A BRIEF INTRODUCTION TO MAPLE

by
Christina Christara

December 2001, rev. August 2006

1. What is Maple?

Maple is a high-level interactive software package for mathematical manipulation and problem solving. It is capable of doing **symbolic** computations, such as integration of $\sin(x)$, and, unlike other conventional mathematical software, return its answers in the form of mathematical objects and notation, and not only in the form of numbers, e.g. return $-\cos(x)$ for the integral of $\sin(x)$. Maple can manipulate matrices and vectors in a high-level, object-oriented manner, solve or manipulate linear and non-linear equations in terms of parameters of the equations, differentiate or integrate functions, etc. Maple has rich graphics capabilities and can handle two- and three-dimensional plots. Its plotting facilities allow users to pass a function in a symbolic form as an argument to a plotting routine, rather than passing a set of discrete data.

Maple is a product of the Waterloo Maple Software Corporation (`http://www.maplesoft.com/`). On CDF, we are currently running Maple 8.

2. How to use Maple.

Maple can be started up by either the

```
% maple
```

or the

```
% xmaple
```

commands typed at the Unix shell prompt. The `maple` command starts up Maple in your current window (most likely an `xterm`), and can work on any ASCII terminal. The `xmaple` command starts up Maple in a new window of the console you are on, assuming you are running Xwindows. The `maple` command gives you a simple and light ASCII interface, while `xmaple` gives you a more powerful, but also complex and heavy interface. With `xmaple` you will get one main window with various menus and buttons, and one (or more) inner windows, called **worksheets**. To run `maple` you have to be logged on `werewolf.cdf`, while `xmaple` runs on all CDF workstations. If you run `xmaple` on `werewolf.cdf`, it is suggested that you start it up with `xmaple &`, so that you can re-use the window in which you started it. Note that you can **not** run `xmaple` from a non-Xwindow system. Thus, if you are at home using a PC and Windows, and telnet (ssh) to `werewolf.cdf`, you can run `maple`, but not `xmaple`.

The standard Maple prompt is `>` . In the case of `xmaple`, this prompt appears in the worksheets. In the following, whatever follows `>` on a line starting with `>` is typed by the user at the Maple prompt.

To quit Maple started up by `maple`, type

```
> quit
```

while to quit Maple which has been started up by `xmaple`, click on the `File-Exit` options of the menu.

2.1. Files, functions, procedures and sessions.

Maple has a lot of built-in functions and procedures (commands). It also allows users to define their own. User procedures are saved in files. Functions and procedures may have one or more input parameters and may return one or more output variables. A file with Maple code does not always need to be a procedure. It can also be a sequence of Maple statements, i.e. a session (also called script). Sessions have neither formal input parameters, nor formal output variables.

Maple distinguishes files in two formats: **Maple language (user) format** or **Maple internal format**. If the file is in Maple language format, the effect of reading the file is the same as entering the sequence of statements of the file in a Maple session, without echoing the statements. Maple language files are ASCII files and can be prepared by any editor. If the file is in Maple internal format, reading the file will update the current Maple environment with the contents of the file, without displaying any information. Maple assumes that files ending in `.m` are in Maple internal format.

To read a file, use the `read` command, e.g.

```
> read ‘myfile‘;
```

or

```
> read ‘mydir/myfile‘;
```

Note the **backquotes**.

2.2. Invoking Maple functions and procedures.

To call a function or procedure, type its name followed by the desired input and output arguments. To save the default output variable in a name, precede the call by the output variable name followed by the `:=` sign. If the function or procedure belongs to a package, you should invoke the package first, using the `with` command. For example,

```
> with(linalg):
```

invokes the Linear Algebra package which includes lots of functions useful for matrix manipulation, e.g. LU decomposition, inverse calculation, eigenvalue and eigenvector calculation, etc. Once the above package has been invoked in the current Maple session, and assuming the name `A` is already assigned a matrix,

```
> inverse(A);
```

will display the inverse of `A`, while

```
> iA := inverse(A);
```

will also save the inverse matrix in the variable name `iA`.

