform

`fft` [Procedure]

`fft (`*TX*, *TY*, *TAMP*, *TFREQ*, *XH*, *T0*, *DTOUR*, *IW*`)`

with :

- *TX* : <real vec.> : real part of the signal
- *TY* : <real vec.> : imaginary part of the signal
- *TAMP* : <complex vec.> : found amplitudes
- *TFREQ* : <real vec.> : found frequencies
- *XH* : <real> : sampling step
- *T0* : <real> : initial time
- *DTOUR* : <real> : "length of the dial turn"
- *IW* : <real> : flag to specify the presence of a window (for the values, (see Chapter 3 [_naf_iw], page 9)).

It computes the Fast Fourier Transform of *TX*+i**TY* and stores the amplitudes and frequencies to the vectors *TAMP* et *TFREQ*.

It keeps only the first "N" elements of *TX* and *TY* such that "N" is the greatest power of 2 less than the size of the vector *TX*.

Remarks: *TX* and *TY* must have the same size.

```
Example :
>vnumC z;
 vnumR freq;
 xin=vnumR[7.:5.:6.:9.];
 yin=vnumR[0:0:0:0];
 fft(xin,yin,z,freq,1,0,pi,0);
 writes(z,freq);
> -2.500000000000000E-01  +0.000000000000000E+00  -1.570796326794897E+00
+2.499999999999993E-01  -1.000000000000000E+00  -7.853981633974483E-01
+6.750000000000000E+00  +0.000000000000000E+00  +0.000000000000000E+00
+2.500000000000007E-01  +1.000000000000000E+00  +7.853981633974483E-01
-2.500000000000000E-01  +0.000000000000000E+00  +1.570796326794897E+00
```

17.3 Inverse Fourier Transform

`ifft` [Procedure]

`ifft (`*TAMP*, *TX*, *TY*, *XH*, *T0*`)`

with :

- *TAMP* : <complex vec.> : amplitudes
- *TX* : <real vec.> : real part of the signal
- *TY* : <real vec.> : imaginary part of the signal
- *XH* : <real> : sampling step
- *T0* : <real> : initial time

It computes the Inverse Fast Fourier Transform of *TAMP* and stores the results to *TX*+i**TY*. *TAMP* is a numerical vector of 2**N+1 complex numbers. On exit, *TX* and *TY* contains 2**N numbers.

```
Example :
>vnumC z;
 vnumR freq;
 xin=vnumR[7.:5.:6.:9.];
 yin=vnumR[0:0:0:0];
 fft(xin,yin,z,freq,1,0,pi,0);
 ifft(z,xout,yout,1,0);
 writes(xout,yout);
> +7.000000000000000E+00    +0.000000000000000E+00
+5.000000000000000E+00  +0.000000000000000E+00
+6.000000000000000E+00  +0.000000000000000E+00
+9.000000000000000E+00  +0.000000000000000E+00
```

17.4 Numerical integration of series or macro

`integnum` [Procedure]

`integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *FICHIER*,
  `{ REAL ,(`*VAR1*, *SER1*, *VI1*`),...}`,
  `{ COMPLEX ,(`*VAR2*, *SER2*, *VI2*`),...});`

`integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *FICHIER*,
  `{ REAL ,(`*VAR1*, *SER1*, *VI1*`),...}`,
  `{ COMPLEX ,(`*VAR2*, *SER2*, *VI2*`),...}`,

RESTARTSTATE);

with :

- *T0* : <real> : initial time
- *TF* : <real> : final time
- *PAS* : <real> : step of the output in the file
- *PREC* : <real> : precision
- *DOPRIMAX* : <real> : maximal step for the integrator
- *PASDOPRI* : <real> : default step for the integrator
- *FICHIER* : <filename> : file for the results
- *VAR* : <variable> : variable
- *SER* : <serie> : serie or operation (right member)
- *VI* : <constant> : initial condition
- *PREVSTATE* : <name> : previous state of the integrator

It performs the numerical integration of the series in the list with the initial conditions and stores the results to the file *FICHIER*.

The used numerical integrator depends on the value of `_integnum` (see Section 3.11 [_integnum], page 13).

The equations are triplets of integrated variable, serie and initial condition.

If *RESTARTSTATE* is specified and doesnot exist, the initial condition are taken from the equations. If *RESTARTSTATE* exists then the initial conditions are taken *RESTARTSTATE* and *T0* is ignored. At the end of the integration, *RESTARTSTATE* is updated. *RESTARTSTATE* is usefull to restart an integration. It is possible to change the integration step, depending on the integrator.

integnum [Function]

`<array of real vec.> = integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*,
`{ REAL ,(`*VAR1*, *SER1*, *VI1*`),...},`
`{ COMPLEX ,(`*VAR2*, *SER2*, *VI2*`),...});`

All parameters are similar to the previous form, except *FICHIER* which is not present.

It performs the numerical integration of the series in the list with the initial conditions and stores the results to an array of numerical vectors. The used numerical integrator depends on the value of `_integnum` (see Section 3.11 [_integnum], page 13).

integnum [Procedure]

`integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *FICHIER*,
`{ REAL ,(`*TVAR1*, *TSER1*, *TVI1*`),...},`
`{ COMPLEX ,(`*TVAR2*, *TSER2*, *TVI2*`),...});`

with :

- *TVAR* : <array of variables> : array of variables
- *TSER* : <array> : array of series (right member)
- *TVI* : <num. vec.> : array or numerical vector of initial conditions

All parameters are similar to the first form, except *TVAR*, *TSER* and *TVI*. The equations are triplets of array of integrated variables, array of series and arrays of initials conditions.

integnum [Procedure]

`<array of real vec.> = integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*,
`{ REAL ,(`*TVAR1*, *TSER1*, *TVI1*`),...},`
`{ COMPLEX ,(`*TVAR2*, *TSER2*, *TVI2*`),...});`

with :

- *TVAR* : <array of variables> : array of variables
- *TSER* : <array> : array of series (right member)
- *TVI* : <num. vec.> : array or numerical vector of initial conditions

