

## A Maple User's Guide

Phil Lucht

Rimrock Digital Technology, Salt Lake City, Utah 84102

last update: Feb 16, 2015

[rimrock@xmission.com](mailto:rimrock@xmission.com)

This is a combination "basic man pages" and user guide for Maple V Release 5 (1997). Although Maple has certainly moved on (currently Maple 18), this document tries to explain how the Maple system "thinks" internally, and that really has not changed much in the last 18 years. It is true that many wonderful User Interface enhancements have been added: context menus, a graphic pane for everything under the sun, fancier entry and plotting methods, modern Help system, document mode, and of course there are new internal features as well. There is after all an ongoing competition with Mathematica and other vendors. However, if you peruse the Maple 18 User Manual you will discover that, below this glossy layer, things are very much the same ( [http://www.maplesoft.com/documentation\\_center/](http://www.maplesoft.com/documentation_center/) ), and you won't find discussed many of the subtle issues presented below.

Maple 18 can convert "classic" Maple .mws files into its current .mws file format, by the way.

The *Maple V* Help system, although a bit strange, has its strong points and has excellent finely detailed man pages for each Maple concept, and the reader should consult such Help pages for the last word. These pages *usually* have examples which can be copied and pasted and run, and have references to related Help pages. For presentation I have selected certain simple Maple functions and commands that illustrate how Maple works. I would guess there are over 1,000 man pages, including more than 500 just for function descriptions. The GUI itself is a bit strange and takes some getting used to, hence lots of screen clip examples below. After some practice, the GUI is quite efficient and intuitive.

There are many free PDF Maple user guides on the web, even for ancient Maple V, and I have collected several of them. I think you will find that this document addresses many issues omitted in these other documents. There are of course also whole books on Maple.

I should mention that in addition to Maple, MATLAB and Mathematica, there are many other freeware and proprietary "symbolic computer algebra systems" floating around. Wiki currently lists about 30 such systems [http://en.wikipedia.org/wiki/List\\_of\\_computer\\_algebra\\_systems](http://en.wikipedia.org/wiki/List_of_computer_algebra_systems) , though this list omits the popular Octave GNU freeware. The term "symbolic" might rule out some systems if they can only do things "numerically".

This table of contents has live links.

GENERAL ASPECTS OF MAPLE .....	5
Keyboard Input Problem.....	5
Restart .....	5
Mousing Around.....	5
Case Sensitivity and Variable Names .....	5
lock file .....	6
Multiple Instances versus Multiple Worksheets .....	6
Linkage to Procedures and Packages .....	6
Management of your own Maple code samples.....	7
The Help System.....	7
Output Display Options (for the worksheet window) ; : .....	8
maplev5.ini file saves settings (not registry) .....	9

styles .....	9
execution groups    statements command lines assignment statements .....	10
enter    shift-enter    C-k (new above)    C-j (new below)    F4 (combine)    F3 (split) .....	10
Adding a new command inside an execution group. ....	11
Two kinds of input notation .....	12
Disabling statements with the Leaf Icon .....	12
Commenting Text      C-t (text mode) C-m (math input mode) # (comment symbol) .....	13
C-shift k    C-shift j .....	13
Pasting Maple code from one worksheet to another .....	15
Pasting HTML code from the web.....	15
section, subsection, indent, outdent .....	16
Greek letters .....	17
Constants :    false gamma infinity true Catalan FAIL Pi I.....	18
ditto operators    %    %%    %%% and %n.....	18
Inert Forms of Operators    value.....	19
macro    alias .....	20
How to replace a default symbol name with another symbol and free up its name .....	21
EXPRESSION MANIPULATION AND EVALUATION IN MAPLE.....	22
type.....	22
Numbers and Base Conversion .....	23
operators.....	25
trunc round frac floor ceil mod .....	26
sequences lists sets      seq    repeat operator \$.....	26
Conversion between Sequences, Sets, Lists, and Arrays .....	29
list      // revisited: doing math with elements of a list .....	31
eval    Eval evalf evalhf .....	32
subs    algsubs.....	34
lhs    rhs    numer    denom.....	35
unprotect    type reserved words and letters in Maple.....	35
unassign .....	36
assume      is about additionally .....	36
collect, coeff, normal, simplify, expand, combine, factor, rationalize, convert.....	40
collect.....	40
normal .....	41
simplify .....	42
expand and combine .....	42
factor .....	43
rationalize.....	43
operands    nops(s) op(3,s).....	44
Suppressing function arguments .....	45
Frustration with Simplification .....	47
Algebra with complex numbers    evalc.....	47
Algebra with trig functions    subs .....	49
NON MATRIX RELATED MATH FUNCTIONS/OPERATIONS .....	50
How to Define a Function of Arguments.....	50
(1) the mapping operator " ->"    user defined functions .....	50
(2) unapply    use to "functionalize" an existing expression.....	52

(3) unapply    used to create the derivative of a function .....	53
fsolve.....	54
solve    variables which are "equations" M equations in M unknowns .....	54
Two meanings of _Z : (1) dummy in RootOf; (2) "any integer" .....	55
dsolve    solving differential equations.....	56
rsolve    solving difference equations (recursion equations).....	66
sum and add    product and mul .....	67
Integration    int Int.....	67
Differentiation    diff Diff D .....	70
piecewise and periodic functions .....	73
Integral and Series Transforms .....	74
dchange    doing PDE work with coordinate transformations .....	75
series    series expansions about a point.....	75
Differential Operators: curl diverge grad laplacian vector Laplacian.....	76
MATRIX/VECTOR RELATED MATH FUNCTIONS/OPERATIONS.....	79
table.....	79
array .....	80
matrix    "matrix vectors" evalm.....	85
The zero matrix and the identity matrix in evaluation .....	87
Stand-alone zero matrix and identity matrix.....	88
Creating a matrix from a function f(i,j).....	88
Creating a matrix from a function f(i,j) using inert operators.....	89
Displaying a Matrix: Unexpected Behaviors.....	90
Maple vectors    convert.....	91
Displaying a Maple vector: Unexpected Behaviors.....	94
Preliminary comments on the linalg package .....	95
grad    transpose .....	96
row col transpose    inverse det diag .....	97
Speed of matrix inversion.....	98
linear algebra using matrices    eigenvectors eigenvalues linsolve.....	99
PLOTTING .....	100
plot .....	100
spacecurve: A 3D parametric plot.....	103
plot3d .....	103
gradplot .....	104
gradplot3d .....	105
fieldplot3D .....	105
fieldplot.....	106
logplot .....	106
loglogplot .....	106
semilogplot.....	106
histogram.....	106
pointplot.....	106
listplot .....	108
animate.....	108
Overlaying multiple plots: lists, PLOT, display() and gridlines .....	109
PROCEDURES AND PROGRAMMING.....	114

Procedures and Programming	Algol.....	114
The for/while/do statements	for do od from to by while.....	114
The if statement	if then and or else elif fi.....	117
Some simple procedure examples	proc end RETURN local global showstat.....	119
Example 1:	.....	119
Example 2:	.....	119
Example 3:	.....	120
Example 4: Showing a mysterious problem and how to fix it: single quotes and deferral	.....	121
Example 5: the permutation tensor	.....	123
Example 6: the arctan2Pi function	.....	123
Example 7: One from the web	.....	124
Example 8: Looking at Maple's own code	.....	125
Debugging with the Debugger	.....	127
print printf sprintf etc	.....	128
MISC TOPICS	.....	130
Spreadsheets	.....	130
Appendix 1: The New User's Tour	.....	130
About Maple	.....	132
Appendix 2: List of Functions in Maple	.....	133
MATLAB linkage	.....	136
Fast Fourier Transform	.....	136
Maple Hot Keys Summary for Windows	.....	136

---

## GENERAL ASPECTS OF MAPLE

---

### Keyboard Input Problem

If after a Windows installation Maple fails to accept keyboard input on its worksheet, try renaming the following file to some other name,

C:\Program Files\Maple V Release 5.1\BIN.WNT\OPENGL32.DLL

---

### Restart

```
[> restart;
```

This Maple command makes Maple forget everything (such as variable names, procedures, etc) and start over. I just wanted to put this up front since I don't know where else to put it. It is a system reset for Maple.

---

### Mousing Around

In Windows, I use an excellent freeware **WizMouse** which allows one to scroll any window pane using a mouse scroll wheel without having to click in it first. Because old Maple V is somewhat of a legacy program (relative to XP or System 7), WizMouse scrolling does not work right. This is repaired by having another freeware **KatMouse** installed and turning it on for Maple work. Both these mouse control programs have little tool tray icons which make turning them on and off very easy.

---

### Case Sensitivity and Variable Names

Maple is very case sensitive. See Pi discussion below! Visual Basic is not case sensitive, by the way, although it retains your caps for cosmetic use. Variable names entered in the usual manner must begin with a letter, and this is true of any "name" you use in Maple, such as a function name. From the horse's mouth:

- A name is usually a symbol, which in its simplest form is a letter followed by zero or more letters, digits, and underscores with lower and upper case letters distinct. The maximum length of a name is system dependent. On 32-bit platforms, it is 524,271 characters; on 64-bit platforms, it is 34,359,738,335 characters.

I suspect that most users are willing to accept this limitation on variable name length.

If you want to have a name begin with something other than a letter, it can be done with tick marks as shown here. Such a name must always be referenced with tick marks.

```

joe := `lbob`;
type(joe,name);

```

*joe = lbob*

*true*

---

```

> lbob := 6;
missing operator or ` `
> `lbob` := 6;

```

*lbob = 6*

## lock file

If you kill a Maple instance and launch a new one too quickly, you sometimes get a message concerning your Maple license and Maple won't do anything. Another situation is that an error condition might prevent you from exiting Maple -- an infinite loop of dialog windows for example -- in which case you have to terminate Maple by brute force (try the Stop icon first, or try typing "quit" ). In both these situations, a "lock file" is left in a certain location and you have to manually delete that file. The file is called **lock.0** (size 0 bytes) and is located in the LICENSE folder in the Maple install directory. The locking mechanism does not prevent you from running as many instances of Maple as you want on the same computer. This file is present any time one or more Maple instances are running.

## Multiple Instances versus Multiple Worksheets

By "instance" we mean you launch an "instance" of the Maple application by double-clicking on the Maple icon. You can have as many Maple instances up and running as you want. Below we suggest you always have one instance dedicated to the Help system, so you are always running at least two Maple instances. The big fact is that different Maple instances "don't know about each other". If you set a := 4 in one instance, other instances have no idea what a is. The instances are completely isolated from each other, as you might expect. Restarting one (restart command) does not restart others, for example.

On the other hand, you can have multiple Maple worksheets going in one instance of Maple, and these worksheets *know all about each other!* Such multiple worksheets live in multiple windows which can be cascaded and so on, just as in any Windows multi-document application. You might have one worksheet which you run once to define a set of procedures that you then use in other worksheets. Eventually you can create your own "packages" if you want and avoid having to do this. (More below on packages. ) The main idea is that, when you have multiple worksheets going within the same Maple instance, all those worksheets are really just one big worksheet.

## Linkage to Procedures and Packages

Suppose you have a Maple file joe.mws which defines a procedure joe(x,y,z), and that you have pulled up that file and executed the procedure. This is like compiling a program, and now this procedure joe(x,y,z) is available to be called by other .mws files which are brought up in the same Maple program instance. In other words, the function joe(x,y,z) has been added to Maple's library of functions such as sin(x).

Maple has many libraries (called "packages") some of which are not linked in when you just start Maple. To activate a library of this kind, use the "with" command such as **with(linalg)** to get the routines of the linear algebra package. See below for more details on procedure writing.

---

## Management of your own Maple code samples

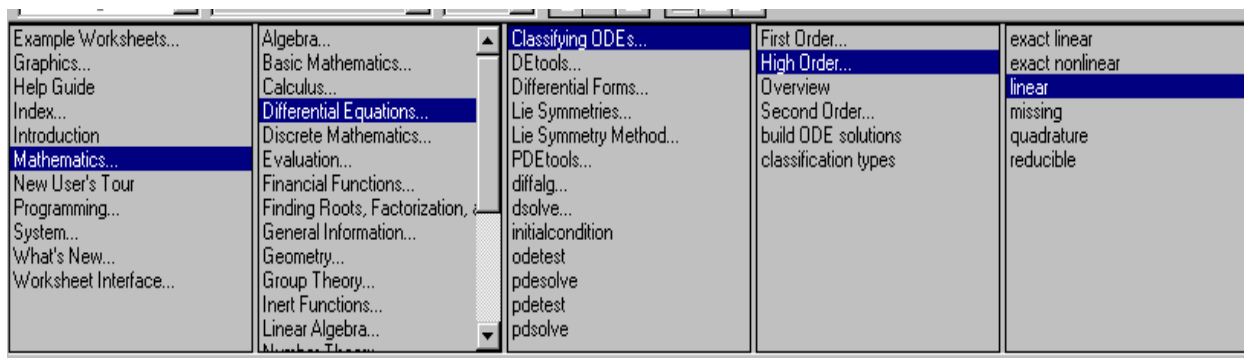
I usually store a Maple file locally, where it was created, but periodically I can search my entire drive for mws files, and then select them all and copy them into a little Maple folder in my "repository". These files are generally not very big, so there is not much cost in this duplication.

Once you have this consolidated folder, you can do a quick Windows Explorer search for all examples of procedures, say, by searching the folder for files containing "proc". Then you can open the found files one at a time to see procedure examples. See "lock" section above if there are problems.

Once inside Maple, be warned that C-f, the find command, is "directional" and does not cycle back from end to beginning, so it is easy to miss something.

---

## The Help System



**Advice:** Run a **whole separate instance of Maple as a help window**. Just type ? in a (white) worksheet window to move to the (yellow) help side of things. In this way, you can **cut and paste examples** from the help world into a live worksheet to see what they do. This works better than doing a horizontal tiling of help + worksheet windows.

Help is not done in the Windows standard, and it is a bit painful to use.

Once you get a help window up, you see a tree/hierarchy of topics at the top (as shown above). This is good, but not if you are looking for some topic and you don't know where it is in the hierarchy. *Most* leaf items in the hierarchy have a man (Help) page, and that is called a "topic" and shows up when you do a "topic search". When you click on different items in the hierarchy, you go there in the same help window. However, some items are *not* man pages, and when you click on them, you unexpectedly fly off to a new window, so this is a bit confusing. As long as this does not happen, you have back and forward buttons on the Help GUI to navigate as in a browser.

The man pages have a yellow background and are read-only and non-executable. So if you want to try a code sample, you cut and paste that code into a white worksheet window. Remember that you can select-all/delete to first clear the window for doing a new paste test. Sometimes a restart helps.

As just noted, you can do the "topic search" to find specific man pages. The "full text search" is **not very smart**. It just searches for ANY of the words you put in a list of words, and double quotes are ignored. Searching is on whole words only, it does not find partial words like gradplo. To prevent the user from getting hundreds of hits, certain words cannot be searched on! For example, if you want to learn

about how a "for loop" works, the full text search returns nothing on "for" , but the topic search does know about "for".

Help is an area needing improvement (and is no doubt improved in current Maple 14).

The "same window" check box prevents a *new* Help window from coming up each time you do a Help search. Otherwise help windows proliferate. They are all on top of each other by default unless you tile or cascade them in the usual manner. I don't find this to be a problem.

The OK button gets you to your selected help screen and kills the help GUI.

The Apply button is better, it lets the Help GUI stay up. This is better if you are looking through lots of Help windows searching for something because the full text search is dumb. For example, suppose you want to find out where the concept of "inert" is defined? A full text search brings up many items because inert is referred to in many places. Sometimes a web search like "maple inert" gets you there faster.

The Help system includes a set of excellent tutorials, here is a list

- (1) [WORKING THROUGH THE NEW USER'S TOUR](#)
- (2) [THE WORKSHEET ENVIRONMENT](#)
- (3) [NUMERICAL CALCULATIONS](#)
- (4) [ALGEBRAIC COMPUTATIONS](#)
- (5) [GRAPHICS](#)
- (6) [CALCULUS](#)
- (7) [DIFFERENTIAL EQUATIONS](#)
- (8) [LINEAR ALGEBRA](#)
- (9) [STATISTICS AND FINANCE](#)
- (10) [PROGRAMMING](#)
- (11) [ON-LINE HELP](#)
- (12) [SUMMARY](#)

Unlike the Help man pages, these tutorials are on active worksheets so you can just execute things willy-nilly to see what they do.

---

### Output Display Options (for the worksheet window) ; :

When you execute a command, if the command line ends with a ; the result is displayed in one of the formats discussed here, but if the command line ends with a : that display is suppressed.

You always enter expressions in linear text such as  $f := \sin(x)^2$ . This is known as the **Maple notation**. This is probably how Maple stores everything internally.

When you execute a command, the result is displayed (sprayed onto the screen) in one of three formats, and you control this from the Options menu. You must set the Option before you execute a command. Once output is displayed, changing the option does not change it's appearance. You must re-execute the command to get a new appearance.



(1) the **Maple notation** already mentioned.