 © C. Christara, 2000-2006

2.3. On-line help.

Maple offers a lot of on-line help. Type

```
> ?topic
```

or

```
> ?topic, subtopic;
```

to get information on any Maple topic or subtopic, respectively. For example,

```
> ?linalg
```

will give you general information on the Linear Algebra package and the names of all its subtopics and

```
> ?linalg, inverse;
```

will tell you how to use the *inverse* built-in function that gives the inverse of a matrix. Similarly, you can use `help(topic)` or `help(topic, subtopic)`.

2.4. Command line editing.

Maple (invoked either by `maple` or by `xmaple`) allows command line editing. That is, you can use the arrow keys at the Maple prompt to correct mistyped commands, or to recall previous commands. The following is a summary of the line editing commands:

Up Arrow, Ctrl-P	recall previous line
Down Arrow, Ctrl-N	recall next line
Left Arrow, Ctrl-B	move left one character
Right Arrow, Ctrl-F	move right one character
Ctrl-D	delete character at cursor
Backspace, Delete	delete character before the cursor
Ctrl-W	move right one word
Ctrl-A	move to beginning of line
Ctrl-E	move to end of line
Ctrl-K	delete to end of current line
Ctrl-U	cancel current line

3. Maple statements and operators.

Any Maple statement ends with a `;` (semicolon) or a `:` (colon). To execute the statement, type carriage return after `;` or `:`. When the statement ends with `;` the result of executing the statement is displayed, while when it ends with `:` the statement is executed silently. For example,

```
> a := 4;
```

followed by carriage return, will cause Maple to assign the value 4 to `a` and to echo `a := 4`, while

```
> a := 4:
```

followed by carriage return, will cause Maple only to assign the value 4 to `a`.

To clear the variable name `a` use

```
> unassign('a');
```

while to clear all variable and procedure names use

```
> restart;
```

Comments in Maple code may be written following the # symbol. Maple ignores whatever follows # on a line.

Maple is **case-sensitive**. Thus the name `xyz` is different from `Xyz`.

The following table gives the most common operators in Maple.

Operator	Meaning
:=	assignment
+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation
^	exponentiation
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
=	equal
<>	not equal
and	logical and
or	logical or
not	logical not

4. Working with matrices in Maple.

Matrices in Maple are represented by two-dimensional arrays with row and column indices starting from 1. Maple has a Linear Algebra package, called `linalg`, which includes several matrix manipulation commands. To use this package, we must first invoke it with

```
> with(linalg):
```

Then, we can call any function in the package.

There are two ways to create a matrix: by using the `array` function (which does not require the invocation of the `linalg` package) or by using the `matrix` function which is part of the `linalg` package. For example,

```
> A := array(1..2, 1..3);
```

creates a 3×2 empty array and assings it to `A`. To give initial values to the matrix entries, use, for example,

```
> A := array([[1, 2, 2], [3, 2, 6]]);
```

or

```
> A := array(1..2, 1..3, [[1, 2, 2], [3, 2, 6]]);
```

which both give

```
             [1    2    2]
        A := [             ]
             [3    2    6]
```

In general,

```
> array(indexfcn, bounds, list);
```

creates an array, where `indexfcn` is an indexing function (see further), `bounds` indicate the dimension of the array and `list` gives the initial entries' values. One of `bounds` or `list` is mandatory, the rest of the arguments are optional. The argument `indexfcn` is used to specify certain matrix types, such as `symmetric, antisymmetric, sparse, diagonal` and `identity`. For example,

```
> A := array(identity, 1..2, 1..2);
```

creates a 2×2 identity matrix. Note that `bounds` is of the form `m1..m2, n1..n2`, where `m1, m2, n1` and `n2` are positive integers. Also, `list` is surrounded by square brackets, the matrix elements are listed row-by-row, separated by commas, and the elements of each row are

also enclosed in square brackets, and separated by commas.

Alternatively, if the `linalg` package has been already invoked,

```
> A := matrix(m, n, list);
```

creates a $m \times n$ matrix with its entries initialized in `list`. The form of `list` is the same as that of `list` in the `array` function call.