All parameters are similar to the second form, except *TVAR*, *TSER* and *TVI.* The equations are triplets of array of integrated variables, array of series and array of initials conditions.

```
Example :
We want to integrate the following system :  { dx/dt = -y , dy/dt= x }
with the initial values x=1 and y=0
between 0 and 2*pi with the step pi/10 and a precision of 1E-12.
The step of the integrator will be pi/20 et the maximal step will be pi/10.
The results will be stored to the file icossin.out
The solution is : x = cos(t) and y = sin(t)

>integnum(0,2*pi, pi/10, 1E-12, pi/10, pi/20,
          icossin.out, REAL, (x, -y, 1),(y,x, 0));

The content of the file icossin.out is :
time                    x                       y
+0.000000000000000E+00 +1.000000000000000E+00 +0.000000000000000E+00
+3.141592653589793E-01 +9.510565162951542E-01 +3.090169943749471E-01
+6.283185307179586E-01 +8.090169943749489E-01 +5.877852522924729E-01
+9.424777960769379E-01 +5.877852522924752E-01 +8.090169943749478E-01
+1.256637061435917E+00 +3.090169943749500E-01 +9.510565162951550E-01
+1.570796326794897E+00 +2.567125951231441E-15 +1.000000000000003E+00
+1.884955592153876E+00 -3.090169943749455E-01 +9.510565162951576E-01
+2.199114857512855E+00 -5.877852522924724E-01 +8.090169943749527E-01
+2.513274122871834E+00 -8.090169943749485E-01 +5.877852522924791E-01
+2.827433388230814E+00 -9.510565162951568E-01 +3.090169943749536E-01
+3.141592653589793E+00 -1.000000000000005E+00 +5.403107187863445E-15
+3.455751918948772E+00 -9.510565162951610E-01 -3.090169943749436E-01
+3.769911184307752E+00 -8.090169943749564E-01 -5.877852522924716E-01
+4.084070449666731E+00 -5.877852522924830E-01 -8.090169943749488E-01
+4.398229715025710E+00 -3.090169943749571E-01 -9.510565162951583E-01
+4.712388980384690E+00 -8.301722685091165E-15 -1.000000000000008E+00
+5.026548245743669E+00 +3.090169943749417E-01 -9.510565162951644E-01
+5.340707511102648E+00 +5.877852522924709E-01 -8.090169943749602E-01
+5.654866776461628E+00 +8.090169943749494E-01 -5.877852522924867E-01
+5.969026041820607E+00 +9.510565162951601E-01 -3.090169943749604E-01
+6.283185307179586E+00 +1.000000000000011E+00 -1.085302505620242E-14
```

```
Example :
> // Instead of storing the result in a file,
> // we store the result in an array q.
> p = pi/10;
p =         0.3141592653589793
```

```
> q = integnum(0, 2*pi, p, 1E-12, p, p/2,REAL,(x,-y,1),(y,x,0));

q [1:3 ] number of elements = 3

> stat(q);
  Array of series
 q [ 1:3  ]
 list of array's elements :
q [ 1 ] =
 Numerical vector q contains 21 double precision reals.
  Size of the array in bytes: 168
q [ 2 ] =
 Numerical vector q contains 21 double precision reals.
  Size of the array in bytes: 168
q [ 3 ] =
 Numerical vector q contains 21 double precision reals.
  Size of the array in bytes: 168
> writes(q);
+0.0000000000000000E+00 +1.0000000000000000E+00 +0.0000000000000000E+00
+3.1415926535897931E-01 +9.5105651629515420E-01 +3.0901699437494706E-01
+6.2831853071795862E-01 +8.0901699437494889E-01 +5.8778525229247280E-01
+9.4247779607693793E-01 +5.8778525229247525E-01 +8.0901699437494767E-01
+1.2566370614359172E+00 +3.0901699437495000E-01 +9.5105651629515486E-01
+1.5707963267948966E+00 +2.6090241078691179E-15 +1.0000000000000027E+00
+1.8849555921538759E+00 -3.0901699437494545E-01 +9.5105651629515764E-01
+2.1991148575128552E+00 -5.8778525229247236E-01 +8.0901699437495267E-01
+2.5132741228718345E+00 -8.0901699437494845E-01 +5.8778525229247902E-01
+2.8274333882308138E+00 -9.5105651629515675E-01 +3.0901699437495334E-01
+3.1415926535897931E+00 -1.0000000000000053E+00 +5.1625370645069779E-15
+3.4557519189487724E+00 -9.5105651629516097E-01 -3.0901699437494379E-01
+3.7699111843077517E+00 -8.0901699437495633E-01 -5.8778525229247180E-01
+4.0840704496667311E+00 -5.8778525229248280E-01 -8.0901699437494901E-01
+4.3982297150257104E+00 -3.0901699437495678E-01 -9.5105651629515842E-01
+4.7123889803846897E+00 -7.9658502016854982E-15 -1.0000000000000080E+00
+5.0265482457436690E+00 +3.0901699437494201E-01 -9.5105651629516430E-01
+5.3407075111026483E+00 +5.8778525229247114E-01 -8.0901699437496011E-01
+5.6548667764616276E+00 +8.0901699437494945E-01 -5.8778525229248657E-01
+5.9690260418206069E+00 +9.5105651629516008E-01 -3.0901699437496022E-01
+6.2831853071795862E+00 +1.0000000000000107E+00 -1.0685896612017132E-14
>
```

integnum [Procedure]

`integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *FICHIER*, *MACRO*, *VI* `);`

with :

- *T0* : <real> : initial time
- *TF* : <real> : final time
- *PAS* : <real> : step of the output in the file
- *PREC* : <real> : precision
- *DOPRIMAX* : <real> : maximal step for the integrator
- *PASDOPRI* : <real> : default step for the integrator
- *FICHIER* : <filename> : file for the results

- *MACRO* : <macro> : name of a macro
- *VI* : <num. vec.> : numerical vector of initial value

It performs the numerical integration using the system defined in the macro with the vector of initial conditions and stores the results to the file *FICHIER*.

The used numerical integrator depends on the value of `_integnum` (see Section 3.11 [_integnum], page 13).

The macro must be refined as `macro nom [N,T,Y,DY] {... }`

This macro will evaluate the derivate of Y at time T (DY=dY/dt=f(Y,T)).

This macro will be called by `integnum` with the following parameter :

- N : dimension of the system
- T : time where the derivative must be evaluated.
- Y : a vector of N numbers which contains the solution at the current time T.
- DY : a vector of N floating-point numbers which must be initialized by integfcn. On exit, it must contain the evaluation of the derivative of Y at the current time T (dy/dt=f(Y,T)).

The macro could access to global objects.

integnum [Function]

`<array of real vec.> = integnum(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *MACRO*, *VI* `);`

All parameters are similar to the previous form, execpt *FICHIER* which is not present.