Diff(sin(x),x)

(2) the **typeset notation** which puts out a bitmap image

$$\frac{\partial}{\partial x} \sin(x)$$

(3) the **character notation**, which constructs a 2D picture out of just characters (think old emails)

d  
-- sin(x)  
dx

(4) The fourth "option" is called **editable math** which means you still get nice the typeset notation, but if you edit a change into it ( using the "standard math editor" see below) and hit return, you get a new command created on the next line with that change made.

I have had no use for notation (3) and always use (2). The one case where (1) is useful is if you want to grab a complex output expression (which you don't want to retype) and paste it somewhere else.

---

### **maplev5.ini file saves settings (not registry)**

There are some registry settings in the HKEY/Software/Waterloo area, but this is not where the interesting stuff is located. Maple (for Windows) uses a conventional INI file called maplev5.ini located at the top level of the operating system folder, such as C:/windows. This file is altered when you do File/Save Settings from a worksheet menu (Save Settings is not present in the help menu system). The Auto Save Settings is supposed to save preferences you have made on each exit, but I am not sure this works right. The file maplev5.ini controls for example how the sub-windows fill the Maple main window when you fire up Maple (cascade, tile, etc). I think some information is saved separately in each .mws file such as specific font choices for that file. Go look at the .ini file to see what it has.

---

### **styles**

Each document can have custom "styles" for how you want different types of Maple text to appear. I think the defaults for styles are part of the main code. A worksheet file .mws can contain overrides for styles, and you can see these with a text editor. See Format/Styles, select a style and then do "modify" to see what things are set to. (An asterisk in that GUI means the default of whatever it is.)



(4)	<pre>&gt; y := cos(x);   z := sin(x);       y := cos(x)       z := sin(x)</pre>	(5)	<pre>&gt; y := cos(x); &gt; z := sin(x);       y := cos(x)       z := sin(x)</pre>
-----	---	-----	--

We might also have these two statements on different command lines as in (5). In either case (4) or (5), we would say that the two statements are in the "same execution group".

So, an execution groups is a set of (one or more) Maple commands all in one left-edge brace, and all are executed at once (well, in top to bottom order of course) when you hit a return with the cursor sitting in some red text within the group. Notice that the blue output display is shown within the left edge brace, it is part of the execution group.

When you want to add another command to an execution group, do "**shift-enter**" and type it, this makes a situation like (4) from (2). An "**enter**" without the shift executes whatever is in the group so far.

Example: If you are writing a "for loop" which contains multiple commands, you want to use shift-enter after entering each line to move to the next line. **You must** have the cursor exactly after the last character on a line before you do shift-enter, otherwise you create an extra blank line which you then have to delete. If you forget the shift, the group executes and you have to recover from your surprise and then keep editing in new lines.

To insert a **new "execution group"** above or below the current one, use C-k or C-j. (C = control). Notice the odd reverse order of the letters j,k for below,above. So k is above. (C-k or C-j very useful! )

The "**enter**" key as noted causes the current execution group to "execute" and moves the vertical insertion point down to the start of the last command of the next execution group. If there is no next execution group, it creates a new empty one: a new brace and a prompt.

If you have a bunch of separate execution groups you want to **combine into one**, select them all together and then hit **F4**.

If you want to split an execution group in two, put the insertion point at the start of a command and hit **F3**. The break occurs just above this command.

When you end a statement with a **:** instead of a **;** (both **statement separators**), the "output" of that statement does not appear in an interactive session. This is useful especially if the expression generated is 500 lines long, say. Or if you are happy with that part of the code and don't need to see its output every time you run through the groups.

In our examples above, the **:=** combination "**assigns**" what is on the right to the variable which is on the **left**. Thus, it is an **assignment statement**.

### **Adding a new command inside an execution group.**

We know we can add a new line to a command using the shift-enter method, but suppose we want to add a *new command* within an execution group? A good way is this:

- (1) cursor at the start of a command above which you want to enter a new command
- (2) hit F3 to put a group break above this command
- (3) C-k to put a new empty group above your command, enter your command
- (4) select the execution groups of interest and combine them into one with F4.

```

[> a := 1;
> b := 3;
[> a := 1;
> b := 3;
[> a := 1;
> |
> c := 3;
[> a := 1;
> b := 2;
> c := 3;
[> a := 1;
> b := 2;
> c := 3;

```

If you want to delete an unwanted blank line, sometimes a **triple click** will select it then delete will delete it. Sometimes it takes work to select a worksheet region in Maple for deletion.

### Two kinds of input notation.

So far, our various commands have all been appearing in "Maple notation". They can also be displayed in "standard math notation" which looks nicer but is harder to edit. Whenever the cursor is placed inside a statement, an extra tool bar appears at the top of the window with some icons. Suppose we have the situation shown in (6)

(6)

```

[> y := cos(x);
> z := int(tan(x), x);
      y := cos(x)
      z := -ln(cos(x))

```

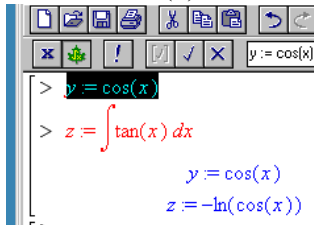
(7)

```

[> y = cos(x)
> z = ∫ tan(x) dx
      y = cos(x)
      z = -ln(cos(x))

```

(8)



```

[> y := cos(x)
> z := ∫ tan(x) dx
      y := cos(x)
      z := -ln(cos(x))

```

We select these red input equations one at a time and hit the black X icon and this toggles them to the "standard math input" display mode and you get (7). Notice that the bold red text of the Maple mode is now non-bold Math-like text, and things like integral signs appear. The output is the same. [ Oddly, when standard math *output* is used, they call it "typeset notation", see Options/Output Display.] If you select some of this standard math input text, it appears in an Excel-like edit white box in the tool bar, as shown for example in (8), and you can edit it there. Notice the black X icon just referred to. You can edit the command in the white window if you want. [ This is the same editor that you use to edit output in the editable math output mode noted above. ]

Having described all this, I have to say that I never use anything but the default input mode.

### Disabling statements with the Leaf Icon.

If you select a statement (in either input mode) and hit the Maple leaf icon, the statement goes black which means it is temporarily turned into "inert text". This is a way to temporarily disable commands in an execution group. Starting from (6), I have made inert the integral statement, and now it is no longer executed when the group is executed.

```

[> y := cos(x);
z := int(tan(x), x);
      y := cos(x)

```

Do not confuse the word "inert" here with the concept described later that functions become inert versions of themselves if you capitalize the first letter, such as Int versus int.

Another way to disable statements is to **prefix them with a # sign**, as done in some other languages. This works fine. The difference is that, as you execute groups one at a time in sequence, a group with black text is completely spaced over, whereas a # statement is still "executed" but does nothing.

<b>Commenting Text</b>	<b>C-t (text mode)</b> <b>C-shift k</b>	<b>C-m (math input mode)</b> <b>C-shift j</b>	<b># (comment symbol)</b>
------------------------	--	--	---------------------------

The C-t takes you to "text mode" (used for adding comments), while C-m takes you back to math input mode.

*In theory* it is nice to be able to add comment text anywhere you want, but in practice it is not very easy to do except in the simple ways described at the end of this section.

Here is a problematic example. Suppose we are entering a multi-line do loop that will eventually look like this (we use shift-enter to create each new line, as explained earlier)

```
> for n from 1 to 3 do
  a[n] := n^2;
od;
                                     a1 = 1
                                     a2 = 4
                                     a3 = 9
```

After we enter the above code, it is not too hard to go back and add a "text" comment to the right of each line. To do this, one places the insertion point *just after* the last command character of a line, hits C-t, and then enters the text, perhaps starting with lots of spaces to move the comment to the right. Thus:

```
> for n from 1 to 3 do      first line of do loop
  a[n] := n^2;             second line of do loop
od;                        last line of do loop
```

However, if you try to enter these commands "on the fly" after you enter each command, it is very painful. Here is one way to do it:

- (1) place the insertion | cursor just after the last character of the comment text
- (2) hit shift-enter to move to the next line
- (3) hit C-m to get back into math input mode. A ? appears and the white Excel entry window appears.
- (4) type x into the white box, hit the check icon, and the question mark in the code becomes an x.
- (6) hit the leftmost black x icon, and then the leaf icon. You now have a red command x.
- (7) edit the red x to be the command you want.

Even if you "post comment", if you want to add a new line to the do loop, you have to go through all of the above nonsense. Due to this complexity, it is a lot simpler to just add comments using the usual # method, and this can be done on the fly and there are no problems at all. Then we have

```
[ > for n from 1 to 3 do      # first line of do loop
      a[n] := n^2;          # second line of do loop
od;                          # third line of do loop
```

As usual, to move to the next line place the insertion point just after the last character on the line, hit shift-enter, then type your next line, which you may choose to start with some spaces as for the middle line above. There is no fancy "auto indent" action as in fancy text editors.

So is there any practical use for the Text mode? Yes, and that is in making a "block comment", so to speak. Let's say we have the above code, and we want to put a nice comment above it saying what the do loop is doing. To do this, with the cursor somewhere in the group, do C-k to create a new execution group above the for loop group. This creates the expected [ > . Then do C-t (the prompt goes away) and enter your text, as many lines as you like, for example

```
[ This do loop computes the foobar of the dingle arm.
Written by John Small on Nov 3,2010.
[ > for n from 1 to 3 do      # first line of do loop
      a[n] := n^2;          # second line of do loop
od;                          # third line of do loop
```

Notice that the above method puts the text in its own execution group, which of course has nothing in it that can really be executed.

When you are dealing with "text", Maple goes into a little word processor mode so you can select fonts and sizes in the usual manner say of Word, using an icon bar that appears at the top of the screen. Default is 12 point Times New Roman which in Maple is relatively small. So once you have some text, you can format it as you like just as you do in Word. For example,

```
[ This do loop computes the foobar of the dingle arm.
  Written by John Small on Nov 3,2010.
[ > for n from 1 to 3 do      # first line of do loop
      a[n] := n^2;          # second line of do loop
od;                          # third line of do loop
```

The comment text is black because that is the *style* for comment text (which can be altered).

Probably at the very top of any Maple worksheet you should put some kind of block comment like this so you can know a year later what the worksheet is trying to do.

If you want to add text at the start of an existing execution group (*within* the execution group), here is a nice way. With cursor anywhere in the group, do **control-shift-k** and just type in your text. Similarly, **control-shift-j** adds text at the end of your execution group.

```

[ Control-shift k and then added this comment
> y := cos(x);
[ Control-shift j and then added this comment

```

You can open up space *between* execution groups and put text there as well. The trick to doing this is to select an execution group bracket so it becomes doubled, then use control-shift-k,j to open up the space above or below, then just type your comment into that blank space:

```

[> a := 2;
[> b := 3;
[> a := 2;
Comment text in space between execution groups
[> b := 3;

```

These are all deep mysteries revealed!

Oh yes, one more thing. You can "comment" code by placing "screen clips" into the code, created by OneNote or EasyCapture or any other method. Put screen clips into a comment block area.

### **Pasting Maple code from one worksheet to another**

This works perfectly, you don't have to adjust anything. Somehow you are going from mws format to the same format (in the next item, we have to go from html to mws, which is a problem).

The help system has lots of sample code with a yellow background. Somehow they have made this code non-executable in the help system, but it must be valid mws format code, because you can copy and paste it into a white worksheet and the copy works perfectly.

If you paste Maple code from Maple into a text editor, then paste it back into a new Maple window, things don't work well, similar to the HTML situation described next.

### **Pasting HTML code from the web**

Sample code in a form you can see is often presented in HTML, since this is a "save as" option in Maple. Sometimes a corresponding mws (Maple work sheet) file is not available. If you try to cut and paste this code into Maple, the result is a little odd. For example, on the left below is some HTML code on a web page I found,

```

> H := Heaviside;
      H := Heaviside
> f := t -> H(t - Pi/6)*sin(2*t);
      f := t -> H(t - Pi/6)*sin(2*t)
> laplace(f(t), t, s);
      1/2 * e^(-s*pi/6) * (2 + sqrt(3)*s) / (s^2 + 4)

```

```

> > H := Heaviside;
      H := Heaviside
> f := t -> H(t - Pi/6)*sin(2*t);
      f := proc (t) options operator, arrow;
      H(t-1/6*Pi)*sin(2*t) end proc
> laplace(f(t), t, s);
      1/2*exp(-1/6*s*Pi)*(2+3^(1/2)*s)/(s^2+4)
>

```

But when I cut and paste this into Maple, I get what you see on the right above. Everything is in one execution group, and the blue output now appears as illegal junk text, and the > symbols are not genuine, they are just dead characters. Here is what you have to do to clean this up:

- first, paste the code directly into a Maple window as is, getting as shown on the right above
- for each line with a command (that is, lines beginning with a bogus red >), select the > and any white space to its right up to the start of the command, hit delete, and then hit F3. This puts each command in a separate execution group, as shown on the left.
- then select and delete the garbage text that was the former Maple output
- execute the commands and appearance should then duplicate what was on the web page.
- if you want to rejoin into a single execution group, then select all the command lines of interest, and hit F4.

### section, subsection, indent, outdent

This is a higher level structure which you can collapse out of sight. If you have some existing code and you want to put it into a section, select the code and do **Format/Indent**:

```

[> a := 1:
[> b := 2:
[> c := 3:

```

```

[-] |
    |
    | [> a := 1:
    | [> b := 2:
    | [> c := 3:
    |
    |
    |

```

```

[-] Jobob
    |
    | [> a := 1:
    | [> b := 2:
    | [> c := 3:
    |
    |
    |

```

```

[+] Jobob

```

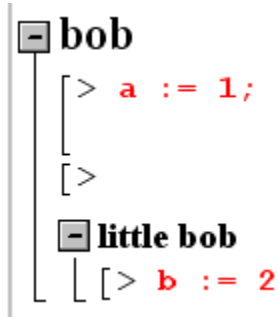
Then just type a name for the section. Hit the - to "collapse" the section as on the right, so it is out of the way. You can select the code you made and restore it to top level by doing **Format/Outdent**.

If the code does not exist yet, you can do **Insert/Section** to create a new section, name it, and then just start typing commands:





The trick here is to do C-j to get to the third picture so you can add code inside the section. You could then add a subsection within bob by doing **Insert/Subsection**, name it, do C-j, and so on.



As you would expect, clicking on the grey boxes makes sections and subsections collapse away or reappear.

## Greek letters

Greek letters have English spellings. When Maple sees "nu", it prints  $\nu$ , for example. You can View the **Symbol Palette** ( menu: View/Palettes/Symbol Palette) and this will tell you how to enter things like infinity or kappa. Here I typed a := and then clicke on a symbol in the palette:

**a := Pi, Theta, theta;**  
 $a := \pi, \Theta, \theta$



In general you just spell the thing out as you would expect. First letter capital gets you the uppercase Greek letter, but there are exceptions:

Pi =  $\pi$  the number  
 pi =  $\pi$  the Greek letter  
 PI =  $\Pi$ , the capital Greek letter  
 GAMMA(x) = the gamma function  
 gamma = the constant .577 (Euler constant),  
 Gamma =  $\Gamma$ , capital letter  
 alpha, beta, delta, gamma, epsilon, zeta( $\zeta$ ), eta, theta, iota, kappa, lambda, mu  
 nu, xi( $\xi$ ), omicron, pi, rho, sigma, tau, upsilon, phi, chi, psi, omega

If you want to use gamma as an ordinary Greek letter, you have to "unprotect it" :

```

[> gamma;evalf(%);
                                      $\gamma$ 
                                     .5772156649
[> gamma := 4;
Error, attempting to assign to `gamma` which is protected
[> unprotect('gamma');evalf(gamma);
                                     .5772156649
[> gamma := 4;
                                      $\gamma := 4$ 

```

If you want a primed variable like x', enter it as `x'` . Unfortunately `nu` does not appear as v' . nu1 does come out as v1, however.

**Constants : false gamma infinity true Catalan FAIL Pi I**

Recall from the Greek letters discussion that Maple accepts kappa for  $\kappa$  and pi for  $\pi$  and alpha for  $\alpha$ . Therefore, the lower-case pi is just  $\pi$ , a Greek letter (very "inert"). And PI is  $\Pi$ , the capital Greek letter. Neither of these has anything to do with 3.14159. The correct Maple symbol for that constant is Pi. From the help: "E is no longer a reserved name in Maple. It has been replaced with exp(1)." The only reserved constants are: <false gamma infinity true Catalan FAIL Pi>. gamma  $\gamma$  is the Euler–Mascheroni constant which appears lots of places including as  $\gamma = -\Gamma'(1) \approx .57$ . Catalan is a similar constant  $\approx .915$ .

The last item I is in fact not a built-in constant, but is a built-in alias for sqrt(-1).

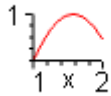
**ditto operators      %    %%    %%%            and    %n**

As each code line is executed, there is a single result and it exists in a "result buffer" called %. So if you fail to name a result, you can refer to it as %, known as the "ditto operator". The previously computed result is saved as %, and the third previously computed one as %%. Here is an example of % use:

```

y := sin(x);
      y = sin(x)
plot(%, x=1..2);

```



So when the plot command is executed, % has the value "sin(x)". If you go off and run some other commands manually and come back, % will have the value of the last command you ran, not what you think.

Complex expressions often group messy sub-expressions into items like %1 and %2, and these can be accessed directly by these names.