To see the contents of an array/matrix, use

```
> print(array_name);
```

To refer to a certain entry of a matrix, use square brackets, and specify the row and column index, i.e. `A[i, j]` denotes the (i,j)th element of `A`.

With the `linalg` package, the functions `rowdim` and `coldim` can be used to find the dimension of a matrix. For example,

```
> ir := rowdim(A);
> ic := coldim(A);
```

will put the number of rows and columns of `A` in variables `ir` and `ic`, respectively.

The following are some of the functions in `linalg`, that do basic matrix operations:

matadd(A, B, c1, c2): computes c1*A + c2*B, where A and B are matrices (or vectors) of matching dimensions, and c1 and c2 are optional scalars. If c1 and c2 are missing, they are assumed to be 1.

multiply(A, B, ...): computes the product of a sequence of matrices (or vectors), assuming the dimensions are appropriate.

transpose(A): gives the transpose of matrix (or vector) A.

inverse(A): computes the inverse of the square matrix A. If the matrix is not invertible a message is displayed.

det(A): computes the determinant of the square matrix A.

5. Saving output and plots from Maple.

To save all Maple output in a file, use the commands `appendto` or `writeto`. For example,

```
> appendto('file_name');
```

will cause all future output to be saved in file `file_name`. Thus the output will not be echoed on the terminal/window. If file `file_name` does not exist, it will be created, while if it exists, the output will be appended to it. On the other hand,

```
> writeto('file_name');
```

will have a similar effect except that, if file `file_name` exists, it will be **overwritten**. To restore echoing the output on the terminal use

```
> writeto(terminal);
```

After saving the output to a file and restoring echoing, the file has to be closed by using

```
> close('file_name');
```

Plots from Maple can be saved using the `interface` command. Thus,

```
> interface(plotdevice = postscript,
  plotoutput = 'myfile.ps');
```

will save the next plot in the file `myfile.ps` in postscript format. If `myfile.ps` exists, it will be **overwritten**. The file `myfile.ps` can be printed on the CDF printer by issuing the command `lpr myfile.ps` at the Unix shell prompt. To reset the plot output to the terminal, use

```
> interface(plotdevice = default);
```

To get the name of the current plotting device use

```
> interface(plotdevice);
```

To get the name of the output file, where the plot will be stored, use

```
> interface(plotoutput);
```