It performs the numerical integration using the system defined in the macro with the vector of initial conditions and stores the results to an array of numerical vectors. The used numerical integrator depends on the value of `_integnum` (see Section 3.11 [_integnum], page 13).

```
Example :
We want to integrate the following system :  { dx/dt = -y , dy/dt= x }
with the initial values x=1 and y=0
between 0 and 2*pi with the step pi/10 and a precision of 1E-12.
The step of the integrator will be pi/20 et the maximal step will be pi/10.
The results will be stored to the file icossin.out
The solution is : x = cos(t) and y = sin(t)

 macro fcn [N,X,Y,DY]
 {
  DY[1]=-Y[2]$
  DY[2]=Y[1]$
 };

 vnumR v;
 resize(v,2);
 v[1]=1;
 v[2]=0;

 integnum(0,2*pi, pi/10, 1E-12, pi/10, pi/20, output.dat, fcn, v);

The content of the file icossin.out is :
time                    x                       y
+0.000000000000000E+00 +1.000000000000000E+00 +0.000000000000000E+00
```

```
+3.141592653589793E-01 +9.510565162951542E-01 +3.090169943749471E-01
+6.283185307179586E-01 +8.090169943749489E-01 +5.877852522924729E-01
+9.424777960769379E-01 +5.877852522924752E-01 +8.090169943749478E-01
+1.256637061435917E+00 +3.090169943749500E-01 +9.510565162951550E-01
+1.570796326794897E+00 +2.567125951231441E-15 +1.000000000000003E+00
+1.884955592153876E+00 -3.090169943749455E-01 +9.510565162951576E-01
+2.199114857512855E+00 -5.877852522924724E-01 +8.090169943749527E-01
+2.513274122871834E+00 -8.090169943749485E-01 +5.877852522924791E-01
+2.827433388230814E+00 -9.510565162951568E-01 +3.090169943749536E-01
+3.141592653589793E+00 -1.000000000000005E+00 +5.403107187863445E-15
+3.455751918948772E+00 -9.510565162951610E-01 -3.090169943749436E-01
+3.769911184307752E+00 -8.090169943749564E-01 -5.877852522924716E-01
+4.084070449666731E+00 -5.877852522924830E-01 -8.090169943749488E-01
+4.398229715025710E+00 -3.090169943749571E-01 -9.510565162951583E-01
+4.712388980384690E+00 -8.301722685091165E-15 -1.000000000000008E+00
+5.026548245743669E+00 +3.090169943749417E-01 -9.510565162951644E-01
+5.340707511102648E+00 +5.877852522924709E-01 -8.090169943749602E-01
+5.654866776461628E+00 +8.090169943749494E-01 -5.877852522924867E-01
+5.969026041820607E+00 +9.510565162951601E-01 -3.090169943749604E-01
+6.283185307179586E+00 +1.000000000000011E+00 -1.085302505620242E-14

Instead of storing the result in a file,
 we store the result in an array q.

q = integnum(0,2*pi, pi/10, 1E-12, pi/10, pi/20, fcn, v);
```

17.5 Numerical integration of external function

`integnumfcn` [Procedure]

`integnumfcn(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *FICHIER*, *LIBRAIRIE*,
`{ REAL ,(`*VAR1*, *VI1*`),...},`
`{ COMPLEX ,(`*VAR2*, *VI2*`),...});`

with :

- *T0* : <real> : initial time
- *TF* : <real> : final time
- *PAS* : <real> : step of the output in the file
- *PREC* : <real> : precision
- *DOPRIMAX* : <real> : maximal step for the integrator
- *PASDOPRI* : <real> : default step for the integrator
- *FICHIER* : <filename> : file for the results
- *LIBRAIRIE* : <filename> : name of the dynamic library (without the extension .so, .dll ou .dylib) where are defined the functions integfcnbegin, integfcn et integfcnend.
- *VAR* : <(array of) variables> : variable or array of variables
- *VI* : <constant or num. vec.> : initial condition or numerical vector of initial conditions

It performs the numerical integration of the function integfcn located in the external library *LIBRAIRIE* with the initial conditions and stores the results to the file *FICHIER*. The used numerical integrator depends on the value of `_integnum` (see Section 3.11 [_integnum], page 13).

The integrated variables are only used for the names in the output file.

`integnumfcn` [Function]

`<array of real vec.> = integnumfcn(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *LIBRAIRIE*,

`{ REAL ,(`*VAR1*, *VI1*`),...},`

`{ COMPLEX ,(`*VAR2*, *VI2*`),...});`

All parameters are similar to the previous form of integnumfcn, execpt *FICHIER* which is not present.

It performs the numerical integration of the function integfcn located in the external library *LIBRAIRIE* with the initial conditions and store the results to an array of numerical vectors. The used numerical integrator depends on the value of `_integnum` (see Section 3.11 [_integnum], page 13).

The name of the integrated variables are ignored by this function.

The functions of the dynamic library must have the following definition :

- void integfcnbegin(const int * neq, void **data);
  - neq : dimension of the system.
  - *data could be initialized by this function and the pointer will be furnished to integfcn and integfcnend. This is a "user" value.

  This function is called before the integration starts.
- int integfcn(const int * neq, const double *t, const double *y, double *yp, void* data);
  - neq : dimension of the system.
  - *t : time where the derivative must be evaluated.
  - y : a vector of neq floating-point numbers which contains the solution at the current time (*t).
  - dy : a vector of neq floating-point numbers which must be initialized by integfcn. On exit, it must contain the evaluation of the derivative of y at the current time (dy/dt=f(y,t)).
  - data : the user value provided by integfcnbegin.

  This function must initialize dy such that dy/dt=f(y,t) and must return 1.
- void integfcnend(void *data);
  - data : the user value provided by integfcnbegin.

  This function is called after the integration stops.

In order to create a dynamic library from a source file written in the language C, it should usually compiled with the following command :

- under linux with gcc : gcc -fPIC -shared fcn.c -o libfcn.so
- under linux with intel c++ : icc -fPIC -shared fcn.c -o libfcn.so
- under MacOS X with gcc : gcc -shared -dynamiclib fcn.c -o libfcn.dylib
- under MacOS X with intel c++ : icc -fPIC -dynamiclib fcn.c -o libfcn.dylib

```
Example :
The dynamic library libfcn.(so,dylib,dll) contains
the results of the compilation of the following source file fcn.c :
#include <math.h>
void integfcnbegin(const int * neq, void **data)
{
}

int integfcn(const int * neq, const double *x, const double *y,
             double *yp, void* data)
{
 yp[0] = sqrt(1.E0+cos(y[1]));
 yp[1] = y[0]+y[1];
 return 1;
}

void integfcnend(void *data)
{
}

To integrate the system { dx/dt = sqrt(1+cos(y)) , dy/dt = x+y }
with the initial condition x=1 and y=0
and to store the result in the file output.dat :

> integnumfcn(0,2*pi, pi/10, 1E-12, pi/10, pi/20, output.dat,
              libfcn, REAL, (x, 1),(y,0));

To integrate the same system
but it stores the result to the array of numerical vectors q :
> q = integnumfcn(0,2*pi, pi/10, 1E-12, pi/10, pi/20,
                  libfcn, REAL, (x, 1),(y,0));
```