A very common use of % is to tack something using it onto the end of a command line":

```

sin(2);evalf(%);
                        sin(2)
                        .9092974268

```

It is easier to do this than add a whole new command just to see what something really is.

## Inert Forms of Operators value

Inert operators **begin with a capital letter** and are often referred to in Maple Help as "**placeholders**". The idea is that you defer the evaluation of an operation until you are ready. You do this so you can make sure you entered your expression correctly (especially complex ones), since it's easier to check it in the typeset notation. Here are some non-complex examples:

<pre> &gt; Diff(sin(x), x); </pre>	$\frac{\partial}{\partial x} \sin(x)$	$\left[ \begin{array}{l} > \text{Sum}(k^2, k=1..3); \\ & \sum_{k=1}^3 k^2 \\ > \text{sum}(k^2, k=1..3); \\ & 14 \end{array} \right.$
<pre> &gt; diff(sin(x), x); </pre>	$\cos(x)$	

You can convert from inert to evaluated forms using the "value" command, as follows:

```

[> value(Diff(sin(x), x));
      cos(x) // same as diff above

```

See the "diff" section below for an example where Diff plays a more essential role.

---

## macro alias

macro: These two commands are similar.

A "macro" of the form `macro(joe = bob)` simply causes the letters joe to be replaced by the letters bob anywhere they appear. Here is an example where we compute  $P_2(0.3)$  twice:

```
LegendreP(2, .3);
-3650000000
macro(joe=LegendreP);
joe;
LegendreP
joe(2, .3);
-3650000000
```

If we ever want to know what "joe" is macro for, we just type `joe`; and it tells us. You can use a macro to make a symbol for any string of letters you get tired of typing,

```
macro(f=x^2+3*y^3);
f;
x2 + 3y3
```

alias: Here is the LegendreP example done with "alias", and things are a bit more complicated:

```
LegendreP(2, .3);
-3650000000
alias(joe=LegendreP);
I,joe
joe;
joe
joe(2, .3);
-3650000000
alias();
I,joe
alias(tom=LegendreQ);
I,joe,tom
tom(1, 0.2);
- .9594534892 + .3141592654 I
LegendreQ(1, 0.2);
- .9594534892 + .3141592654 I
```

First of all, whenever you do an alias, it gives you a list of all existing aliases, and it just happens that the symbol `I` is an alias for  $\sqrt{-1}$  that "comes with" Maple. The way aliases work is basic: in any command the string `joe` is converted to string `LegendreP` and then processing occurs as usual. On output, `LegendreP`

is converted back to joe, so you see joe. There does not seem to be any way to ask Maple what the alias joe stands for after you create it.

You *remove* alias joe by saying `alias(joe=joe)`. Several aliases can be put into a single command as in the example below.

### **How to replace a default symbol name with another symbol and free up its name**

Example: The symbol  $I$  by default is  $\sqrt{-1}$ . Perhaps you want  $I$  to be some electric current, and you want  $j$  to be the symbol for  $\sqrt{-1}$ . You first say `alias(I=I)` which removes the default Maple alias that  $I$  stands for  $\sqrt{-1}$ . After doing this, you can use  $I$  as a normal variable name. Second, say `alias(j=sqrt(-1))` to cause the symbol  $j$  to be  $\sqrt{-1}$ . Of course after doing this  $j$  cannot be used as a variable name. These can be combined into the single command `alias(I=I, j= sqrt(-1))`.

---

## EXPRESSION MANIPULATION AND EVALUATION IN MAPLE

---

### type

In a normal computer language, data objects are explicitly "typed" in some way. You might have something that is a structure that contains three expressions and then 10 3D points, for example. Or you might have something that is a list, or a polynomial, or a matrix. In Maple, there are many such types, but they tend to be hidden from sight, and if you want to know the type of some object, **you have to ask!**

Ultimately, most everything in any Maple structure is going to be a mathematical "expression". A rational number is a special case of an expression, and an integer is a special case of a rational number. More generally expressions are things like  $\sin(x) + 2^y$ .

There are a few exceptions in that some objects can be "strings" of text and a few other things like that.

Normal languages don't make distinctions like "polynomial" or "numeric", because normal languages don't process expressions the way Maple does!

It is important for a Maple user to know the nature of his or her objects. If you want to know if object A is a matrix, you ask Maple as follows:

```
A := 3.2;
                                     A = 3.2
type(A, 'matrix');
                                     false
```

So no, the number 3.2 is not a matrix. Here is a list of types in Maple, and you see matrix in the list:

<a href="#">!</a>	<a href="#">.</a>	<a href="#">And</a>	<a href="#">NONNEGATIVE</a>	<a href="#">Not</a>
<a href="#">Or</a>	<a href="#">PLOT</a>	<a href="#">PLOT3D</a>	<a href="#">Point</a>	<a href="#">Range</a>
<a href="#">RootOf</a>	<a href="#">TEXT</a>	<a href="#">`**`</a>	<a href="#">`*`</a>	<a href="#">`+`</a>
<a href="#">`^`</a>	<a href="#">algebraic</a>	<a href="#">algext</a>	<a href="#">algfun</a>	<a href="#">algnum</a>
<a href="#">algnumext</a>	<a href="#">anyfunc</a>	<a href="#">anything</a>	<a href="#">arctrig</a>	<a href="#">array</a>
<a href="#">atomic</a>	<a href="#">boolean</a>	<a href="#">complex</a>	<a href="#">complexcons</a>	<a href="#">constant</a>
<a href="#">cubic</a>	<a href="#">dependent</a>	<a href="#">disjctc</a>	<a href="#">equation</a>	<a href="#">even</a>
<a href="#">evenfunc</a>	<a href="#">expanded</a>	<a href="#">exprseq</a>	<a href="#">facint</a>	<a href="#">float</a>
<a href="#">fraction</a>	<a href="#">freeof</a>	<a href="#">function</a>	<a href="#">hfarray</a>	<a href="#">identical</a>
<a href="#">indexed</a>	<a href="#">indexedfun</a>	<a href="#">infinity</a>	<a href="#">integer</a>	<a href="#">intersect</a>
<a href="#">laurent</a>	<a href="#">linear</a>	<a href="#">list</a>	<a href="#">listlist</a>	<a href="#">literal</a>
<a href="#">logical</a>	<a href="#">mathfunc</a>	<a href="#">matrix</a>	<a href="#">minus</a>	<a href="#">monomial</a>
<a href="#">name</a>	<a href="#">negative</a>	<a href="#">negint</a>	<a href="#">nonneg</a>	<a href="#">nonnegint</a>
<a href="#">nonposint</a>	<a href="#">nothing</a>	<a href="#">numeric</a>	<a href="#">odd</a>	<a href="#">oddfunc</a>
<a href="#">operator</a>	<a href="#">point</a>	<a href="#">polynom</a>	<a href="#">posint</a>	<a href="#">positive</a>
<a href="#">prime</a>	<a href="#">procedure</a>	<a href="#">protected</a>	<a href="#">quadratic</a>	<a href="#">quartic</a>
<a href="#">radalgfun</a>	<a href="#">radalgnum</a>	<a href="#">radext</a>	<a href="#">radfun</a>	<a href="#">radfunext</a>
<a href="#">radical</a>	<a href="#">radnum</a>	<a href="#">radnumext</a>	<a href="#">range</a>	<a href="#">rational</a>
<a href="#">ratpoly</a>	<a href="#">realcons</a>	<a href="#">relation</a>	<a href="#">rgf seq</a>	<a href="#">scalar</a>
<a href="#">series</a>	<a href="#">set</a>	<a href="#">specfunc</a>	<a href="#">sqrt</a>	<a href="#">square</a>
<a href="#">string</a>	<a href="#">symbol</a>	<a href="#">symmfunc</a>	<a href="#">table</a>	<a href="#">taylor</a>
<a href="#">trig</a>	<a href="#">type</a>	<a href="#">uneval</a>	<a href="#">union</a>	<a href="#">vector</a>

Notice that list and set are in the list, but sequence is not in the list. In the Help you can click on any of these items to learn more about it. Maple allows the user to create new types, and there is much more type technology inside Maple, but that is beyond the scope of this document or my knowledge of Maple.

In the rest of this document, we will often use the type command to learn about an object's type.

## Numbers and Base Conversion

Numbers can be integers like 100, fixed point like 100.3, or floating point like  $1e3 = 1000$ . The default of course is base 10. The number of digits of precision for numbers is 10 by default, but can be set to any number desired

```

evalf(Pi);
                                     3.141592654
Digits := 50;
                                     Digits := 50
evalf(Pi);
                                     3.1415926535897932384626433832795028841971693993751

```

Math is done in software in Maple so a larger Digits value just slows things down. See evalhf in Maple help for how to make use of floating point hardware for increased computation speed.

Maple can deal with the hexadecimal base 16 as shown in these examples:

```
# convert to hex from decimal
a := convert(123456,hex,decimal);
                                     a = 1E240
# convert back to decimal from hex
convert(a,decimal,hex);
                                     123456
# convert to decimal from hex using direct entry
convert(`1E240`,decimal,hex);
                                     123456
```

Notice that 1E240 is used here as a name and since this name does not begin with a letter, it must always be surrounded by tick marks when directly entered. If one wants the digits of a number in a *list* so they can be processed in some manner, one can say

```
convert(123456,base,10);
                                     [6, 5, 4, 3, 2, 1]
convert(123456,base,16);
                                     [0, 4, 2, 14, 1]
.
```

where notice the least significant digit is first, which is reversed from the normal way digits are displayed. Also, notice that E = 14 for hex. This last command form works for any base, not just standard bases. It is possible to take a list of digits in one base and convert it to a list of digits in some other base. In this example, which uses the lists above, we convert from base 16 to base 10. The second command shows that it is always easy to get back from a digit list in any base to decimal number (nops = # elem in list)

```
c := convert([0,4,2,14,1],base,16,10);
                                     c = [6, 5, 4, 3, 2, 1]
n := sum(c[i+1]*10^i,i=0..nops(c)-1);
                                     n = 123456
.
```

For octal and binary, the tick marks must not be used in the direct entry. We repeat the above example for octal (binary works exactly the same way)

```
# convert to octal from decimal
a := convert(123456,octal,decimal);
                                     a = 361100
# convert back to decimal from octal
convert(a,decimal,octal);
                                     123456
# convert to decimal from octal using direct entry
convert(361100,decimal,octal);
                                     123456
convert(123456,base,8);
                                     [0, 0, 1, 1, 6, 3]
```



Warning: One can only convert if one of the bases is decimal. For example, in the first line below the octal keyword is ignored and the number entered is treated as decimal,

```

convert(361100,hex,octal); # does not do what you expect
                               5828C
convert(361100,hex,decimal);
                               5828C

```

To convert then from octal to hex one must do it in two steps, for example

```

convert(361100,decimal,octal):convert(%,hex,decimal);
                               1E240

```

## operators

First, here is the list

!	\$	%%%	%%	%
**	*	+	-	.
/	<	<=	<>	=
>	>=	@	@@	^
and	arithmetic	boolean	constant	ditto
factorial	fraction	integer	intersect	minus
mod	name	neutral	not	nullstring
operators	or	ratpoly	selection	set
uneval	union			

The usual arithmetic operators are +, -, \*, / ; exponentiation is either ^ or \*\*.

Boolean test operators are =, <, >, <=, >= and <> for not equal.

Boolean operators are and, or and not. There is no xor or xnor.

Operators %, %% and %%% are called ditto operators and refer to the last, second last, and third last item computed.

Operator \$ forms a sequence as in these examples (the first two are the same)

```

n^2 $ n = 1..3;
                               1, 4, 9
seq(n^2, n=1..3);
                               1, 4, 9
n^2 $3;
                               n^2, n^2, n^2

```

Operator ! is the usual factorial operator so n! = factorial(n).

Operator precedence is fairly standard with ^ the highest, then \* and /, and finally + and -. But even higher than these is function argument binding. Some examples are shown here on the left:

<code>a+b*c^2;</code>		<code>a/b/c/d;</code>	
<code>a + (b*c)^2;</code>	$a + b c^2$	<code>a*b/c/d;</code>	$\frac{a}{b c d}$
<code>sin(x)^2;</code>	$a + b^2 c^2$	<code>a/b*c/d;</code>	$\frac{a b}{c d}$
<code>sin(x^2);</code>	$\sin(x)^2$	<code>a/b*c/d;</code>	$\frac{a c}{b d}$
	$\sin(x^2)$		

The usages on the right are legal but seem a bit dangerous.

### **trunc round frac floor ceil mod**

From the Help system (comments added)

For  $x \geq 0$ , **trunc**(x) is the greatest integer less than or equal to x. For  $x < 0$ ,  $\text{trunc}(x) = -\text{trunc}(-x)$ . In other words, **trunc**(x) truncates toward 0, unlike **floor** which truncates down.

**round**(x) rounds x to the nearest integer.  $\text{Round}(0.5) = 1$  and  $\text{round}(-0.5) = -1$ .

**frac**(x) is the fractional part of x, that is,  $\text{frac}(x) = x - \text{trunc}(x)$ .

For  $x \geq 0$ , some computer languages use **Rem**(x) for this function, but not Maple.

**floor**(x) is the greatest integer less than or equal to x.

For  $x \geq 0$ , some computer languages use **Int**(x) for this function, but not Maple.

**ceil**(x) is the smallest integer greater than or equal to x.

**x mod m** or **modp(x,m)** is the usual modulo function, m normally a positive integer.

For  $x \geq 0$ , some computer languages use **x%m** for this function, but not Maple.

For Maple, % is the ditto operator, see above.

### **sequences lists sets seq repeat operator \$**

There are three *distinct data structure items* discussed here: sequence, set, list. In addition, we discuss a certain operator "seq" which lets us easily build any of these data structures.

A "**sequence**" is a comma-separated list of expressions, and concatenation works as you would expect. Here is an example showing the creating of a sequence q, and then a concatenation of the sequence with more items. We show how you access an element of a sequence using square brackets; the first item in a sequence is item number 1.

```

> q := k,m+1,8,7;
                                q = k, m + 1, 8, 7
> q[2];
                                m + 1
> p := q,9,potato;
                                p = k, m + 1, 8, 7, 9, potato
> r := [q,9];
                                r = [k, m + 1, 8, 7, 9]
> r[2];
                                m + 1
> type(r, 'sequence'); type(r, 'list'); type(r, 'set');
Error, type `sequence` does not exist
                                true
                                false

```

We took the liberty of jumping ahead here and made a "list" which is object r, see below. Maple verifies that r is a list and it is not a set. It cannot say whether it is a sequence, because, although sequences exist in Maple, they are not a "type", as was noted in the last section. If we want to select a portion of a sequence we can do it this pretty obvious way:

```

q[2..3];
                                m + 1, 8

```

You can construct a sequence using the **seq operator**, lots of examples are provided in the help:

```

seq( i^2, i=1..5 );
                                1, 4, 9, 16, 25
seq( sin(Pi*i/6), i=0..6 );
                                0,  $\frac{1}{2}$ ,  $\frac{1}{2}\sqrt{3}$ , 1,  $\frac{1}{2}\sqrt{3}$ ,  $\frac{1}{2}$ , 0
seq( x[i], i=1..5 );
                                 $x_1, x_2, x_3, x_4, x_5$ 
X := [seq( i, i=0..6 )];
                                X = [0, 1, 2, 3, 4, 5, 6]
{seq( i^2 mod 7, i=X )};
                                 $(x+y)^2, (x+y)^3$ 
seq((x+y)^a, a=2..3);

```

If you want a sequence of identical items, here is the easy way using the **\$ operator** (see diff later)

```

k$5;
                                k, k, k, k, k
[joe, bob]$3;
                                [joe, bob], [joe, bob], [joe, bob]

```

A **"set"** is a sequence surrounded by {...}, and a **"list"** is a sequence surrounded by [...]. Notice above that you can use seq to make a set or a list in the obvious manner. Here are more examples,

```

seq(i^2, i=[1, 2, 3, 4, 5]);
                                1, 4, 9, 16, 25
[seq(i^2, i=1..5)];
                                [1, 4, 9, 16, 25]

```

This shows how to use seq to make a "list" instead of a "sequence". Note that i=1..5 is in fact a "list" as shown in the first item above.

A **"list" has order to it, a "set" has unordered elements or "members"**. Both lists and sets have "elements", these are the items. Order versus no order makes a big difference when you do something like plot a set of points and connect the points with lines to get a smoother curve!

Now here are examples of a sequence, set and list where the members are 2D vectors (points):

```

p := seq([i, i^2], i=1..4);
                                p = [1, 1], [2, 4], [3, 9], [4, 16]
q := {seq([i, i^2], i=1..4)};
                                q = {[2, 4], [1, 1], [3, 9], [4, 16]}
r := [seq([i, i^2], i=1..4)];
                                r = [[1, 1], [2, 4], [3, 9], [4, 16]]

```

In passing, it should be noted that either of the last two objects (the set and the list) can be used to make a **scatter plot** of the 2D data using the command pointplot(q) or pointplot(r), see plotting later.