6. Some examples.

```
> # invoke the linalg package
> with(linalg):
> # define a matrix
> A := matrix([[1, x], [2, y]]);
                    [1    x]
               A := [      ]
                    [2    y]
> # see the contents of A
> print(A);
                [1    x]
                [      ]
                [2    y]
> # find the determinant of A
> det(A);
                y - 2 x
> # find the inverse of A
> inverse(A);
      [     y            x     ]
      [- --------     -------- ]
      [  -y + 2 x     -y + 2 x ]
      [                        ]
      [    2              1    ]
      [ --------     - --------]
      [ -y + 2 x       -y + 2 x]
> # define another matrix
> B := matrix([[1, 0], [1, 1]]);
                   [1    0]
              B := [      ]
                   [1    1]
> # add matrices
> C := matadd(A, B);
                [2      x  ]
         C :=   [          ]
                [3    y + 1]
> # define yet another matrix
> E := matrix([[1, 2, 3], [4, 5, 6]]);
                [1    2    3]
           E := [           ]
                [4    5    6]
> # add matrices of incompatible dimensions
> F := matadd(A, E);
Error, (in matadd) matrix dimensions incompatible
> # define equations
> eq1 := a11*x1 + a12*x2 = b1;
        eq1 := a11 x1 + a12 x2 = b1
> eq2 := a21*x1 + a22*x2 = b2;
        eq2 := a21 x1 + a22 x2 = b2
```

```
> # solve equations in terms of parameters
> solve({eq1, eq2}, {x1, x2});
        -a22 b1 + b2 a12         a11 b2 - b1 a21
 {x1 = - ----------------, x2 = ----------------}
        a11 a22 - a21 a12       a11 a22 - a21 a12
> # define and solve equations (again)
> eq1 := 3*x1 + 6*x2 = 5;
        eq1 := 3 x1 + 6 x2 = 5
> eq2 := -2*x1 + x2 = 0;
        eq2 := -2 x1 + x2 = 0
> solve({eq1, eq2}, {x1, x2});
         {x2 = 2/3, x1 = 1/3}
> # solve non-linear equations
> solve(sin(x)-0.5*x, x);
              1.895494267
> solve(sin(x)-1./2.*x, x);
              1.895494267
> # 1/2 is a symbolic expression
> # in this case, the result is not very useful
> solve(sin(x)-1/2*x, x);
        RootOf(_Z - 2 sin(_Z))
> # get the actual value in floating-point
> evalf(solve(sin(x)-1/2*x, x));
              1.895494267

> # differentiate a function
> diff(sin(x), x);
                cos(x)
> # integrate a function
> int(tan(x), x);
              -ln(cos(x))
> # integrate a function in an interval
> int(1+x^2, x = 1..2);
                 10/3
> # expand a product or a math function
> expand((x+1)^5);
   5      4       3       2
  x  + 5 x  + 10 x  + 10 x  + 5 x + 1
> expand(sin(x+y));
     sin(x) cos(y) + cos(x) sin(y)
> # simplify an expression
> simplify((x^5-1)/(x-1));
          4    3    2
         x  + x  + x  + x + 1

> # plotting
> # save current output file and device
> oldout := interface(plotoutput);
          oldout := terminal
> olddev := interface(plotdevice);
             olddev := x11
> # get the plot in a file in postscript format
> interface(plotdevice = postscript,
  plotoutput = ‘examples.ps‘);
> # a 2D plot of sin(x) from -Pi to Pi
> # Pi is reserved name for the number pi.
> plot(sin(x), x = -Pi..Pi);
> # get subsequent plots on the X window system
> interface(plotdevice = x11, plotoutput = oldout);
> # a 3D plot
> plot3d(sin(x*y), x = -Pi..Pi, y = -Pi..Pi);
> # get a better resolution
> plot3d(sin(x*y), x = -Pi..Pi, y = -Pi..Pi,
  grid = [50, 50]);
> # restore the device
> interface(plotdevice = olddev);

> # using a procedure in a user prepared file
> # Assume the following is in file mat_add.mp
# An example of a user defined procedure
# Maple has a built-in function for matrix addition
# but assume we want to write our own.
mat_add := proc(A, B)

local A_row, A_col, B_row, B_col, bounds, C, i, j;

A_row := rowdim(A);
A_col := coldim(A);
B_row := rowdim(B);
B_col := coldim(B);
# The syntax of the if statement is
# if conditional_expression then statement_sequence
# |elif conditional_expression then statement_sequence|
# |else statement_sequence| fi;
# where whatever is in | | is optional.
if A_row <> B_row or A_col <> B_col then
# The ERROR function causes an immediate return to
# the point where the current procedure was invoked.
    ERROR(‘matrix dimensions do not match‘);
fi;
C := matrix(A_row, A_col);
# A typical syntax of the for loop is
# for i |from j by k| to l do statement_sequence od;
# If "from j", "by k" are missing, j, k default to 1.
for i to A_row do
    for j to A_col do
        C[i, j] := A[i, j] + B[i, j];
    od
od;
# Return a matrix definition.
return(op(C)); # op is the operand function
end;
# end of file mat_add.mp
> # now read the file and call our own procedure
> read ‘mat_add.mp‘:
> C := mat_add(A, B);
                 [2      x  ]
           C := [            ]
                 [3    y + 1]
```

Additional references

There are lots of books on Maple, as well as help available on the internet. See, for example, http://www.mapleapps.com/ http://www.maplesoft.com/academic/students/tutorials.aspx and http://www.indiana.edu/~statmath/math/maple/.

 © C. Christara, 2000-2006