`integnumfcn` [Procedure]

`integnumfcn(` *T0*, *TF*, *PAS*, *PREC*, *DOPRIMAX*, *PASDOPRI*, *FICHIER*, *LIBRAIRIE*,
  `{ REAL ,(`*VAR1*, *VI1*`),...},`
  `{ COMPLEX ,(`*VAR2*, *VI2*`),...} ,`
  `evalnum, REAL ,`*TB*, *TB_0*, *TB_PAS*,*TF*, *TF_0*, *TF_PAS* `)`

with :

- *T0* : <real> : initial time
- *TF* : <real> : final time
- *PAS* : <real> : step of the output in the file
- *PREC* : <real> : precision
- *DOPRIMAX* : <real> : maximal step for the integrator
- *PASDOPRI* : <real> : default step for the integrator
- *FICHIER* : <filename> : file for the results
- *LIBRAIRIE* : <filename> : name of the dynamic library (without the extension .so, .dll ou .dylib) where are defined the functions integfcnbegin, integfcn et integfcnend.
- *VAR* : <(array of) variables> : variable or array of variables

- *VI* : <constant or num. vec.> : initial condition or numerical vector of initial conditions
- *TB* : <array of real vec.> : array of time series for the backward time
- *TB_0* : <real> : initial time of TB
- *TB_PAS* : <real> :time-step of TB
- *TF* : <array of real vec.> : array of time series for the forward time (between *T0* and *TF*)
- *TF_0* : <real> : initial time of TF
- *TF_PAS* : <real> : time-step of TF

It performs the numerical integration of the function integfcn located in the external library *LIBRAIRIE* with the initial conditions and time series. It stores the results to the file *FICHIER*. The numerical integrator specified by `_integnum` must be ADAMS (see Section 3.11 [_integnum], page 13).

This function supplies to the function integfcn the extracted data from the time series *TB* or *TF* for the time where the second member must be evaluated. For example, if the second member must be evaluated at the time t, it extracts from *TB* or *TF* the data corresponding to the time t. For the beginning of the ADAMS method, it interpolates the data of *TB* and supplies these interpolated values to the function integfcn. For the time between *T0* and *TF*, it only extracts the values without any interpolation. For example, if *TB* and *TF* are arrays of 4 vectors of 1000 numbers, the function integfcn will be called with a vector of 4 numbers.

The function requires that *TB_0* and *TF_0* must have the same value as *T0*. Moreover, *PAS* must be a multiple of *TB_PAS* and *TF_PAS*.

The integrated variables are only used for the names in the output file.

The functions of the dynamic library must have the following definition :

- void integfcnbegin(const int * neq, void **data);
  - neq : dimension of the system.
  - *data could be initialized by this function and the pointer will be furnished to integfcnend. This is a "user" value.

  This function is called before the integration starts.
- int integfcn(const int * neq, const double *t, const double *y, double *yp, const double* tabdata);
  - neq : dimension of the system.
  - *t : time where the derivative must be evaluated.
  - y : a vector of neq floating-point numbers which contains the solution at the current time (*t).
  - dy : a vector of neq floating-point numbers which must be initialized by integfcn. On exit, it must contain the evaluation of the derivative of y at the current time (dy/dt=f(y,tabdata(t),t)).
  - tabdata : a vector with the extracted values from TB or TF at the time (*t)

  This function must initialize dy such that dy/dt=f(y,tabdata(t),t) and must return 1.
- void integfcnend(void *data);
  - data : the user value provided by integfcnbegin.

  This function is called after the integration stops.

In order to create a dynamic library from a source file written in the language C, it should usually compiled with the following command :

- under linux with gcc : gcc -fPIC -shared fcn.c -o libfcn.so
- under linux with intel c++ : icc -fPIC -shared fcn.c -o libfcn.so
- under MacOS X with gcc : gcc -shared -dynamiclib fcn.c -o libfcn.dylib
- under MacOS X with intel c++ : icc -fPIC -dynamiclib fcn.c -o libfcn.dylib

```
Example :
The dynamic library libfcn.(so,dylib,dll) contains
the results of the compilation of the following source file fcn.c :
#include <math.h>
void integfcnbegin(const int * neq, void **data)
{
}

int integfcn(const int * neq, const double *x, const double *y,
             double *yp, const double *tabdata)
{
 yp[0] = sqrt(1.E0+cos(y[1])*tabdata[0]);
 yp[1] = y[0]*tabdata[1]+y[1];
 return 1;
}

void integfcnend(void *data)
{
}

To integrate the system
{ dx/dt = sqrt(1+cos(y)*sin(a*t)) , dy/dt = cos(b*t)*x+y }
with a=0.5, b=0.9 and the initial condition x=1 and y=0
and to store the result in the file output.dat :
> t0 = 0;
> tf = 2*pi;
> pas=pi/10;
> vnumR tabb[1:2], tabf[1:2];
> t=-30*pas,t0,pas;
> a=0.5; b=0.9;
> tb[1]=sin(a*t); tb[2]=cos(b*t);
> t=t0, tf+2*pas,pas;
> tf[1]=sin(a*t); tf[2]=cos(b*t);
>
> integnumfcn(t0,tf, pas, 1E-12,pas, pas/2, output.dat,
              libfcn, REAL, (x, 1),(y,0),
              evalnum, REAL, tb, t0, -pas, tf, t0, pas);
```

17.6 Integral

`integral` [Function]

`integral(`<num. vec.> *TY*, <real> *XH*, <integer> *ORDRE*, <object identifier> *N*) `integral`(<num. vec.> *TY*, <real> *XH*, <integer> *ORDRE*, <object identifier> *N*, "kahan")

with :

- *TY* : numerical vector
- *XH* : setp between each value in the vector
- *ORDRE* : integration order from 1 to 6
- *N* : index of the last element used in TY

It computes the integral, using Newton-Cotes formulas, of order *ORDRE*, of the numerical vector *TY* where data are equally spaced of *XH*.

It returns in *N* the index of the last element used to compute the integral. The value of *N* verifies : *N*=int((size(*TY*)-1)/*ORDRE*)**ORDRE*+1

If the option "kahan" is set, the function uses the Kahan method to perform the summation (compensated summation).