The number of elements in a set or list can be determined using the nops command described elsewhere in this document. For example, both the set {q} and the list [q] have length 4 :

```

q := k, m+1, 8, 7;
                                q = k, m+1, 8, 7
nops({q}); nops([q]);
                                4
                                4

```

It is not necessary that all the items in a sequence, set or list be the same type, although the examples above might give that impression. Here are some examples which use concatenation with the q,r above

```

s := q, 10;
                                s = {[2, 4], [1, 1], [3, 9], [4, 16]}, 10
t := r, {5, 6, 7};
                                t = [[1, 1], [2, 4], [3, 9], [4, 16]], {5, 6, 7}

```

In the first sequence, there are two items, the first being a set and the second a number. In the second sequence, there are two items, the first is a list of 2D points, the second is a set of three numbers. Obviously there is much flexibility in Maple, but certain operations are going require certain data structures to work properly. For example, you cannot call `pointplot(p)` where `p` is a sequence.

Finally, here is an example of a *nested* structure: At the highest level, we have `v` which is a list of 2 items. Each item is itself a list. These secondary lists each have three items in them. The third item is a set of two lists! Each of those lists has 3 numeric elements. Notice the interesting access `v[i][j]` which can be generalized to any level of nesting -- multiple square brackets.

```
v := [eigenvectors(A)];
      v := [[-2, 2, {[1, 1, 0], [-1, 0, 1]}], [4, 1, {[1, 1, 2]}]]
v[1][1]; # The first eigenvalue
      -2
v[1][2]; # It's multiplicity
      2
v[1][3]; # It's eigenvectors
      {[1, 1, 0], [-1, 0, 1]}
```

This nested list is pretty handy for showing the eigenvector information of a matrix, you can put all the interesting information in the same list, and you can access anything you want.

Hopefully the reader is convinced that sequences, lists and sets provide at least all the flexibility that one might have using C structures in the C language, or C classes in the C++ language (since expressions can be functions as well as data).

## Conversion between Sequences, Sets, Lists, and Arrays

### 1. Conversion from Set to Sequence and back to Set

```
seta := {1, 2, 3, 4};
      seta := {1, 2, 3, 4}
seqa := op(seta);
      seqa := 1, 2, 3, 4
setaa := {seqa};
      setaa := {1, 2, 3, 4}
```

## 2. Conversion from List to Sequence and back to List

```
lista := [a,b,c,d];  
      lista := [a, b, c, d]  
seqa := op(lista);  
      seqa := a, b, c, d  
listaa := [seqa];  
      listaa := [a, b, c, d]
```

## 3. Conversion from List to Set and back to List. Note how this filters out duplicate items.

```
lista := [x,y,z,t];  
      lista := [x, y, z, t]  
seta := convert(lista, set);  
      seta := {x, y, t, z}  
listaa := convert(seta, list);  
      listaa := [x, y, t, z]
```

```
lista := [x,y,y,t];  
      lista := [x, y, y, t]  
seta := convert(lista, set);  
      seta := {x, y, t}  
listaa := convert(seta, list);  
      listaa := [x, y, t]
```

## 4. Convert from Array to List and back to Array.

```
arraya := array([1,2,3,4]);  
      arraya := [1, 2, 3, 4]  
type(arraya, list); type(arraya, array);  
      false  
      true  
lista := convert(arraya, list);  
      lista := [1, 2, 3, 4]  
type(lista, list); type(lista, array);  
      true  
      false  
arrayaa := array(lista);  
      arrayaa := [1, 2, 3, 4]  
type(arrayaa, list); type(arrayaa, array);  
      false  
      true
```

---

## list // revisited: doing math with elements of a list

Consider this sample code where we create a list of 5 lists:

```
> r := [seq([i,i,i],i=1..5)];
      r := [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4], [5, 5, 5]]
> r[3];
      [3, 3, 3]
> type(r[3], 'list'); type(r[3], 'vector');
      true
      false
> r[6];
Error, invalid subscript selector
```

Here we create a list of five 3D "vectors", filling them with some dummy data. We display one of the vectors. We cannot access  $r[6]$  because it does not exist. We ask if  $r[3]$  is really a vector, and Maple says no. The word 'vector' refers to something we shall see later which is a "Maple vector". Our object  $r[3]$  does not pass muster to be a Maple vector, it is only a "list", but it can do certain things that a real Maple vector can do:

```
> r[3] := [1,2,3];
      r3 := [1, 2, 3]
> r[4] := [5,6,7];
      r4 := [5, 6, 7]
> r[5] := r[3]+r[4];
      r5 := [6, 8, 10]
> r[5] := 10*r[3];
      r5 := [10, 20, 30]
> r[5] := r[3]*r[4];
      r5 := [1, 2, 3][5, 6, 7]
> r[5] := evalm(r[3]&*r[4]);
      r5 := 38
> r[4] := sin(r[3]);
Error, sin expects its 1st argument, x, to be of type algebraic, but received [1, 2, 3]
> for i from 1 to 3 do r[4,i] := sin(r[3,i]) od;
      r4,1 := sin(1)
      r4,2 := sin(2)
      r4,3 := sin(3)
```

We then set new data into two vectors of the list, and proceed to add these two vectors, and multiply a vector by a scalar, both successful actions. We hopefully try a pairwise multiplication, but this produces a mysterious object that does not seem useful. In the next line we try computing a dot product pretending

that we do have Maple vectors, using a method shown below, and it works! Our next attempted action with the sine is rejected, so must be done "in longhand" as the next line shows.

The point here is that certain basic mathematical operations can be performed on list elements which are compatible with each other. Basically, we can add, subtract, and multiply by scalars.

We could if we wanted repeat all the above with true Maple vectors. Here we comment out the first line with a # to allow easy comparison,

```
> #r := [seq([i, i, i], i=1..5)];
> r := [seq(vector([i, i, i]), i=1..5)];
           r := [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4], [5, 5, 5]]
> r[3];
           [3, 3, 3]
> type(r[3], 'list'); type(r[3], 'vector');
           false
           true
```

So now r[3] is a vector and not a list. You cannot tell this fact by staring at the object, you have to use the type command because a list (of this kind) and a vector look the same. The vector math shows slight variations, for example

```
r[1]+r[2];
           [1, 1, 1]+[2, 2, 2]
evalm(%);
           [3, 3, 3]
```

When the objects are real Maple vectors, we have to use evalm to get the addition we want.

### **eval Eval evalf evalhf**

Examples:

```
eval(x^2 + 2, x = a)           // evaluate the expression e "at the point x=a" (one variable)
eval(x^2 + 2*y, [x = a, y=b]) // if more than one variable, must use the "list notation"
eval(e)                        // evaluate expression e through all levels of recursion
eval(e,2)                      // evaluate only going back 2 levels of recursion
```

The last two items are useful when you have a chain of equations a:=b where b:=c and so on. As for the "evaluation at a point", the substitution is made as specified, then the thing is computed as far as it can be computed. You cannot combine the x=a and n arguments together.

Example:



```
> f := 5 + x^2;
      2
      f := 5 + x
> eval(f,x=2);
      9
```

As usual, the inert Eval attempts to defer the evaluation, not meaningful in all cases. Here is an example:

```
> Eval(int(sin(x),x), x=y);
      (-cos(x)) |
      |x = y
> value(%);
      -cos(y) // see "value" above
```

Notice that the **subs** command puts in the thing you want, but does not then evaluate the result. (below)

When you say **evalf**(something), you are asking for an actual floating point number. And if you are going for speed, you might use **evalhf**(something) which makes use of the computer's hardware floating point capability, but it does not work with Bessel functions! [ Notes TBA on evalhf which does not naturally exist in Maple. ]

Here is a favorite example where second argument indicates **number of decimal places** required:

```
evalf(Pi,30);
      3.14159265358979323846264338328
```

Of course evalf is doing software floating point, so like the Krell power source, it will do however many places asked of it, such as 500,000. There are memory limitations.

The **evalf** is usually called internally when you do **eval** on a function at some random argument location. But for simple arguments, this does not happen and you have to do it yourself:

```
eval(sin(x),x=.9);
      .7833269096
eval(sin(x), x=1);
      sin(1)
evalf(%);
      .8414709848
```

If something is running way too slow, you might be able to speed it up using **evalhf**() which uses the computer's floating point hardware (double-precision) for floating point calculations. However, some special function calls like BesselJ don't use this so you might not win. Since Digits = 10 by default, software computations produce 10 digits while the double-precision hardware does 64 bits which means about 16 decimal places. For example,

```
evalf(sqrt(5));
      2.236067978
evalhf(sqrt(5));
      2.236067977499790
```

It is claimed by some that numerical integration always uses evalhf if Digits is set less than 16 or so, at least for parts of the integration code that can use the hardware effectively.

---

## subs alsubs

Methods of making a **substitution** in an expression. The first makes a substitution without doing an eval, the sub just sits in there where you put it.

```
subs( sin(x)=y, sin(x)/sqrt(1-sin(x)) );
```

$$\frac{y}{\sqrt{1-y}}$$

"The function **alsubs** performs an algebraic substitution, replacing occurrences of a with b in the expression f. It is a generalization of the subs command, which only handles syntactic substitution."

You might wonder how subs and eval differ. Here is an example showing that eval "evaluates" all it can

```
expr := sin(x)/cos(x);
```

$$expr := \frac{\sin(x)}{\cos(x)}$$

```
subs(x=0, expr);
```

$$\frac{\sin(0)}{\cos(0)}$$

```
eval(expr, x=0);
```

0

Notice also the different argument order for subs and eval.

Another example fills a gap in Maple's convert capability (cannot convert ln to inverse trig) for  $Q_1(i\zeta)$ :

```
h;
```

$$-1 + \frac{1}{2} I \zeta \ln\left(\frac{-I + \zeta}{\zeta + I}\right)$$

```
i := subs(ln((zeta-I)/(zeta+I))=-2*I*arccot(zeta), h);
```

$$i := -1 + \zeta \operatorname{arccot}(\zeta)$$


---

## lhs rhs numer denom

These operators just give quick access to the "left hand side" or "right hand side" of an equation, or to the "numerator" or "denominator" of a fraction. These same operations can always be done with the more general op command discussed below. Here are some simple examples:

```
eq1 := joe = sin(x);
      eq1 := joe = sin(x)
lhs(eq1);
      joe
rhs(eq1);
      sin(x)
myfrac := joe/sin(x);
      myfrac :=  $\frac{joe}{sin(x)}$ 
numer(myfrac);
      joe
denom(myfrac);
       $\frac{1}{sin(x)}$ 
op(1,eq1);
      joe
op(2,eq1);
      sin(x)
op(1,myfrac);
      joe
op(2,myfrac);
       $\frac{1}{sin(x)}$ 
```

## unprotect type reserved words and letters in Maple

Suppose you want to use D as a variable name. If you try to do so, you find that it is a protected symbol, used for differentiation. But you can remove this use and then use it as you like. For example

```
> D := 1;
Error, attempting to assign to `D` which is protected
> unprotect(D);
> D := 1;

> type(A, protected); // This is how you check to see if something is predefined, like Pi is for example
false
```

Maple has a list of 30 **reserved words**

```
and by do done elif
else end fi for from
if in intersect local minus
mod not od option options
or proc quit read save
stop then to union while
```

but there are also some **reserved** (or at least special) **single letters**, a little harder to identify -- you have to go through the Help system one letter at a time in the **topic search**. Here are the basic ones

C a function used to generate code (on the list in Appendix below)  
D differential operator:  $D(\sin) = \cos$        $D(\sin)(x) = \cos(x)$   
I imaginary  $i$   
O used to mean "order of" in some output displays  
c used for commutators in the commutat package

Although things like J appear in reference to Bessel function, the Maple name is BesselJ(x).

Here are some other reserved names referring to constants

GAMMA	$\Gamma$	not just a Greek letter, it refers to the "gamma function" $\Gamma(x)$
gamma	$\gamma$	again, is used for Euler's constant .57
Catalan		another constant
Pi	$\pi$	another constant

And of course there are *lots* of reserved names which are official function names, including

GAMMA	$\Gamma$	not just a Greek letter, it refers to the "gamma function" $\Gamma(x)$
-------	----------	--

These function names are all shown in the Appendix below. The GAMMA one is special because normally this would just indicate an upper case Greek letter. This list includes Chi and Psi functions which are not official Greek letters in Maple since mixed case.

## unassign

This just removes an assigned meaning to a symbol, for example

```
E := 3;
E;
3
unassign('E');
E;
E
```

## assume      is      about      additionally

Some examples: `assume(a >= 0)`, `assume(theta >= 0, theta < Pi)`, `assume(a, real)` .

For a given variable, you must put everything into a **single assume statement**, otherwise the last assume statement you execute wipes out the results of earlier ones, though this problem can be fixed using the **additionally** command. One something is assumed, you can inquire about it using **is** and **about**.

```

[> assume(a>=0); # this also implies that a is real
[> is(a>=0);
                                     true
[> about(a);
Originally a, renamed a~:
  is assumed to be: RealRange(0,infinity)
[> assume(a,real); # this wipes out the previous assume
[> is(a,real);
                                     true
[> is(a>=0);
                                     false
[> additionally(a>=0); # in addition to a real, want a >= 0
[> is(a>=0);
                                     true
[> is(a,real);
                                     true
[> about(a);
Originally a, renamed a~:
  is assumed to be: RealRange(0,infinity)

```

Once you assume something about a variable, that **variable will appear with a tilde suffix** in all subsequent stuff as a reminder. You can turn this off with the Options (recommended).

How do you unassume? There is no "unassume" command. You do this by saying `a := 'a'`; which seems strange. Here from the Help: "Assumptions made on names may be erased (cleared) by unassigning names. For example, having assumed that x is positive, `x := 'x'`; clears this assumption made on x." (See unassign elsewhere.)

Why would you want to assume something about a variable? One reason is to allow Maple to clear out square roots and other fractional powers.

### Examples:

<pre> bob := d^2*(f/d^2)^(1/2); simplify(bob); assume(d&gt;0); simplify(bob); </pre>	<pre> bob := d^2*(f/d^3)^(1/3); simplify(bob); assume(d&gt;0); simplify(bob); </pre>
$bob = d^2 \sqrt{\frac{f}{d^2}}$ $d^2 \sqrt{\frac{f}{d^2}}$ $d \sqrt{f}$	$bob = d^2 \left(\frac{f}{d^3}\right)^{\left(\frac{1}{3}\right)}$ $d^2 \left(\frac{f}{d^3}\right)^{\left(\frac{1}{3}\right)}$ $d \sqrt[3]{f}$

Maple frequently seems incredibly stupid about canceling things in numerator and denominator as in these examples. Its reason is that it doesn't know which root you want:

$$y = \sqrt{x^2} \Rightarrow y = +x, -x \qquad y = (x^3)^{1/3} \Rightarrow y = x, e^{i\pi/3}x, e^{2i\pi/3}x$$

You always assume the first root, but Maple has to be more careful to not mislead you.

A second typical use of assume is to make sure integrals converge:

Example:

```
[> restart;
> f := Int(exp(-a*x), x=0..infinity);
```

$$f := \int_0^{\infty} e^{(-a x)} dx$$

```
> value(f); # this converts inert Int to int to do integral
Definite integration: Can't determine if the integral is convergent.
Need to know the sign of --> a
Will now try indefinite integration and then take limits.
```

$$\lim_{x \rightarrow \infty} -\frac{e^{(-a x)} - 1}{a}$$

```
> assume(a>0);
> value(f); # now Maple can do the integral to get 1/a
```

$$\frac{1}{a}$$

Maple does not "deduce" all the information a person would given a set of assumed facts, so it is usually necessary to put in everything but the kitchen sink if you want things to work right. For example, if you have a situation where  $0 < v < h < \mu < k < \rho$  (which arises in elliptical coordinates), you need to say

```
assume(h>0, k>0, k>h, rho>0, rho>k, rho>h, rho>mu, rho>nu, mu>0, mu>h, mu<k, mu>nu, nu>0, nu<h);
```

Then when things like  $\sqrt{(\rho-k)^2}$  appear, Maple will know this is  $\rho-k$  and not  $k-\rho$ , for example.

As noted earlier, you can learn what Maple is assuming about a variable with the **about** command. For example, after the above assume we get

```
> about(rho);
Originally rho, renamed rho~:
Involved in the following expressions with properties
-rho+k assumed RealRange(-infinity, Open(0))
-rho+h assumed RealRange(-infinity, Open(0))
-rho+mu assumed RealRange(-infinity, Open(0))
-rho+nu assumed RealRange(-infinity, Open(0))
is assumed to be: RealRange(Open(0), infinity)
also used in the following assumed objects
[-rho+h] assumed RealRange(-infinity, Open(0))
[-rho+k] assumed RealRange(-infinity, Open(0))
[-rho+nu] assumed RealRange(-infinity, Open(0))
[-rho+mu] assumed RealRange(-infinity, Open(0))
```

The "assume facility" as Maple likes to call it has a few unexpected side effects. On the left below perhaps you assumed B was real for some reason, then later on you assign  $A := B$  and after that you assign  $B := 2$ . You might logically expect A to be 2 as this point, but it is not! You try to fix the problem by "assigning the assumed variable name", but Maple thinks little of that idea: The way to deal with this problem is shown on the right

```

> restart;
> assume('B',real);
> A := B;
                                     A := B~
> B := 2;
                                     B := 2
> B;
                                     2
> A;
                                     B~
> value(A);
                                     B~
> evalf(A);
                                     B~
> B~ := 2;
missing operator or `;`

```

```

restart;
assume('B',real);
A := B;
                                     A := B~
A := eval(A,B=2);
                                     A := 2
A;
                                     2

```

I think the real problem illustrated in the above example is that "you cannot assume on a constant", even if the assumption is true for that constant. Consider first this code:

```

[> restart;
> B := 2;
                                     B := 2
> assume(B,real);
Error, (in assume) cannot assume on a constant object

```

where the error message indicates the problem. But if we do the assume first and *then* do the assignment, as in the previous example, there is no error message and the user is a bit mystified. Maple seems to just quietly reject the assignment of B to a constant in already-defined expressions which include B.