Reference : For Newton-Cotes formulas, see the book "Introduction to Numerical Analysis", chapter integration, of Stoer and Burlisch.

Kahan, William (January 1965), "Further remarks on reducing truncation errors", Communications of the ACM 8 (1): 40, `http://dx.doi.org/10.1145%2F363707.363723`

```
Example :
> t=0,10000;
t    Tableau de reels : nb reels =10001
> t=t*pi/10000;
> X=-cos(t)+1;
> for j=1 to 6 { s=integral(sin(t),pi/10000,j,N); delta=s-X[N];};
s = 1.99999998355066
delta = -1.64493392240672E-08
s = 1.99999999999999
delta = -7.54951656745106E-15
s = 1.99999995065198
delta = 5.55111512312578E-15
s = 2
delta = -4.44089209850063E-16
s = 2
delta = 8.88178419700125E-16
s = 1.99999921043176
delta = 5.77315972805081E-15
```

17.7 Numerical iteration

`iternum` [Procedure]

`iternum(` *nbiteration*, *passortie*, *fichierresultat* ,

{ `REAL ,(` *variable1*, *serie1*, *valeurinitiale1*`),...`},

{ `COMPLEX ,(` *variable2*, *serie2*, *valeurinitiale2*`),...`}`);`

with :

- *nbiteration* : <integer> : number of iterations
- *passortie* : <integer> : output step
- *fichierresultat* : <filename> : result file
- *variable1* : <variable> : iterated variable
- *serie1* : <operation> : serie (right member)

– *valeurinitiale1* : <constant> :initial value of the variable

It performs *nbiteration* iterations on the equations specified in lists. it writes results to file every *passortie* iterations.

The equations are triplets of the iterated variable, serie and initial value.

The initial value could be a real or a complex number.

```
Example :
> iternum(10,1,sortie.txt,REAL,(un,un+2,3));
Le fichier "sortie.txt" contient :
time                  un
+0.000000000000000E+00 +3.000000000000000E+00
                     1 +5.000000000000000E+00
                     2 +7.000000000000000E+00
                     3 +9.000000000000000E+00
                     4 +1.100000000000000E+01
                     5 +1.300000000000000E+01
                     6 +1.500000000000000E+01
                     7 +1.700000000000000E+01
                     8 +1.900000000000000E+01
                     9 +2.100000000000000E+01
                    10 +2.300000000000000E+01

> iternum(5,1,sortie.txt,REAL,(vn,un-1,3),(un,vn+un,2));
Le fichier "sortie2.txt" contient :
time                  vn                     un
+0.000000000000000E+00 +3.000000000000000E+00 +2.000000000000000E+00
                     1 +1.000000000000000E+00 +5.000000000000000E+00
                     2 +4.000000000000000E+00 +6.000000000000000E+00
                     3 +5.000000000000000E+00 +1.000000000000000E+01
                     4 +9.000000000000000E+00 +1.500000000000000E+01
                     5 +1.400000000000000E+01 +2.400000000000000E+01
```

17.8 Interpolation

`interpol` [Procedure]

`interpol(LINEAR` , *TX*, *DTX*, *TY*)

`interpol(QUADRATIC` , *TX*, *DTX*, *TY*)

`interpol(SPLINE` , *TX*, *DTX*, *TY*)

`interpol(HERMITE` , *TX*, *DTX*, *TY*, *DEG*)

with :

– TX : <real vec.> : vector containing the time of the known data *DTX*.
– DTX : <real vec.> : vector of known data *DTX*=f(*TX*).
– TY : <real vec.> : vector of time to find the values.
– DEG : <integer> : maximal degree of the interpolating polynomial.

It performs the linear, spline, quadratic or hermitian interpolation. It returns a numerical vector of interpolated values at the time *TY* from the known data (*TX*, *DTX*).

Remarks : it assumes that *TX* et *TY* are sorted in ascending or descending order.

If *TY*[i] isn't in the interval [*TX*[0], *TX*[size(*TX*)]], then *DTY*[i] is extrapolated.

The algorithm of the hermitian interpolation is described in the Stoer and Burlish's book, 1993, "Introduction to Numerical Analysis, Chap 2, pages 52-57".

```
Example :
> x1=0,2*pi,2*pi/59;
> x2=0,3,0.5;
> sl=interpol(LINEAR,x1,cos(x1),x2);
> writes("%.2E   %.4E %.4E\n",x2, cos(x2), sl);
0.00E+00 1.0000E+00 1.0000E+00
5.00E-01 8.7758E-01 8.7652E-01
1.00E+00 5.4030E-01 5.3958E-01
1.50E+00 7.0737E-02 7.0719E-02
2.00E+00 -4.1615E-01 -4.1576E-01
2.50E+00 -8.0114E-01 -8.0001E-01
3.00E+00 -9.8999E-01 -9.8920E-01
```

17.9 Coordinate system conversions

17.9.1 ell_to_xyz

`ell_to_xyz` [Procedure]

`ell_to_xyz(` *ELL*, *MU*, *X*, *XP* `);`

with :

- ELL : <array of real vec.> : array of 6 vectors of real numbers which contain the elliptic coordinates.
  - ELL[1] : semi-major axis
  - ELL[2] : mean longitude
  - ELL[3] : eccentricity * cos(longitude of the pericenter)
  - ELL[4] : eccentricity * sin(longitude of the pericenter)
  - ELL[5] : sin(inclination/2) * cos(longitude of the ascending node)
  - ELL[6] : sin(inclination/2) * sin(longitude of the ascending node)
- MU : <real> : product of gravitation constant and the sum of the masses.
- X : <array of real vec.> : array of 3 vectors of real numbers which contain the positions.
- XP : <array of real vec.> :array of 3 vectors of real numbers which contain the velocities.

It converts the elliptic coordinates *ELL* to cartesian coordinates *X* (positions) and *XP* (velocities).

```
Example :
/* calcul d’une seule valeur */
GM= 0.2959122082855911d-03;
mu = GM*1.05;
n=2*pi/221.6;
varpi = 138*pi/180;


vnumR ell[1:6];
resize(ell,1);
ell[1][1] = (mu/n**2)**(1./3.);
ell[2][1] = varpi;
ell[3][1] = 0.54*cos(varpi);
ell[4][1] = 0.54*sin(varpi);
ell[5][1] = 0.0;
ell[6][1] = 0.0;
```

```
ell_to_xyz(ell,mu,x,xp);
writes(x);
writes(xp);
```

```
Example :
/* calcul de plusieurs valeurs contenues dans un fichier */
mu = 1;

vnumR ell[1:6];
read(file1,ell);

ell_to_xyz(ell,mu,x,xp);

writes(x);
writes(xp);
```

17.9.2 xyz_to_ell

`xyz_to_ell` [Procedure]

`xyz_to_ell(` *X*, *XP*, *MU*, *ELL* `);`

with :

- X : <array of real vec.> : array of 3 vectors of real numbers which contain the positions.
- XP : <array of real vec.> :array of 3 vectors of real numbers which contain the velocities.
- MU : <real> : product of gravitation constant and the sum of the masses.
- ELL : <array of real vec.> : array of 6 vectors of real numbers which contain the elliptic coordinates.
  - ELL[1] : semi-major axis
  - ELL[2] : mean longitude
  - ELL[3] : eccentricity * cos(longitude of the pericenter)
  - ELL[4] : eccentricity * sin(longitude of the pericenter)
  - ELL[5] : sin(inclination/2) * cos(longitude of the ascending node)
  - ELL[6] : sin(inclination/2) * sin(longitude of the ascending node)

It converts the cartesian coordinates *X* (positions) and *XP* (velocities) to the elliptic coordinates *ELL*.

```
Example :
mu = 1;

vnumR x[1:3],xp[1:3];
read(file2,x,xp);

xyz_to_ell(x,xp,mu,ell);

writes(ell);
```

18 Utilities

18.1 help

`help` [Procedure]

`help;`

`?;`

Display help.

On Unix and MacOS X operating systems, the help program uses the software info (provided by Texinfo) to display help. On Windows operating systems, the help file (chm format) is displayed.

`help` [Procedure]

`help <motreserve>;`

`? <motreserve>;`

It displays the help in the reference manual for the keyword of TRIP. On Unix and MacOS X operating systems, this function calls the program info (provided by Texinfo).

```
Example :
>  ? vartrip;
```

18.2 exit

`exit` [Procedure]

`quit;`

`exit;`

Stop the execution of TRIP.

18.3 cls

`cls` [Procedure]

`cls;`

Clear the current screen (similar to the Unix command clear).

18.4 pause

`pause` [Procedure]

`pause;`

Display a message, suspend execution until the user hits a key. If the user hits the key `RET`, then the execution continues. If the user hits the key `q+RET` or `e+RET`, then the execution stops and the user returns to the prompt.

`pause(`<integer> *x*`);`

Suspend execution for *x* seconds.

```
Example :
> for j=1 to 3 { j; pause; time_s; pause(2); time_t; };
1
Appuyez sur la touche 'return' pour continuer
ou la touche 'q' ou 'e' pour revenir au prompt ...
```

```
02.000s
2
Appuyez sur la touche ’return’ pour continuer
ou la touche ’q’ ou ’e’ pour revenir au prompt ...

02.000s
3
Appuyez sur la touche ’return’ pour continuer
ou la touche ’q’ ou ’e’ pour revenir au prompt ...

02.000s
```

18.5 msg

`msg` [Procedure]

`msg` <string> ;

`msg(`<string> *textformat*`);`

Display unformatted messages to the screen.

This message must a be string or a text between double-quotes. The messages could be on several lines.

`msg` [Procedure]

`msg(`<string> *textformat*`,` <real> *x*`,` ... `);`

Display formatted messages to the screen with(out) real constants.

The real constants must be formatted. The format is the same as the command printf in language C (see Section 7.6 [str], page 33, for the valid conversion specifiers). This message must a be string or a text between double-quotes. The messages could be on several lines.

To display double-quotes, two double-quotes must be used.

Under the numerical mode NUMRATMP only, the integers or rational numbers are converted to double-precision floating-point numbers before they are displayed if the format is '%g', '%e' or '%f'. if the format is '%d' or '%i', the integers or rational numbers are written without any conversion.

The function `msg` (see Section 7.7 [msg], page 34) has the same behavior but writes the result into a string of characters.

```
Example :
> file="fichier1.dat"$
> msg"écriture du fichier "+file;
écriture du fichier fichier1.dat
> msg "je vais faire un guillemet :
""cette chaine est encadre par des guillemets"".";

je vais faire un guillemet :
"cette chaine est encadre par des guillemets".

> msg("pi=%g pi=%.8E\n",pi,pi);
pi=3.14159 pi=3.14159265E+00
```

18.6 error

`error` [Procedure]

`error(`<string> *textformat*`);`

Raise an error and display the unformatted text to the screen.

This message must a be string or a text between double-quotes. The message could be on several lines.

```
Example :
> x=2;
x =                        2
> if (x==2) then { error("raise an error : x=2"); };
TRIP[ 3 ] : raise an error : x=2
Commande :'if (x==2) then { error("raise an error : x=2"); };'
                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

18.7 try

`try` [Procedure]

`try {` <body> *bodytry* `} catch {` <body> *bodycatch* `};`

Normally, it executes all statements of *bodytry*. if an error occurs during this execution, then it ignores the remaining statements of *bodytry* and it executes the statements of *bodycatch*.

If the global variable *_info* is equal to `on`, then an information message is displayed when the error occurs in the statements of *bodytry*. If the global variable *_info* is equal to `off`, then no information message is displayed when the error occurs in the statements of *bodytry*.

If an error occurs during the execution of the statements of *bodycatch* then an error is generated. This error could be catched if the current statement try-catch is inside another statement try-catch.

It is allowed to imbricate a statement try-catch inside the statements of *bodytry* or of *bodycatch*.

```
Example :
> // displays a warning message but it continues
> _info on;
_info on
> b = 0;
b =                        0
> q = 2;
q =                           2
> try { s = "invalid"+q; b = 2; } catch { b = 1; };
TRIP[ 1 ] :  Can't add
Command  :' try { s = "invalid"+q; b = 2; } catch { b = 1; };'
                      ^^^^^^^^^^^

 Information :  Continue on error (catch statement)
b =                           1
>
> // does not display a warning message but it continues
> _info off;
_info off
> b = 0;
b =                        0
> q = 2;
```

```
q =                          2
> try { s = "invalid"+q;  b = 2;  } catch { b = 1; };
b =                          1
>
```

18.8 delete

`delete` [Procedure]

`delete(` <object identifier> `);`

Delete any object (serie, variable, array, ...).

Remarks :

- Any dependencies are set to 0.
- Angular variables depending on this object will be converted to polynomial variable.
- Truncation depending on this object are deleted.