Here is a real-world example where the eval command makes several replacements at once. The assumes are done in this case to allow Maple to generate a relatively simple expression for the integral :

```
> assume(A,real,B,real,C,real, A+B > 0);
> Int(ln(A + B*cos(theta)+ C*sin(theta)), theta = 0..2*Pi);
```

$$\int_0^{2\pi} \ln(A + B \cos(\theta) + C \sin(\theta)) d\theta$$

```
> f := value(%);
> Ref := Re(evalc(f));
```

$$\text{Ref} := 2 \frac{\pi (2\pi B C - B^2 \ln(A+B) + C^2 \ln(A+B)) (-B^2 + C^2)}{(-B^2 + C^2)^2 + 4 C^2 B^2} + 4 \frac{\pi (-\pi C^2 + \pi B^2 + 2 B C \ln(A+B)) C B}{(-B^2 + C^2)^2 + 4 C^2 B^2}$$

```
>
> eval(Ref, [A=r1^2 + x^2 + y^2, B=-2*x*r1, C = -2*y*r1]);
```

$$2 \frac{\pi (8\pi x r1^2 y - 4x^2 r1^2 \ln(r1^2 + x^2 + y^2 - 2x r1) + 4y^2 r1^2 \ln(r1^2 + x^2 + y^2 - 2x r1)) (-4x^2 r1^2 + 4y^2 r1^2)}{(-4x^2 r1^2 + 4y^2 r1^2)^2 + 64y^2 r1^4 x^2} + 16 \frac{\pi (-4\pi y^2 r1^2 + 4\pi x^2 r1^2 + 8x r1^2 y \ln(r1^2 + x^2 + y^2 - 2x r1)) y r1^2 x}{(-4x^2 r1^2 + 4y^2 r1^2)^2 + 64y^2 r1^4 x^2}$$

### **collect, coeff, normal, simplify, expand, combine, factor, rationalize, convert**

These all have their special quirks. I have found that in the end you cannot really make Maple do exactly what you want in displaying the final results, but you can try these things. Each has its own Help.

**collect** groups terms by some item you select, a typical case would be

```
f := x^3 + (x-a)*(x+b);
```

$$f := x^3 + (x - a)(x + b)$$

```
collect(f, x);
```

$$x^3 + x^2 + (-a + b)x - ab$$

The second argument of **collect** can be a function as in the following example. Recall that % means "the last thing computed". Note that it helps to "expand" before "collecting" to get the simplest result :



```

f := (sin(x) + 3*x^2 + x)^3;
                                3
                                f := (sin(x) + 3 x^2 + x)
collect(%, sin(x));
                                sin(x)^3 + (9 x^2 + 3 x) sin(x)^2 + ((3 x^2 + x) (6 x^2 + 2 x) + (3 x^2 + x)^2) sin(x) + (3 x^2 + x)^3
expand(f);
                                sin(x)^3 + 9 sin(x)^2 x^2 + 3 sin(x)^2 x + 27 sin(x) x^4 + 18 sin(x) x^3 + 3 sin(x) x^2 + 27 x^6 + 27 x^5 + 9 x^4 + x^3
collect(%, sin(x));
                                sin(x)^3 + (9 x^2 + 3 x) sin(x)^2 + (18 x^3 + 3 x^2 + 27 x^4) sin(x) + 9 x^4 + x^3 + 27 x^6 + 27 x^5
joe := coeff(%, sin(x)^2);
                                joe := 9 x^2 + 3 x

```

The last command `coeff` shows how to "pick off" a desired coefficient.

One can "collect" in terms of multiple variables in different ways. In the example below, `f` is a polynomial of degree 2 in `x` and `y`. You might want to just see this polynomial "as is" with like terms grouped (distributed option), or you might instead like to see it collected first on `x` and secondarily on `y`. Both ways are possible:

```

> f := (a + b*x + c*y + d*x*y)*(A + B*x + C*y + E*x*y);
                                f := (a + b x + c y + d x y) (A + B x + C y + E x y)
> collect(f, [x, y], distributed);
                                d E x^2 y^2 + (b E + d B) x^2 y + b B x^2 + (c E + d C) x y^2 + (a E + c B + b C + d A) y x + (a B + b A) x + c C y^2
                                + (a C + c A) y + a A
> collect(f, [x, y]);
                                (d E y^2 + (b E + d B) y + b B) x^2 + ((c E + d C) y^2 + (a E + c B + b C + d A) y + a B + b A) x + c C y^2
                                + (a C + c A) y + a A

```

`normal` puts things where possible into a ratio of factored expressions, for example.

```

g := 1 + (x^2 - y^2) / (x - y)^3;
                                g := 1 + (x^2 - 4) / (x - 2)^3
normal(g);
                                x^2 - 3 x + 6
                                (x - 2)^2

```

**simplify** uses things like trig rules to attempt to simplify an expression, and also combines terms and does obvious algebra things a human would do. Here is an example from real code which first displays a matrix in a messy form (as four row vectors), and then simplify gets things under control.

```
> WZ := evalm(U&*WP&*U);
WZ =
[0, 0, 0, 0]
[-W1(1 - 1/2*B) - 1/2*W2(1 - B) + W1(1 + 1/2*B) + 1/2*W2(1 + B),
-W1(1 - 1/2*B) - 1/2*W2(1 - B) - W0 - W1(1 + 1/2*B) - 1/2*W2(1 + B), -1/2*W2(1 - B) + W0 - 1/2*W2(1 + B),
-1/2*W2(1 - B) + 1/2*W2(1 + B)]
[-W1(1 - 1/2*B) - 1/2*W2(1 - B) + W1(1 + 1/2*B) + 1/2*W2(1 + B), -1/2*W2(1 - B) + W0 - 1/2*W2(1 + B),
-W1(1 - 1/2*B) - 1/2*W2(1 - B) - W0 - W1(1 + 1/2*B) - 1/2*W2(1 + B), -1/2*W2(1 - B) + 1/2*W2(1 + B)]
[0, -W1(1 - 1/2*B) + W1(1 + 1/2*B), -W1(1 - 1/2*B) + W1(1 + 1/2*B), -2*W1(1 + 1/2*B) - 2*W1(1 - 1/2*B)]
> simplify(%);
      [ 0          0          0          0
      W1 B + W2 B  -2 W1 - W2 - W0  -W2 + W0  W2 B
      W1 B + W2 B   -W2 + W0   -2 W1 - W2 - W0  W2 B
      0           W1 B         W1 B         -4 W1]
```

Sometimes simplify will "unsimplify" something you have done earlier, like a collect operation.

See section below "Frustration with Simplification".

**expand and combine** work in opposite directions, but results are not always what you expect:

```
> y := (a+b)^4;
      y := (a+b)^4
> expand(%);
      a^4 + 4 a^3 b + 6 a^2 b^2 + 4 a b^3 + b^4
> combine(%);
      a^4 + 4 a^3 b + 6 a^2 b^2 + 4 a b^3 + b^4
> factor(%);
      (a+b)^4
```

```
· h := cos(a+b);
      h := cos(a+b)
· k := expand(h);
      k := cos(a) cos(b) - sin(a) sin(b)
· combine(k);
      cos(a+b)
```

Combine has many options encouraging the use of knowledge of some set of rules, see Help. Here is a simple example:

```
f := sin(x)^5;
```

$$f := \sin(x)^5$$

```
combine(f, trig);
```

$$\frac{1}{16} \sin(5x) - \frac{5}{16} \sin(3x) + \frac{5}{8} \sin(x)$$

**factor** seems pretty smart in general.

**rationalize** gets rid of square roots in denominators, a very simple example being:

```
> 2/(2-sqrt(2));  
  
2  
-----  
2-√2  
  
> rationalize(%);  
  
2+√2
```

**convert** has many operating modes depending on its second argument, which can take all these values:

<a href="#">Airy</a>	<a href="#">Bessel</a>	<a href="#">D</a>	<a href="#">Ei</a>	<a href="#">GAMMA</a>
<a href="#">Heaviside</a>	<a href="#">ODEs</a>	<a href="#">PLOT3Doptions</a>	<a href="#">PLOToptions</a>	<a href="#">POLYGONS</a>
<a href="#">RootOf</a>	<a href="#">StandardFunctions</a>	<a href="#">and</a>	<a href="#">array</a>	<a href="#">base</a>
<a href="#">binary</a>	<a href="#">binomial</a>	<a href="#">bytes</a>	<a href="#">confrac</a>	<a href="#">convert/'*'</a>
<a href="#">convert/'+'</a>	<a href="#">decimal</a>	<a href="#">degrees</a>	<a href="#">diff</a>	<a href="#">disjctc</a>
<a href="#">double</a>	<a href="#">equality</a>	<a href="#">erf</a>	<a href="#">erfc</a>	<a href="#">exp</a>
<a href="#">expln</a>	<a href="#">expsincos</a>	<a href="#">factorial</a>	<a href="#">float</a>	<a href="#">fullparfrac</a>
<a href="#">hex</a>	<a href="#">horner</a>	<a href="#">hypergeom</a>	<a href="#">int</a>	<a href="#">list</a>
<a href="#">listlist</a>	<a href="#">ln</a>	<a href="#">mathorner</a>	<a href="#">matrix</a>	<a href="#">metric</a>
<a href="#">mod2</a>	<a href="#">multiset</a>	<a href="#">name</a>	<a href="#">numericproc</a>	<a href="#">octal</a>
<a href="#">or</a>	<a href="#">parfrac</a>	<a href="#">permlist</a>	<a href="#">piecewise</a>	<a href="#">polar</a>
<a href="#">polynom</a>	<a href="#">pwlist</a>	<a href="#">radians</a>	<a href="#">radical</a>	<a href="#">rational</a>
<a href="#">ratpoly</a>	<a href="#">set</a>	<a href="#">signum</a>	<a href="#">sincos</a>	<a href="#">sqrfree</a>
<a href="#">std</a>	<a href="#">stdle</a>	<a href="#">string</a>	<a href="#">symbol</a>	<a href="#">table</a>
<a href="#">tan</a>	<a href="#">trig</a>	<a href="#">vector</a>		

Note that trig includes trig and hyperbolic functions and you cannot force one or the other. Newer Maple adds `trigh` to the above list to force conversion into hyperbolics (although Maple V.5 does have the type `trigh`). Often the command `combine(f, trig)` will do the trick (combine expands things like  $\sin(a+b)$ ).

Example: This shows the use of the combine command just mentioned.

```
GAMMA(1/2+I*tau)*GAMMA(1/2-I*tau);
```

$$\Gamma\left(\frac{1}{2}+I\tau\right)\Gamma\left(\frac{1}{2}-I\tau\right)$$

```
f:=simplify(%);
```

$$f := -\frac{\pi}{\sin\left(\frac{1}{2}\pi(-1+2I\tau)\right)}$$

```
convert(f, trig);
```

$$-\frac{\pi}{\sin\left(\frac{1}{2}\pi(-1+2I\tau)\right)}$$

```
combine(f, trig);
```

$$\frac{\pi}{\cosh(\pi\tau)}$$

Example: Here we use Bateman v1, p 132(37) to evaluate the  $Q_1(z)$  function using "standard functions":

```
f := (1/3)*(z-1)^(-2)*hypergeom([2, 2], [4], 2/(1-z));
```

$$f := \frac{1}{3} \frac{\text{hypergeom}\left([2, 2], [4], 2\frac{1}{1-z}\right)}{(-1+z)^2}$$

```
g := convert(f, StandardFunctions): simplify(%);
```

$$\frac{1}{2}z \ln\left(\frac{1+z}{-1+z}\right) - 1$$

Example: partial fractions

```
f := (x^3+x)/(x^2-1);
```

$$f := \frac{x^3+x}{x^2-1}$$

```
convert(f, parfrac, x);
```

$$x + \frac{1}{x-1} + \frac{1}{x+1}$$

**operands**      **nops(s)**      **op(3,s)**

An operand is one of the items acted upon by operators *at top level* in a Maple expression. If top level is a function, then the arguments of the function are operands. Function nops(s) tells you how many operands there are in an expression (at top level), and op lets you pick out one of these operands.

<code>s := f(a,b,c);</code>	<code>s := a*b*c;</code>	<code>s := a*b*c+4;</code>	<code>s := a*(b+1)*c;</code>
$s := f(a, b, c)$	$s := a b c$	$s := a b c + 4$	$s := a (b + 1) c$
<code>nops(s);</code>	<code>nops(s);</code>	<code>nops(s);</code>	<code>nops(s);</code>
3	3	2	3
			<code>op(2,s);</code>
			b+1

If you want to get a list of all the operands Maple thinks there are in an expression, do `op(s)`. Using this `op` notation, you can dig down into a very complex Maple expression and extract any sub-expression you want. You just do it recursively: `a = op(2,s)`; `b = op(7,a)`; and so on.

### Suppressing function arguments

Example 1: Sometimes it is necessary to display arguments of a function to make things "work right", but then in the result one might prefer not to see the arguments. Consider:

```
g := Diff(Diff(x^2*f,x),y);
```

$$g = \frac{\partial^2}{\partial y \partial x} x^2 f$$

```
value(g);
```

$$0$$

```
g := Diff(Diff(x^2*f(x,y),x),y);
```

$$g = \frac{\partial^2}{\partial y \partial x} x^2 f(x,y)$$

```
value(g);
```

$$2x \left( \frac{\partial}{\partial y} f(x,y) \right) + x^2 \left( \frac{\partial^2}{\partial y \partial x} f(x,y) \right)$$

```
subs(f(x,y) = op(0,f(x,y)), %);
```

$$2x \left( \frac{\partial}{\partial y} f \right) + x^2 \left( \frac{\partial^2}{\partial y \partial x} f \right)$$

```
op(0,f(x,y));
```

$$f$$

When the undefined function `f` has no explicit arguments, the deferred `Diff` differentiations activated by the `value` statement give the result 0 since Maple assumes `f` is a constant. This is repaired by changing `f` to `f(x,y)`. If expressions are long and/or there are many arguments, it might be desirable to suppress these arguments in a final result, and the above shows one way to do this by extracting the function name using the `op` command. See elsewhere in this document for comments on `Diff`, `value`, `%` and `subs`.

Example 2: This fancier example illustrates several things at once: how to get nice subscripts on a vector function, how to tell Maple not to throw out derivatives of unknown functions, and how to get Maple to suppress function arguments after they have been explicitly added. It is the suppression of the arguments that involves the "op" command which is used below to "pick off" just the function name as `op 0`, causing the arguments of the function to go away, as in the previous example. Things are very delicate because Maple always wants to compute derivatives of things which it often interprets as constants. One constantly has to tell Maple to "defer" evaluation by various methods. One is by using `Diff` instead of `diff`

when Diff occurs perhaps in a sum structure. Another is by putting apostrophes around the subs command shown below (another example of doing this appears in Example 5 of the Procedures section below).

So, consider the following piece of code which generates an expression for a variable q (the bulk of the code has been snipped away). The three functions  $B_r$ ,  $B_\theta$  and  $B_\phi$  are never specified, they are just some generic functions of the variables  $(r,\theta,\phi)$ .

```

xp[1] := r:
xp[2] := theta:
xp[3] := phi:
Bp := vector( [B[xp[1]],B[xp[2]],B[xp[3]] ] );
 $Bp := [B_r, B_\theta, B_\phi]$ 

```

```

.....
q;

```

$$-2 \frac{B_r(r, \theta, \phi)}{r^2} + 2 \frac{\frac{\partial}{\partial r} B_r(r, \theta, \phi)}{r} + \left( \frac{\partial^2}{\partial r^2} B_r(r, \theta, \phi) \right) - 2 \frac{\cos(\theta) B_\theta(r, \theta, \phi)}{r^2 \sin(\theta)} - 2 \frac{\frac{\partial}{\partial \theta} B_\theta(r, \theta, \phi)}{r^2} - 2 \frac{\frac{\partial}{\partial \phi} B_\phi(r, \theta, \phi)}{r^2 \sin(\theta)} + \frac{\frac{\partial^2}{\partial \phi^2} B_r(r, \theta, \phi)}{r^2 \sin(\theta)^2} + \frac{\cos(\theta) \left( \frac{\partial}{\partial \theta} B_r(r, \theta, \phi) \right)}{r^2 \sin(\theta)} + \frac{\frac{\partial^2}{\partial \theta^2} B_r(r, \theta, \phi)}{r^2}$$

The object q is generated by expressions such as the following,

$\text{Diff}(Bp[2](xp[1],xp[2],xp[3]),xp[2])$  meaning  $\partial_\theta B_\theta(r,\theta,\phi)$ ,

where the three arguments of  $B_\theta$  are explicitly shown to prevent Maple from thinking  $B_\theta$  is a constant with respect to any of the arguments. Once the object q has been computed, one might like to suppress all the arguments since they clutter up the expression. This seemingly trivial task can be done by the following additional code:

```

> 'subs(Bp[1](xp[1],xp[2],xp[3]) = op(0,Bp[1](xp[1],xp[2],xp[3])),q) ':
> 'subs(Bp[2](xp[1],xp[2],xp[3]) = op(0,Bp[2](xp[1],xp[2],xp[3])),%) ':
> qsup := 'subs(Bp[3](xp[1],xp[2],xp[3]) = op(0,Bp[3](xp[1],xp[2],xp[3])),%) ':
> qsup;

```

$$-2 \frac{B_r}{r^2} + 2 \frac{\frac{\partial}{\partial r} B_r}{r} + \left( \frac{\partial^2}{\partial r^2} B_r \right) - 2 \frac{\cos(\theta) B_\theta}{r^2 \sin(\theta)} - 2 \frac{\frac{\partial}{\partial \theta} B_\theta}{r^2} - 2 \frac{\frac{\partial}{\partial \phi} B_\phi}{r^2 \sin(\theta)} + \frac{\frac{\partial^2}{\partial \phi^2} B_r}{r^2 \sin(\theta)^2} + \frac{\cos(\theta) \left( \frac{\partial}{\partial \theta} B_r \right)}{r^2 \sin(\theta)} + \frac{\frac{\partial^2}{\partial \theta^2} B_r}{r^2}$$

The result is for display purposes only. It is unstable in the sense that if it is "evaluated", the last three terms vanish because Maple thinks the last three derivatives vanish, as shown in the next code line,

```

eval(qsup);

```

$$-2 \frac{B_r}{r^2} + 2 \frac{\frac{\partial}{\partial r} B_r}{r} + \left( \frac{\partial^2}{\partial r^2} B_r \right) - 2 \frac{\cos(\theta) B_\theta}{r^2 \sin(\theta)} - 2 \frac{\frac{\partial}{\partial \theta} B_\theta}{r^2} - 2 \frac{\frac{\partial}{\partial \phi} B_\phi}{r^2 \sin(\theta)}$$

## Frustration with Simplification

One frustration in using Maple is that it often has trouble simplifying the individual terms in a sum of complicated terms. Here is a example showing a work-around that sometimes helps:

```
[> restart;
> assume(theta>0,theta<Pi);
> Int(sin(x)/(sqrt(1-cos(x))*sqrt(cos(x)-cos(theta))),x=0..theta);
```

$$\int_0^{\theta} \frac{\sin(x)}{\sqrt{1-\cos(x)} \sqrt{\cos(x)-\cos(\theta)}} dx$$

```
> f := int(sin(x)/(sqrt(1-cos(x))*sqrt(cos(x)-cos(theta))),x=0..theta);
```

$$f := \frac{1}{4} \frac{\pi (2\sqrt{-\cos(\theta)} \sin(\theta) + \sin(\theta) + \sqrt{-2\cos(\theta)+2} \sqrt{2} \sqrt{\sin(\theta)})}{\sqrt{1-\cos(\theta)} \sqrt{\sin(\theta)}}$$

```
> acc := 0;
f1 := expand(f);
for n from 1 to nops(f1) do
  t[n] := simplify(op(n,f1));
  acc := acc + t[n];
od;
f2 := acc;
```

$$f2 = \pi$$

Here no amount of bashing on expression f with operations like "simplify(f)" will reduce it to  $\pi$  (unless you have a "lucky run" based on random internal ordering of things). Maple does not realize for example that it can cancel  $\sqrt{\sin\theta}$  even though the assume guarantees an unambiguous root. The code loop expands expression f into a sum of two terms and then simplifies those terms one at a time and constructs a new expression as the sum of those simplified terms. In this way, we discover that the integral equals  $\pi$ . Here is the text for the above example, paste it into a worksheet and it will run:

```
restart;
assume(theta>0,theta<Pi);
Int(sin(x)/(sqrt(1-cos(x))*sqrt(cos(x)-cos(theta))),x=0..theta);
f := int(sin(x)/(sqrt(1-cos(x))*sqrt(cos(x)-cos(theta))),x=0..theta);
acc := 0;
f1 := expand(f);
for n from 1 to nops(f1) do
  t[n] := simplify(op(n,f1));
  acc := acc + t[n];
od;
f2 := acc;
```

## Algebra with complex numbers evalc

Maple assumes variables are complex, so if one wants them to be real, one must say so. Consider for example,

```
restart;
for n in {a,b,c,d} do assume(n,real): od;
z := (a+I*b)/(c+I*d);
```

$$z = \frac{a+Ib}{c+Id}$$

```
abs(z^2);
```

$$\frac{a^2+b^2}{c^2+d^2}$$

```
Re(z);
```

$$\Re\left(\frac{a+Ib}{c+Id}\right)$$

Maple understands our intention that a,b,c,d are supposed to be real numbers, but it is still a bit recalcitrant in doing things. It easily finds the value of  $|z|^2$ , but balks at computing  $\text{Re}(z)$ . This problem is remedied by the evalc command:

```
z := evalc(z);
```

$$z = \frac{ac}{c^2+d^2} + \frac{bd}{c^2+d^2} + I\left(\frac{bc}{c^2+d^2} - \frac{ad}{c^2+d^2}\right)$$

```
Re(z): simplify(%);
```

$$\frac{ac+bd}{c^2+d^2}$$

```
Im(z): simplify(%);
```

$$-\frac{bc+ad}{c^2+d^2}$$

Here is another example :

```
assume(k,real,P,real);
f := (1-exp(-I*k*P))/(1-exp(-I*k));
```

$$f = \frac{1 - e^{(-IkP)}}{1 - e^{(-Ik)}}$$

```
abs(f^2): simplify(%);
```

$$\frac{-1 + \cos(kP)}{-1 + \cos(k)}$$

```
f1 := evalc(f);
```

$$f1 = \frac{(1 - \cos(kP))(1 - \cos(k))}{(1 - \cos(k))^2 + \sin(k)^2} + \frac{\sin(kP)\sin(k)}{(1 - \cos(k))^2 + \sin(k)^2} + I\left(\frac{\sin(kP)(1 - \cos(k))}{(1 - \cos(k))^2 + \sin(k)^2} - \frac{(1 - \cos(kP))\sin(k)}{(1 - \cos(k))^2 + \sin(k)^2}\right)$$

```
f_r := simplify(Re(f1));
```

$$f_r = -\frac{1}{2} \frac{1 - \cos(k) - \cos(kP) + \cos(kP)\cos(k) + \sin(kP)\sin(k)}{-1 + \cos(k)}$$

```
f_r := simplify(Im(f1));
```

$$f_r = \frac{1}{2} \frac{-\sin(kP) + \sin(kP)\cos(k) + \sin(k) - \sin(k)\cos(kP)}{-1 + \cos(k)}$$



## Algebra with trig functions subs

Maple sometimes does not do what you would normally do. Consider

```
f := a/(1-cos(x)^2);
```

$$f := \frac{a}{1 - \cos(x)^2}$$

```
simplify(%);
```

$$-\frac{a}{-1 + \cos(x)^2}$$

You might like to see  $\sin^2(x)$  in the denominator, but Maple sees no need to do that. So you have to tell Maple to do this "manually" :

```
subs(cos(x) = sqrt(1-sin(x)^2), f);
```

$$\frac{a}{\sin(x)^2}$$

Here is an example combining methods of the last two sections:

```
assume(k,real): assume(P,real);
```

```
f := (1-exp(-I*k*P))/(1-exp(-I*k));
```

$$f := \frac{1 - e^{(-I k P)}}{1 - e^{(-I k)}}$$

```
f1 := abs(f^2);
```

$$f1 := \frac{(1 - \cos(k P))^2 + \sin(k P)^2}{(1 - \cos(k))^2 + \sin(k)^2}$$

```
f2 := simplify(f1);
```

$$f2 := \frac{-1 + \cos(k P)}{-1 + \cos(k)}$$

```
f3 := subs(cos(k) = 1 - 2*sin(k/2)^2, f2);
```

$$f3 := -\frac{1 - 1 + \cos(k P)}{2 \sin\left(\frac{1}{2} k\right)^2}$$

```
f4 := subs(cos(k*P) = 1 - 2*sin(k*P/2)^2, f3);
```

$$f4 := \frac{\sin\left(\frac{1}{2} k P\right)^2}{\sin\left(\frac{1}{2} k\right)^2}$$

---

## NON MATRIX RELATED MATH FUNCTIONS/OPERATIONS

---

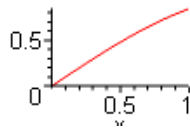
### How to Define a Function of Arguments

(1) the mapping operator " -> "

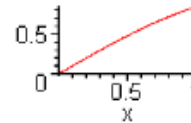
user defined functions

Here are two ways to do the same thing:

```
f := sin(x);  
f := sin(x)  
eval(f, x=2);  
sin(2)  
plot(f, x=0..1);
```



```
f := x -> sin(x);  
f := sin  
f(2);  
sin(2)  
plot(f(x), x=0..1);
```



If you want to evaluate something by putting an argument like  $f(x)$ , then you have to use the "mapping" method as shown on the right, and include the argument when plotting. You can do multiple variables for example

```
g := (x,y) -> sin(x)*cos(y) + x*y;
```

The arrow object  $\rightarrow$  has a little man page in Programming/Procedures and Functions, first item.

In Maple, a function is a function of a list of variables (arguments). I don't think you can define functions of more complicated objects, such as a function  $f(\mathbf{r})$  where  $\mathbf{r}$  is a vector. You can of course do  $f(x,y,z)$  with three scalar arguments.



## (2) unapply use to "functionalize" an existing expression

The name of this command is fairly obscure, but it can do something extremely useful. Suppose you have a long series of manipulations that create a complex expression "f" and in this expression the variables x and y happen to appear. You would like this thing to magically become a function F(x,y) so you can conveniently compute F(3,5), say. How do you do that? In other words, how to you create a function (which is a procedure call) in Maple if all you have is an expression?

The following method does not work:

```
f := sin(x);
                                     f := sin(x)
F := (x) ->f;
                                     F := x ->f
F(2);
                                     sin(x)
```

You want F(2) to be sin(2), not sin(x). As in the previous example above, if you type sin(x) to the right of the arrow, it *does* do what you want, but if you already have sin(x) sitting in an expression f, it does not work. The unapply command comes to the rescue:

```
f := sin(x);
                                     f := sin(x)
F := unapply(f, x);
                                     F := sin
F(2);
                                     sin(2)
```

Here is a two variable example which better fits our original statement of the conundrum:

```
f := sin(x)*y^3;
                                     f := sin(x)y^3
F := unapply(f, x, y);
                                     F := (x, y) -> sin(x)y^3
F(2, 3);
                                     27 sin(2)
evalf(%);
                                     24.55103052
```

Unapply "unapplies" all the arguments so you can "apply" them later.

### (3) unapply used to create the derivative of a function

Here is a famous application of the unapply idea. Suppose you want to make a Maple function which is the derivative of another function. You might dimly start out like this:

```
[> f := x^3;
                                     f:=x^3
[> df := (x) ->diff(f,x);
                                     df:=x -> diff(f,x)
[> df(2);
Error, (in df) wrong number (or type) of parameters in function diff
```

The problem is that x is set to 2, then it tries to compute diff(f,2) which makes no sense. We want to compute the derivative first, and THEN set x = 2. We want to "unapply" this x thing and get it in there at a later time. Here is one way to do it:

```
f := x^3;
                                     f:=x^3
df := unapply(diff(f,x),x);
                                     df:=x -> 3x^2
df(2);
                                     12
```

Here is another way to do it, where we first make f be a true function of x, not just an expression which includes x. In this case, notice that we put f(x) and not just f as the first argument of diff.

```
f := (x) -> x^3;
                                     f:=x -> x^3
h := unapply(diff(f(x),x),x);
                                     h:=x -> 3x^2
h(3);
                                     27
```

Maple has this notion built in to its D operator, and you get the same result more quickly this way

```
D(f)(3);
                                     27
```

The thing D(f) is a function, like sin is a function, and then the argument is 3, as in sin(3). See differentiation for multiple variables and partial derivatives.

## fsolve

Suppose you want to know where two curves intersect,

```
y := sin(x);
z := exp(-x);
E := fsolve(y-z, x, 0..3);
```

$$y := \sin(x)$$
$$z := e^{(-x)}$$
$$E := .5885327440$$

It solves for a value of the second argument which makes the first argument be 0. The search range of the second argument is given in the third argument. Once it finds one root, it stops, so you might have to provide your own range to get another root. Probably it is doing Newton iteration, the f in fsolve.

---

## solve variables which are "equations" M equations in M unknowns

You can of course do this in the matrix sense of  $A\mathbf{x}=\mathbf{b}$  using linsolve discussed in the matrix section below. Here is another way to do it, by example. We are solving four linear equations in 4 unknowns here. Notice in the first four lines that you can **define a variable to "be" an equation**. This is a little unusual for normal programming languages. This is where "=" is used in Maple, as opposed to ":=".

```
e1 := A + p = 2*x;
e2 := B + q = 3*x;
e3 := C + r = 4*x;
e4 := p + q + r = t;
```

$$e1 := A + p = 2x$$
$$e2 := B + q = 3x$$
$$e3 := C + r = 4x$$
$$e4 := p + q + r = t$$

```
S := solve( {e1, e2, e3, e4}, {p, q, r, x} );
```

$$S := \left\{ p = -\frac{7}{9}A + \frac{2}{9}B + \frac{2}{9}C + \frac{2}{9}t, q = -\frac{2}{3}B + \frac{1}{3}A + \frac{1}{3}C + \frac{1}{3}t, r = -\frac{5}{9}C + \frac{4}{9}A + \frac{4}{9}B + \frac{4}{9}t, x = \frac{1}{9}A + \frac{1}{9}B + \frac{1}{9}C + \frac{1}{9}t \right\}$$

```
p+r;
```

$$p + r$$

```
assign(S);
```

$$-\frac{1}{3}A + \frac{2}{3}B - \frac{1}{3}C + \frac{2}{3}t$$

Notice that the solutions p,q,r,x cannot be accessed by Maple until the assign(S) statement has been executed as shown above. This goes through the sequence of solutions and converts each solution into an assign statement. Prior to this p,q,r,x are just "text".

Here is an example using a directly stated equation,

```
solve(y=x^2 + 1,x);
```

$$\sqrt{y-1}, -\sqrt{y-1}$$

On the other hand, if you just put an expression, it solves the equation expression = 0 (ie, roots)

```
solve(x^2 + 1,x);
```

$$i, -i$$

**Two meanings of `_Z`: (1) dummy in RootOf; (2) "any integer"**

Solutions that solve comes up with sometimes contain the RootOf operator with its `_Z` variable, which is a bit confusing. Here is some sample code showing what this means:

```
f := RootOf(a*x^2 + b*x + c = 0,x);
```

$$f := \text{RootOf}(a\_Z^2 + b\_Z + c)$$

```
convert(f,radical);
```

$$\frac{1 - b + \sqrt{b^2 - 4ac}}{2a}$$

The symbol `_Z` is just a dummy variable like `x` in the equation  $ax^2+bx+c = 0$ . In general if we have  $f = \text{RootOf}(g(\_Z))$ , that means  $f = \_Z = a$  root of  $g(\_Z) = 0$ . For some reason, Maple does not actually solve for the root unless you force it to, as in the second command above. For polynomials of degree 5 and higher, neither humans nor Maple can find analytic roots. Maple can do some degree 4 if you are lucky.

The second use of `_Z` is illustrated in this example which illustrates several things at once:

```
f := a*cos(x)^2 + b*sin(x-c)^2;
```

$$f := a \cos(x)^2 + b \sin(x-c)^2$$

```
fd1 := diff(f,x);
```

$$fd1 := -2a \cos(x) \sin(x) + 2b \sin(x-c) \cos(x-c)$$

```
sols := solve(fd1=0,x);
```

$$sols :=$$

```
fd1m := combine(fd1,trig);
```

$$fd1m := -a \sin(2x) + b \sin(2x-2c)$$

```
sols := solve(fd1m=0,x);
```

$$sols := \frac{1}{2} \arctan\left(\frac{b \sin(2c)}{-a + b \cos(2c)}\right)$$

```
EnvAllSolutions := true;
```

```
sols := solve(fd1m=0,x);
```

$$sols := \frac{1}{2} \arctan\left(\frac{b \sin(2c)}{-a + b \cos(2c)}\right) + \frac{1}{2} \pi \_Z$$

Remember this is Maple V. Maple is unable to solve the equation  $fd1 = 0$ , but when it is advised to use simple trig identities by the combine statement, then it *can* solve the equation as shown. Then when the `_EnvAllSolutions` variable is turned on, it adds  $\frac{1}{2}\pi\_Z$  to the solution. (If the option to add tildes to the end of "assumed variables" is turned on, this appears as  $\frac{1}{2}\pi\_Z\sim$ ). It means  $(\pi/2)*N$  where N is any integer, so here `_Z` is sort of representing the field of integers, Z being a standard symbol for that use. Notice that this use of `_Z` is completely unrelated to the dummy variable usage above. In our example, the reason for adding  $(\pi/2)*N$  is that if  $2w = \tan^{-1}(z)$  gives a solution w, then  $2w+N\pi$  is also a solution, which then means that  $w + (\pi/2)N$  is also a solution. In newer Maple, one might see  $\frac{1}{2}\pi\_Z1\sim$  or  $\frac{1}{2}\pi\_Z2\sim$  and so on, to allow for multiple different arbitrary integers. Also, one can say `solve(fd1m=0,x,allsolutions=true)` to set the env variable shown, but in Maple V you must do that as shown above.