```
Example :
> file="fichier1.dat";
file = "fichier1.dat"
> X=expi(x,1,0);
1*X

> delete(file);
> delete(x);
La variable angulaire X est convertie en variable polynomiale.
```

18.9 reset

`reset` [Procedure]

`reset;`

Reinitialize TRIP : TRIP global variables are set to default values and delete all object identifiers in memory.

18.10 include

`include` [Procedure]

`include` <filename> ;

`include` <string> ;

It loads the file "fichier" and run it.

This file must be located in the current directory or in the directory specified by the variable `_path`.

The file extension could have any value. The recommanded file extension is `.t` .

Remarks : If the file is an object identifier of type string, then it loads the file which its name is the contents of the string.

```
Example :
>include ellip;
>include fct.t;
> file="fperplanumH";
file = "fperplanumH"
> include file; /*exécute le fichier "fperplanumH" */
```

18.11 @@

`@@` [Procedure]

`@@;`

Reinitialize TRIP. Then it loads and executes the last file executed by the command `include`.

18.12 vartrip

`vartrip` [Procedure]

`vartrip;`

Displays the status of all TRIP global variables.

```
Example :
> vartrip;

        Etat des variables globales de Trip :

                    _affc      =    6
                    _affdist   =    0
                    _affhomog  =    0
                    _echo      =    0
                    _path      =    /exemples/
                    _history   =    /unixfiles/
                    _hist      =    ON
                    _naf_iprt  =    -1
                    _naf_icplx =    1
                    _naf_iw    =    1
                    _naf_isec  =    1
                    _naf_dtour =    6.283185307179586E+00
                    _naf_nulin =    1
                    _naf_tol   =    1.000000000000000E-10
                    _time      =    OFF
                    _comment   =    OFF
                    _affvar    =    ( )
```

18.13 bilan

`bilan` [Procedure]

`bilan;`

Displays all object identifiers in memory with their type :

SERIE	the object is a serie.
VARIABLE	the object is a variable.
CONST	the object is a constant.
MATRIXR	the object is a numerical matrix of real numbers.
MATRIXC	the object is a numerical matrix of complex numbers.
TAB	the object is an array.
TABVAR	the object is a array of variables.
VNUMR	the object is a numerical vector of real numbers.
VNUMC	the object is a numerical vector of complex numbers.
STRING	the object is a string.
EXTERNAL STRUCTURE	the object is an external structure.

FILE	the object is a file.
REMOTE OBJECT	the object is a remote object stored on a SCSCP server.
SCSCP CLIENT	the object is a connection to a SCSCP server.

`bilan mem` [Procedure]

`bilan mem;`

For each serie or array of series, it displays the memory used and the number of terms.

The total number of terms and total memory used for series are displayed.

```
Example :
> bilan;
A   VAR
ASR SERIE
ASRp    SERIE
RSA SERIE
X   VAR
_AsR    SERIE
_CosE   SERIE
_Expiv  SERIE
_RsA    SERIE
_RsASinv    SERIE
_SinE   SERIE
e   VAR
ep  VAR
g   VAR
ga  VAR
lp  VAR
x   VAR
> bilan mem;
          Nom                 Memoire (octets)               Nb de termes
--------------------        --------------------        --------------------
                 ASR                        1704                          64
                ASRp                        1704                          64
                 RSA                        2136                          81
                _AsR                        1704                          64
               _CosE                        2112                          80
              _Expiv                        1728                          64
                _RsA                        2136                          81
            _RsACosv                        2112                          80
           _RsAExpiv                        1728                          64
            _RsASinv                        2112                          80
               _SinE                        2112                          80
--------------------        --------------------        --------------------
                                           21288                         802
```

18.14 stat

`stat` [Procedure]

`stat;`

It displays all objects in memory. These objects are ordred by their type. For each object, it displays a small description. For the series, it displays their contents.

```
Example :
> S = (1+x+y)**2$
> vnumR R$
> vnumC C$
> dim T[1:2]$
> stat;
 Summary of objects :
  Series:
S(x,y) =
                         1
 +                      2*y
 +                      1*y**2
 +                      2*x
 +                      2*x*y
 +                      1*x**2


  Variables :
Variable x type : 2     ordres :   2 2 2 2
 dependances :
 variables dependant de celle-ci :


Variable y type : 2     ordres :   3 3 3 3
 dependances :
 variables dependant de celle-ci :




Arrays of series :

T [1:2 ] number of elements = 2


Double precision real vectors :
 Numerical vector R contains 0 double precision reals.

Double precision complex vectors :
 Numerical vector C contains 0 double precision complex.

>
```

`stat` [Procedure]

`stat(` <object identifier> `);`

`stat(` <object identifier> , "puismin" `);`

`stat(` <object identifier> , "puismax" `);`

`stat(` <object identifier> , "puismin" , "puismax" `);`

Depending on the object type, it displays information on number of terms and the memory used by the object.

If the option "puismin" is set, the function displays the minimal degree of each variable in the serie.

If the option "puismax" is set, the function displays the maximal degree of each variable in the serie.

```
Example :
> s = (1+x+y)**2$
> stat(s);
 serie s (  x , y )
 number of variables : 2   size of the descriptor : 96 bytes
 number of terms : 6   size : 608 bytes
> stat(s,"puismin","puismax");
 serie s (  x , y )
 number of variables : 2   size of the descriptor : 96 bytes
 number of terms : 6   size : 608 bytes
 puismin : x ^  0 ,  y ^  0
 puismax : x ^  2 ,  y ^  2
```

18.15 save_env

`save_env` [Procedure]

`save_env;`

Save TRIP global variables to a stack.

```
Example :
> _mode;
    _mode = POLP
> save_env;
> _mode=POLH;
    _mode = POLH
> rest_env;
> _mode;
    _mode = POLP
```

18.16 rest_env

`rest_env` [Procedure]

`rest_env;`

Restore the TRIP global variables saved by `save_env` from a stack.

```
Example :
> _path="/users/";
        _path = /users/
> save_env;
> _path="/users/toto/";
        _path = /users/toto/
> /*instructions utilisant le chemin specifique */;
> rest_env;
> _path;
        _path = /users/
```

18.17 random

`random` [Function]

`random(<integer> x)`

Return a pseudo-random number in the range from 0 and x-1.

```
Example :
> random(10);
         1
> random(10);
         7
> random(10);
         0
> random(10);
         9
> random(10);
         8
```

18.18 nameof

`nameof` [Function]

`nameof(`<object identifier> *x*`)`

It returns as a string the name of the object.

If the object is an expression, then the empty string `""` is returned.

```
Example :
> c=3$
> n1= nameof(c);
n1 = "c"
> s="abcde"$
> n2 = nameof(s);
n2 = "s"
>
```

18.19 typeof

`typeof` [Function]

`typeof(`<object identifier> *x*`)`

It returns as a string the type of the object. The returned strings are described in the command `bilan` (see Section 18.13 [bilan], page 201).

If the object doesn't exist, then the empty string `""` is returned.