## **dsolve      solving differential equations**

The `dsolve` command can solve ODE's (and systems of ODE's) of any order, and it can do so either analytically (though it may not succeed) or numerically. The ODE's need not be *linear* ODE's. There are many features and options in this large sub-world of Maple, see Help. Maple has a separate command `pdsolve` for finding analytic (but not numerical) solutions to PDE's. We are talking Maple V here and no doubt both these worlds have expanded. Below we do three examples first analytically, then we repeat them numerically. The examples are a first-order ODE, a second-order ODE, and a system of 2 ODE's. A final numerical Example 4 shows how to use `dsolve` to plot magnetic field lines.

### **ANALYTIC**

**Example 1.** In our first basic example, we specify an ODE and then have Maple solve it,

```
restart; with(plots):
eq1 := diff(u(x),x) + 3*x = 0; # ODE equa to be solved
      eq1 :=  $\left(\frac{\partial}{\partial x} u(x)\right) + 3x = 0$ 
f := dsolve(eq1); # find solution (if it can)
      f :=  $u(x) = -\frac{3}{2}x^2 + \_C1$ 
```

Integration constants begin with an underbar as for `_C1` shown. There is no mechanism for stating a boundary condition in this form of the `dsolve` command. If you happen to omit the `= 0` from `eq1`, then `eq1` is an expression instead of an equation, but the `dsolve` call then adds the `= 0` for you, assuming that is what you meant.

The object `f` is not an expression, it is an equation, just as object `eq1` is an equation. We can obtain the right side of the equation `f` by picking off the second operand as follows,



```
u := op(2,f); # pick off right side of equation
```

$$u := -\frac{3}{2}x^2 + \_C1$$

```
u := rhs(f); # same action as above
```

$$u := -\frac{3}{2}x^2 + \_C1$$

Perhaps more usefully, we would like to have  $u(x)$  be a function of  $x$ , so instead of the above command we could do this (see elsewhere in this document for "unapply"),

```
u := unapply(rhs(f),x); # make u be a "function" of x
```

$$u := x \rightarrow -\frac{3}{2}x^2 + \_C1$$

```
u(y); # verify that u(x) behaves as a function
```

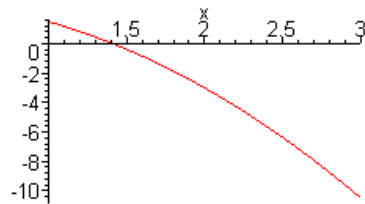
$$-\frac{3}{2}y^2 + \_C1$$

To plot the solution, we set in a value for the constant, and plot away,

```
\_C1 := 3; # set constant like any other constant
```

$$\_C1 = 3$$

```
plot(u(x),x=1..3);
```



## Example 2: 2nd order

```
restart; with(plots):
```

```
eq1 := diff(u(x),x,x) + 3*x = 0;
```

$$eq1 := \left( \frac{\partial^2}{\partial x^2} u(x) \right) + 3x = 0$$

```
f := dsolve(eq1);
```

$$f := u(x) = -\frac{1}{2}x^3 + \_C1 x + \_C2$$

and see above for how to pick off the function  $u(x)$ . Now there are two integration constants.

### Example 3: A system

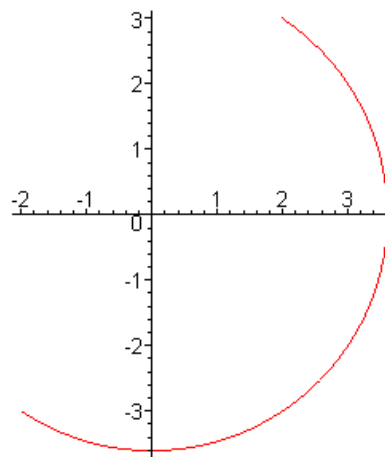
```
restart;
eq1 := diff(x(t),t) = y(t); # this system has 2 equa
      eq1 :=  $\frac{\partial}{\partial t}x(t) = y(t)$ 
eq2 := diff(y(t),t) = -x(t);
      eq2 :=  $\frac{\partial}{\partial t}y(t) = -x(t)$ 
f := dsolve({eq1,eq2},{x(t),y(t)}); # find solution
      f := {x(t) = cos(t)_C1 + sin(t)_C2, y(t) = -sin(t)_C1 + cos(t)_C2}
f; # solution is a set of two equations
      {x(t) = cos(t)_C1 + sin(t)_C2, y(t) = -sin(t)_C1 + cos(t)_C2}
f[1]; # first equation
      x(t) = cos(t)_C1 + sin(t)_C2
```

The output "f" of the dsolve command is now a *set* (recall that {...} means a set) of two equations. In the command itself we specified a *set* of two equations which comprise the system to be solved, followed by a *set* of the functions we want Maple to solve for. As before, we extract functions x(t) and y(t),

```
x := unapply(rhs(f[1]),t); # create a function x(t)
      x := t → cos(t)_C1 + sin(t)_C2
y := unapply(rhs(f[2]),t); # create a function y(t)
      y := t → -sin(t)_C1 + cos(t)_C2
y(T);
      -sin(T)_C1 + cos(T)_C2
```

At this point we could set in some constants and do a plot,

```
_C1 := 2: _C2 := 3:
plot([x(t),y(t),t = 0..Pi],scaling=constrained);
```



## NUMERICAL

If Maple cannot find an analytic solution to your ODE or system of ODE's, it can generate a numeric solution without too much trouble. We shall repeat the above examples numerically.

### Example 1 (numeric)

The first step is to "make the call" :

```
restart; with(plots):
eq1 := diff(U(x),x) + 3*x = 0;

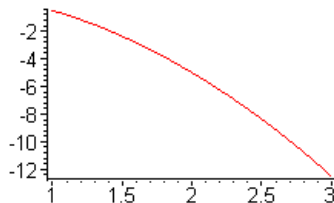
$$eq1 := \left( \frac{\partial}{\partial x} U(x) \right) + 3x = 0$$

f := dsolve({eq1,U(0) = 1},{U(x)},type=numeric,
output=listprocedure);
f:= [x = (proc(x) ... end), U(x) = (proc(x) ... end)]
f[2];
U(x) = (proc(x) ... end)
```

Here we use "some other symbol" (namely U) for the function name, so we can safely use u(x) as described below. Notice that the equation and boundary condition(s) appear in the first set {...}, while the function(s) you want to know about appear in the second set. The type=numeric of course instructs Maple to do a numerical solution, and the output= item tells Maple to present its solution as a "list of procedures". Actually, what you get is a list of two equations. The second equation says that dsolve has created some procedure called U(x) which evaluates the solution of our ODE at some point x. The first equation is a sort of identity procedure x which we won't worry about. Since this is a numeric solution, one might visualize the first dummy procedure "x" as providing the x values and the second procedure "U" as providing the y values for a plot of the solution.

We can take a quick look at some value of x, and do a fast plot as well using the special odeplot call,

```
f(2.53245);
[x(2.53245) = 2.53245, U(x)(2.53245) = -8.619954503749995]
odeplot(f, [x,U(x)], 1..3);
```



where [x,U(x)] specifies the two axes of the desired plot. The actual solution U(x) is a *procedure* which probably has a set of constants for the solution and interpolates them in some reasonable fashion to provide the illusion of a continuous solution.

But if you want to *use* this solution, you have to jump through another hoop which is this (where we now arrive at the desired u(x) object) ( rhs means to extract the right hand side of an equation)

```

u := rhs(f[2]);
                                     u := proc(x) ... end
u(2.53245);
                                     -2.055784594340562
u(0);
                                     1.

```

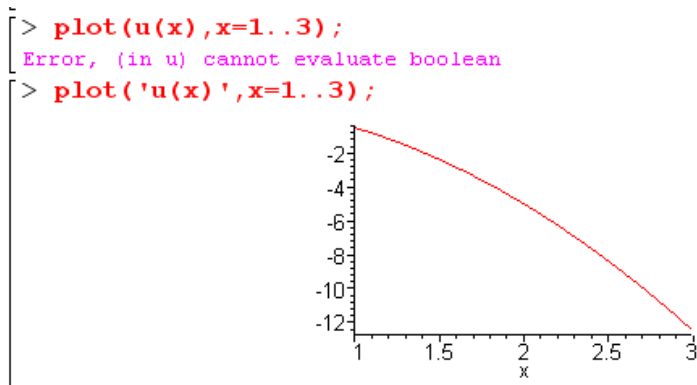
Now we can evaluate  $u(x)$  for any  $x$ , and we confirm the boundary condition in the last line.  
 Just for fun, we can examine the pedigree of  $u$ ,

```

type('u(x)', function);
                                     true
type(u, operator);
                                     false
type(u, procedure);
                                     true

```

This is the same as the pedigree of  $\sin$  as discussed in the  $\rightarrow$  operator section. The reason we have to add the single quotes in the first line is that this function does not have the little protective "wrapper" to repair a certain "first time called" problem. This is discussed in our section on Procedures, Example 4. We could add the wrapper which tests "if type(x,numeric)" as shown there, or we can just use the single quotes to delay evaluation one time and the problem is fixed. Here is another example where the single quotes are needed for this same reason (the function is probed at the start without a numeric argument and this kicks out an error)

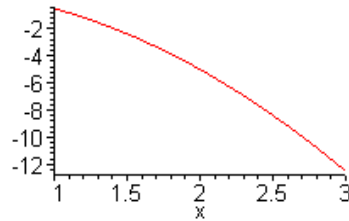


Here is the wrapper fix just for the record:

```

u_ := proc(x) if type(x,numeric) then u(x) else 'u_(x)'; fi end;
      u_ := proc(x) if type(x,numeric) then u(x) else 'u_(x)' fi end
plot(u_(x),x=1..3);

```



I think this little issue has been repaired in later versions of Maple.

If you want the derivative of  $u(x)$ , you cannot just use  $\text{diff}(u(x),x)$  or  $\text{diff}('u(x)',x)$  because  $u(x)$  is not an expression which can be differentiated, it is really a procedure call. One solution is to reformulate the ODE and solve it for the derivative (which in this example happens to be trivial). Another way is to just roll your own approximate derivative,

```

Du := x -> (u(x+1e-6)-u(x))/1e-6;
Du(1);

```

-3.0000

In the next example, `dsolve` provides both the function and the derivative since it involves a 2nd order ODE.

### Example 2 (numeric)

First we have the call to `dsolve`,

```

restart; with(plots):
eq1 := diff(U(x),x,x) + 3*x = 0;

```

$$eq1 = \left( \frac{\partial^2}{\partial x^2} U(x) \right) + 3x = 0$$

```

f := dsolve({eq1,U(0)=1,D(U)(0)=2},{U(x)},type=numeric,
output=listprocedure);

```

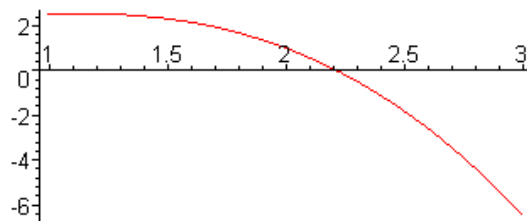
$$f := \left[ x = (\text{proc}(x) \dots \text{end}), U(x) = (\text{proc}(x) \dots \text{end}), \frac{\partial}{\partial x} U(x) = (\text{proc}(x) \dots \text{end}) \right]$$

The new feature here is that there are now two boundary conditions required, and they both go into the set with the equation as shown. Instead of saying  $u'(0) = 2$ , we write  $D(u)(0) = 2$  or  $(D@@1)(u)(0) = 2$ . For a third-order ODE, one would specify a third boundary condition as  $(D@@2)(u)(0) = 0$  and it goes into the set as well. For a quick sample point and a quick plot,

```
f(2.53245);
```

```
[x(2.53245) = 2.53245, U(x)(2.53245) = -2.055784594340564, (∂/∂x U(x))(2.53245) = -7.619954503750000]
```

```
odeplot(f, [x, U(x)], 1..3);
```



We can "extract" the function  $u(x)$  as in the previous example,

```
u := rhs(f[2]);
```

```
u := proc(x) ... end
```

```
u(2.53245);
```

```
-2.055784594340562
```

The first derivative can be similarly extracted,

```
Du := rhs(f[3]);
```

```
Du := proc(x) ... end
```

```
Du(2.53245);
```

```
-7.619954503750000
```

### Example 3 (numeric) A system

The call is

```
restart; with(plots):
```

```
eq1 := diff(X(t), t) = Y(t);
```

$$eq1 := \frac{\partial}{\partial t} X(t) = Y(t)$$

```
eq2 := diff(Y(t), t) = -X(t);
```

$$eq2 := \frac{\partial}{\partial t} Y(t) = -X(t)$$

```
f := dsolve({eq1, eq2, X(0) = 0, Y(0) = 1}, {X(t), Y(t)}, type=numeric,  
output=listprocedure);
```

```
f := [t = (proc(t) ... end), Y(t) = (proc(t) ... end), X(t) = (proc(t) ... end)]
```

Notice that the two boundary conditions are in the first set with the two equations. Extraction of the functions  $x(t)$  and  $y(t)$  proceeds as before,

```

x := rhs(f[2]);
y := rhs(f[3]);
x(2+Pi);

```

```

x := proc(t) ... end
y := proc(t) ... end
.4161468464895813

```

Sometimes one sees the extraction done slightly differently,

```

x := subs(f, X(t));
x := subs(f, Y(t));

```

```

x := proc(t) ... end
x := proc(t) ... end

```

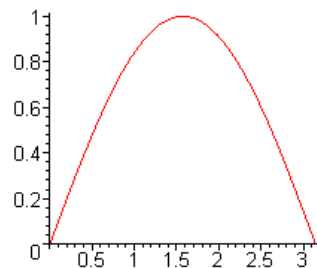
The results are clearly just the same, but the advantage is that you don't have to know the number of the equation in list `f` where your function of interest resides. In the `subs` syntax, the first item here is a list of three replacement expressions while the second item is the object into which those substitutions are made. So two of the substitutions in list `f` do nothing, while one replaces `X(t)` by `proc(t) ... end` (for example).

We now have several `odeplot` choices:

```

odeplot(f, [t, X(t)], 0..Pi);

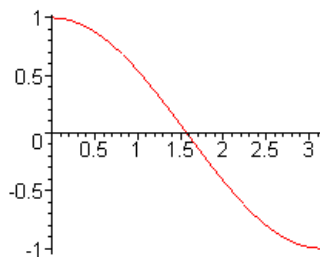
```



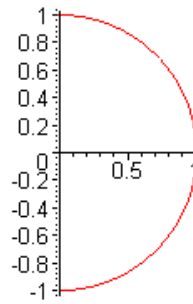
```

odeplot(f, [t, Y(t)], 0..Pi);

```

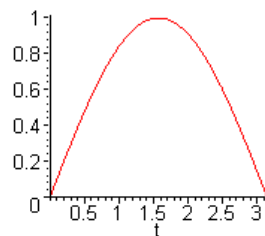


```
odeplot(f, [X(t), Y(t)], 0..Pi, scaling = constrained);
```

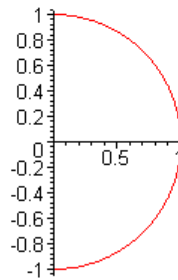


Alternatively, we can use the usual plotting routines instead, bearing in mind the first-time issue requiring the quotes just discussed in the numeric Example 1 above:

```
plot('x(t)', t=0..Pi);
```



```
plot(['x(t)', 'y(t)', t=0..Pi], scaling = constrained);
```



#### Example 4 (numerical)      Another system of two ODE's

Here is a real-world use of dsolve to plot magnetic field lines for two parallel cylinders each carrying a uniform current density and having radii  $a_1$  and  $a_2$  and center separation  $b$ . We do not give an explanation here, but just present the code as a useful example of dsolve solving a simple ODE system. This code runs much faster than an iterative tangent tracking routine with the similar accuracy. *That* kind of routine does  $\mathbf{r}(n+1) = \mathbf{r}(n) + \delta \frac{[\mathbf{H}(n)]}{|\mathbf{H}(n)|}$  with  $\delta$  some small value.