```
Example :
> typeof(2);
"CONST"
> typeof(1+x);
"SERIE"
> t=1,10;
 t  double precision real vector : number of elements =10
> if (typeof(t)=="VNUMR") then { stat(t); };
 Numerical vector t contains 10 double precision reals.
  Size of the array in bytes: 80
>
```

18.20 file_fullname

`file_fullname` [Function]

`file_fullname(`<string> *name*`)`

It builds an absolute file name. All later usage of the returned string will ignore the variable *_path*.

```
Example :
> _path;
_path = /users/guest/data/
> vnumR t;
> fn=file_fullname("/tmp/mydata.txt");
fn  = nom de fichier '/tmp/mydata.txt'
> read(fn,t);
```

18.21 Execution time

18.21.1 time_s

`time_s` [Procedure]

`time_s;`

Initialize the starting time for computations in functions `time_l` and `time_t`.

18.21.2 time_l

`time_l` [Procedure]

`time_l;`

Displays the consumed user CPU time, the elapsed real time, and the consumed system CPU time, since the last call to `time_l`.

If no previous call to `time_l` was performed, then it displays the user time used since the last call to `time_s`.

```
Example :
> time_s; for j=1 to 3 { (1+x+y+z+t+u+v)**18$ time_l; };
utilisateur 00.313s  - reel 00.183s  - systeme 00.057s  - (202.43% CPU)
utilisateur 00.307s  - reel 00.170s  - systeme 00.055s  - (213.11% CPU)
utilisateur 00.311s  - reel 00.173s  - systeme 00.055s  - (211.93% CPU)
```

18.21.3 time_t

`time_t` [Procedure]

`time_t;`

Displays the consumed user CPU time, the elapsed real time, and the consumed system CPU time, since the last call to `time_s`.

`time_t` [Procedure]

`time_t(`<object identifier> *usertime* , <object identifier> *realtime* `);`

Displays the consumed user CPU time, the elapsed real time, and the consumed system CPU time, since the last call to `time_s`. *usertime* will contain the sum of the user and system CPU time and *realtime* the elapsed real time.

```
Example :
> time_s; for j=1 to 3 { (1+x+y+z+t+u+v)**18$ time_t; };
  time_t(usertime, realtime);
utilisateur 00.310s  - reel 00.178s  - systeme 00.055s  - (205.29% CPU)
utilisateur 00.615s  - reel 00.347s  - systeme 00.111s  - (209.44% CPU)
utilisateur 00.928s  - reel 00.519s  - systeme 00.168s  - (211.23% CPU)
utilisateur 00.928s  - reel 00.519s  - systeme 00.168s  - (211.20% CPU)
```

```
> usertime;
usertime =                1.097452
> realtime;
realtime =      0.5196029999999999
```

18.22 Call the shell

! [Procedure]

! <string> ;

Execute a shell command specified in a string.

```
Example :
> dir ="ls -al";
dir = "ls -al"
> ! dir;
total 8136
drwxr-sr-x  20 xxxx   ttt         1536 Sep 27 17:16 .
drwxr-sr-x  22 xxxx   ttt          512 Sep 06 12:16 ..
-rw-r-----   1 xxxx   ttt          281 Sep 08 1998  .Guidefaults
-rw-------   1 xxxx   ttt          204 Sep 27 10:46 .Xauthority
```

! [Procedure]

str(! <string>);

Execute a shell command specified in a string and return the standard output as a string. If the error output is not empty, an error is raised.

```
Example :
> s=str(!"uname");
s = "Darwin
"
```

19 References

- GMP - GNU Multiple Precision Arithmetic Library
  version 6.0.0, 2014
  Torbjorn Granlund et al.
  `http://www.gmplib.org/`
- LTDL - GNU Libtool - The GNU Portable Library Tool
  version 2.4.6, 2014
  `http://www.gnu.org/software/libtool/`
- MPFR: A multiple-precision binary floating-point library with correct rounding
  2007, ACM Trans. Math. Softw. 33, 2 (Jun. 2007) 13.
  Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P.
  DOI= `http://doi.acm.org/10.1145/1236463.1236468`
  `http://www.mpfr.org`
- MPC - A library for multiprecision complex arithmetic with exact rounding
  Version 1.0.3, February 2015
  Andreas Enge and Mickaël Gastineau and Philippe Théveny and Paul Zimmermann
  `http://mpc.multiprecision.org/`
- Symbolic Computation Software Composability Protocol (SCSCP) specification
  Version 1.3, 2009
  S.Freundt, P.Horn, A.Konovalov, S.Linton, D.Roozemond.
  `http://www.symcomp.org/scscp`
- OpenMath content dictionary scscp1
  D. Roozemond
  `http://www.win.tue.nl/SCIEnce/cds/scscp1.html`
- OpenMath content dictionary scscp2
  D. Roozemond
  `http://www.win.tue.nl/SCIEnce/cds/scscp2.html`
- SCSCP C Library - A C/C++ library for Symbolic Computation Software Composibility Protocol
  Version 1.0.2, May 2016
  M. Gastineau
  `http://www.imcce.fr/trip/scscp/`

Appendix A Supported OpenMath Content Dictionaries

The following list are the OpenMath objects supported by the SCSCP server and client.

CD	**Symbol**
arith1	abs, divide, minus, plus, power, times, unary_minus
alg1	one, zero
bigfloat1	bigflat
arith1	abs, divide, minus, plus, power, times, unary_minus
complex1	argument, complex_cartesian, complex_polar, conjugate, imaginary, real
fieldname1	C, Q, R
interval1	interval_cc, integer_interval, interval
linalg2	matrix, matrixrow, vector
list1	list
logic1	not, and, xor, or, true, false
nums1	e,i, infinity, NaN, pi, rational
polyd1	poly_ring_d_named, SDMP, DMP, term
polyu	poly_u_rep, term
polyr	poly_r_rep, term
setname1	C, N, P, Q, R, Z
transc1	arccos, arccosh, arcsin, arcsinh, arctan, arctanh, cos, cosh, exp, ln, log, sin, sinh, tan, tanh

The following list are the OpenMath objects supported only by the SCSCP server.

CD	**Symbol**
scscp2	get_allowed_heads, get_transient_cd, get_signature, store, retrieve, unbind

The following list are the OpenMath objects supported only by the SCSCP client.

CD	**Symbol**
scscp2	symbol_set, signature, service_description,

Appendix B Index of the global variables

Appendix C Index of the procedures

Appendix D Full index

(Index is nonexistent)

Table of Contents