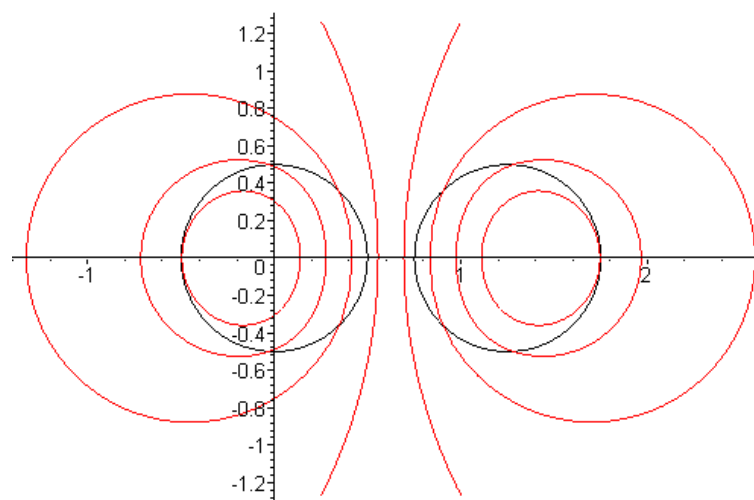


```

restart; alias(I=I, j=sqrt(-1)): with(plots):with(plottools):
K := I/(2*Pi):
I := 2*Pi:
Hx := - (y/r1)*H1 - (y/r2)*H2:
Hy := (x/r1)*H1 + ((x-b)/r2)*H2:
H1 := K*(Heaviside(r1-a1)*(1/r1) + Heaviside(a1-r1)*(r1/a1^2)):
H2 := -K*(Heaviside(r2-a2)*(1/r2) + Heaviside(a2-r2)*(r2/a2^2)):
r1 := sqrt(x^2+y^2):
r2 := sqrt((x-b)^2+y^2):
b := 1.25: a1 := .5: a2 := .5:
Hx_ := unapply(Hx,x,y):
Hy_ := unapply(Hy,x,y):
eq1 := diff(X(s),s) = Hx_(X(s),Y(s)):
eq2 := diff(Y(s),s) = Hy_(X(s),Y(s)):
# set Ncurves to plot
Ncurves := 8:
for J from 1 to Ncurves do
  print("starting curve J = ",J);
  # find parametric curve described by x=X(s) and y=Y(s)
  ff := dsolve({eq1,eq2,X(0)=J*b/(Ncurves+1)+.0001,Y(0)=0},{X(s),Y(s)},
type=numeric, method=gear, output=listprocedure):
  Y_ := subs(ff,Y(s)); #needed to directly access the Y(s) data
  # find range for parameter s so each curve just closes with no overlap
  S := 0;
  # closure goes too high
  while (Y_(S) >= 0) and (Y_(S) < b) do
    S := S +.1;
  od:
  p[J] := odeplot(ff,[X(s),Y(s)],-S...S,scaling=constrained,numpoints=140):
od:

c1 := circle([0,0],a1,color=black):
c2 := circle([b,0],a2,color=black):
display([seq(p[n],n=1..Ncurves)],c1,c2,thickness=1);

```



**rsolve solving difference equations (recursion equations)**

Example: Given  $X_{k+1} = aX_k + p$  with  $p = (1-a)/2$  and  $X_0 = 0$ , solve for  $X_k$ :

```
p := (1-a)/2;
Xk := rsolve({ X(k+1) = a*X(k) + p, X(0) = 0 }, X): simplify(%);
```

$$-\frac{1}{2}a^k + \frac{1}{2}$$

Example: Given  $o_{n+1} = -a o_n + b i_n$  with  $o_0 = c$ , solve for  $o_n$  where  $i_n$  is some unspecified sequence. This example shows how to cause the solution to be a function of  $n$ , and provides examples of the following other Maple functions, all discussed in this document: subs, unapply, value.

```
f := rsolve({o(n+1) = -a*o(n) + b*i[n], o(0) = c}, o);
```

$$f := c(-a)^n + \left( \sum_{n_0=1}^n (-a)^{(n-n_0)} b i_{n_0-1} \right)$$

```
f := subs(n[0] = j, f);
```

$$f := c(-a)^n + \left( \sum_{j=1}^n (-a)^{(n-j)} b i_{j-1} \right)$$

```
o := unapply(f, n);
```

$$o := n \rightarrow c(-a)^n + \left( \sum_{j=1}^n (-a)^{(n-j)} b i_{j-1} \right)$$

```
o(4);
```

$$c a^4 + \left( \sum_{j=1}^4 (-a)^{(4-j)} b i_{j-1} \right)$$

```
value(%);
```

$$c a^4 - a^3 b i_0 + a^2 b i_1 - a b i_2 + b i_3$$

If the subs step is not done, o(4) turns  $n_0$  which is  $n[0]$  into  $4_0$  which is  $4[0]$ . The result comes out the same at the end, but it just looks a bit odd with  $4[0]$  as the summation variable. As one can see by displaying the output in "Maple notation" instead of "Typeset notation", the sum in the o(4) line above is the inert Sum function. This is then executed when value is applied. We could have used  $i(n)$  in place of  $i[n]$  in the original rsolve line, then the output involves  $i(0)$ ,  $i(1)$  and so on.

## sum and add    product and mul

Use **sum** or **product** for symbolic, use **add** or **mul** for numeric. Add works with evalfh, sum does not.

```
add(n, n=1..4);
                                     10
sum(n, n=1..m);
                                     1/2 (m+1)^2 - 1/2 m - 1/2
mul(n, n=1..4);
                                     24
product(n, n=1..m);
                                     Γ(m+1)
```

Anything here can be an expression. See Help for discussion of putting things in single quotes.

---

## Integration            int Int

Here are some examples where we include the inert form at the start, then indefinite, then definite. On the right is an example of a **double integral**.

```
> Int(sin(x), x);
      ∫ sin(x) dx
> int(sin(x), x);
      -cos(x)
> int(sin(x), x=0.3..0.5);
      .0777539272

> Int(Int(x^2*y, x), y);
      ∫∫ x^2 y dx dy
> int(int(x^2*y, x), y);
      1/6 x^3 y^2
```

If you want to *force* a **numerical** integration, use evalf(Int...). Here is an example where the expected result is 0, but we use some extra arguments to make the integration go faster (see Help). Notice that the first attempt without evalf just returns a statement of the desired integral.

```
int(LegendreP(nu, mu, z)*LegendreP(nup, mu, z), z = -1..1);
      ∫-11 LegendreP(4.3, -2.3, z) LegendreP(6.3, -2.3, z) dz

evalf(Int(LegendreP(nu, mu, z)*LegendreP(nup, mu, z), z = -1..1, 4, _CCquad));
      .3000 10-9 - .3000 10-9 I
```

As just noted, Maple can do **multiple variable** integrations. This example illustrates a few more facts:

```
Int(Int(Int(sin(x)*cos(y)*z^2,x = 1..2),y = 3..4),z=5..6);
```

$$\int_5^6 \int_3^4 \int_1^2 \sin(x) \cos(y) z^2 dx dy dz$$

```
value(%);
```

$$\frac{91}{3}(-\cos(2) + \cos(1))(\sin(4) - \sin(3))$$

```
expand(%);
```

$$-\frac{91}{3} \cos(2) \sin(4) + \frac{91}{3} \cos(2) \sin(3) + \frac{91}{3} \cos(1) \sin(4) - \frac{91}{3} \cos(1) \sin(3)$$

```
int(int(int(sin(x)*cos(y)*z^2,x = 1..2),y = 3..4),z=5..6);
```

$$-\frac{91}{3} \cos(2) \sin(4) + \frac{91}{3} \cos(1) \sin(4) + \frac{91}{3} \cos(2) \sin(3) - \frac{91}{3} \cos(1) \sin(3)$$

```
evalf(%);
```

-26.05078866

```
evalf(Int(Int(Int(sin(x)*cos(y)*z^2,x = 1..2),y = 3..4),z=5..6));
```

-26.05078865

```
evalf(Int(Int(Int(sin(x)*cos(y)*z^2,x = 1..2,5),y = 3..4,5),z=5..6,5));
```

-26.051

For the first red command line, the inert Int is used, so we just get a statement of the integral. In this statement, notice that the integral signs with their endpoints are in the same order as the differentials. The second command line value(%) [ recall % is the last computed quantity ] causes the inert Int's to become active int's and then the integral is performed analytically as shown. The third red line just expands this into a sum of terms for comparison with the output of the fourth red line. Using value(%) is easier than rewriting the integral using int's as shown on this fourth red line. On the third last line we evaluate the analytic integral to see its numeric value. The second last line forces Maple to do a numerical calculation of the triple integral to an accuracy set by the global Digits parameter (10). The last line indicates that only 5 digits of accurate are required so the numerical integration is then much faster. Notice the close agreement between the analytic and numerical results at 10 decimal places.

Maple can do **contour integrations** in the following sense. Consider this example,

$$J = \oint dz \frac{f(z)}{(z-a)(z-b)^2} = 2\pi i \sum_{\text{residues}} \quad q = \frac{f(z)}{(z-a)(z-b)^2}$$

where we assume that f(z) is analytic within the contour and that poles a and b are located within the contour. The Maple calculation is then

```

restart;readlib(residue);
proc(f,a) ... end
q := f(z)/((z-a)*(z-b)^2);
q := 
$$\frac{f(z)}{(z-a)(z-b)^2}$$

J := 2*Pi*I*( residue(q,z=a) + residue(q,z=b));
J := 
$$2I\pi \left( \frac{f(a)}{-2ab+b^2+a^2} - \frac{D(f)(b) + \frac{f(b)}{-b+a}}{-b+a} \right)$$

f := z ->exp(5*z);
f := 
$$f = z \rightarrow e^{(5z)}$$

J;
J := 
$$2I\pi \left( \frac{e^{(5a)}}{-2ab+b^2+a^2} - \frac{5e^{(5b)} + \frac{e^{(5b)}}{-b+a}}{-b+a} \right)$$


```

Here we first compute the integral for arbitrary numerator function  $f(z)$ , then we select  $f(z) = e^{5z}$  to get the integral for that case. Maple's contribution here is the residue command. This method of course does not work if  $f(z)$  has a branch point within the contour, in which case it is not analytic inside the contour.

If the denominator is not in factored form, other Maple tools can help, for example:

```

f := b*z^2+2*A*z - a;
f := 
$$f = bz^2 + 2Az - a$$

s := solve(f,z);
s := 
$$s = \frac{1-2A+2\sqrt{A^2+ba}}{2}, \frac{1-2A-2\sqrt{A^2+ba}}{2}$$

s[1];

$$\frac{1-2A+2\sqrt{A^2+ba}}{2}$$

f1 := (z-s[1])*(z-s[2]);
f1 := 
$$f1 = \left( z - \frac{1-2A+2\sqrt{A^2+ba}}{2} \right) \left( z - \frac{1-2A-2\sqrt{A^2+ba}}{2} \right)$$

g := z^(m-1)*(a*z^2+b)/f1;
g := 
$$g = \frac{z^{(m-1)}(az^2+b)}{\left( z - \frac{1-2A+2\sqrt{A^2+ba}}{2} \right) \left( z - \frac{1-2A-2\sqrt{A^2+ba}}{2} \right)}$$

r1 :=residue(g,z=s[1]);
r1 = 
$$\frac{1}{2} \frac{-2 \left( \frac{-A+\sqrt{A^2+ba}}{b} \right)^{(m-1)} aA\sqrt{A^2+ba} + \left( \frac{-A+\sqrt{A^2+ba}}{b} \right)^{(m-1)} b^3 + \left( \frac{-A+\sqrt{A^2+ba}}{b} \right)^{(m-1)} ba^2 + 2 \left( \frac{-A+\sqrt{A^2+ba}}{b} \right)^{(m-1)} aA^2}{b\sqrt{A^2+ba}}$$


```

## Differentiation      diff   Diff   D

This opening barrage of examples shows how to do ordinary and partial derivatives.

```
f := x^5*y^3;
x$3;
diff(f,x);
diff(f,x,x); diff(f,x$2);
diff(f,x$3,y$2);
```

$$f := x^5 y^3$$
$$x, x, x$$
$$5 x^4 y^3$$
$$20 x^3 y^3$$
$$20 x^3 y^3$$
$$360 x^2 y$$

The second item is just a reminder that the \$n operator creates a sequence of identical entries.

Recall that Diff is the inert form of diff, which means it does not evaluate until told to do so by the value operator,

```
f := Diff(sin(x),x);
value(f);
```

$$f := \frac{\partial}{\partial x} \sin(x)$$
$$\cos(x)$$

The following example shows a situation where using Diff is absolutely essential. Matrix S just provides data for the example. The next two lines show that  $\text{diff}(S[a,a],\theta) = 0$  because  $S[a,a] = S_{a,a}$  is a constant since a is undefined at this point. In the next line  $A := \text{sum}(\text{diff}(S[a,a],\theta),a=1..3)$  the diff is executed first because it is in the innermost expression, and one gets  $\text{diff}(S[a,a],\theta) = 0$  for the reason just explained, so the sum is the sum of three zeros. We need to *defer* the diff operation until after the sum has been expanded so diff will then act on something like  $S[1,2]$  which is in fact a function of  $\theta$ . That deferral is provided by using Diff in the second command  $A := \text{sum}(\text{Diff}(S[a,a],\theta),a=1..3)$ . Since sum is lower case, the sum is carried out and then assigned to A. Then the final value command causes the Diff to execute.

```

S := matrix(3,3,S_);
      S :=  $\begin{bmatrix} \sin(\theta) \cos(\phi) & r \cos(\theta) \cos(\phi) & -r \sin(\theta) \sin(\phi) \\ \sin(\theta) \sin(\phi) & r \cos(\theta) \sin(\phi) & r \sin(\theta) \cos(\phi) \\ \cos(\theta) & -r \sin(\theta) & 0 \end{bmatrix}$ 
S[a,a];
       $S_{a,a}$ 
diff(S[a,a],theta);
      0
A := sum(diff(S[a,a],theta),a=1..3);
      A = 0
A := sum(Diff(S[a,a],theta),a=1..3);
       $A := \left(\frac{\partial}{\partial \theta} \sin(\theta) \cos(\phi)\right) + \left(\frac{\partial}{\partial \theta} r \cos(\theta) \sin(\phi)\right) + \left(\frac{\partial}{\partial \theta} 0\right)$ 
value(A);
       $\cos(\phi) \cos(\theta) - r \sin(\theta) \sin(\phi)$ 

```

The D operator (see unapply discussion earlier) can provide higher derivatives as follows:

```

f := (x) ->x^6;
       $f := x \rightarrow x^6$ 
D(f);
       $x \rightarrow 6x^5$ 
(D@@1)(f);
       $x \rightarrow 6x^5$ 
(D@@2)(f);
       $x \rightarrow 30x^4$ 
(D@@2)(f)(sqrt(2));
      120

```

If we don't specify a function f, it is easier to see what is going on:

```

D(f);
      D(f)
(D@@1)(f);
      D(f)
(D@@2)(f);
       $(D^{(2)})(f)$ 
(D@@2)(f)(sqrt(2));
       $(D^{(2)})(f)(\sqrt{2})$ 

```

Here are examples which involve partial derivatives

```

f := (x,y) -> x^3*y + 3*y^3*x;
                                      $f := (x,y) \rightarrow x^3 y + 3 y^3 x$ 
D[1](f) ;
                                      $(x,y) \rightarrow 3 x^2 y + 3 y^3$ 
D[2](f) ;
                                      $(x,y) \rightarrow x^3 + 9 y^2 x$ 
D[1,2](f) ;
                                      $(x,y) \rightarrow 3 x^2 + 9 y^2$ 
D[1](D[1,2](f)) ;
                                      $(x,y) \rightarrow 6 x$ 
D[1](D[1,2](f))(1,2) ;
                                     6

```

The object  $D[1](f)$  is  $\partial f/\partial x$  because  $x$  is the first argument in the argument list of  $f$ . The objects shown above are  $\partial_x f$ ,  $\partial_y f$ ,  $\partial_x \partial_y f$ ,  $\partial_x^2 \partial_y f$ ,  $[\partial_x^2 \partial_y f](1,2)$ . I don't think there is a way to combine the @@ notation with the partial derivatives, so you have to build up higher derivatives manually as shown. We repeat the above with no function defined

```

D[1](f) ;
                                      $D_1(f)$ 
D[2](f) ;
                                      $D_2(f)$ 
D[1,2](f) ;
                                      $D_{1,2}(f)$ 
D[1](D[1,2](f)) ;
                                      $D_{1,1,2}(f)$ 
D[1](D[1,2](f))(1,2) ;
                                      $D_{1,1,2}(f)(1,2)$ 

```

The Help page on D has yet more to say, but we have given the main ideas here.

---



## piecewise and periodic functions

Here is an example of a piecewise function:

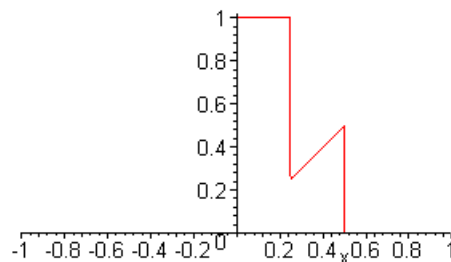
```
f := x -> piecewise(x > 0 and x < 1/4, 1, x > 1/4 and x < 1/2, x, 0);
```

$$f := x \rightarrow \text{piecewise}\left(0 < x \text{ and } x < \frac{1}{4}, 1, \frac{1}{4} < x \text{ and } x < \frac{1}{2}, x, 0\right)$$

```
f(x);
```

$$\begin{cases} 1 & -x < 0 \text{ and } x - \frac{1}{4} < 0 \\ x & \frac{1}{4} - x < 0 \text{ and } x - \frac{1}{2} < 0 \\ 0 & \text{otherwise} \end{cases}$$

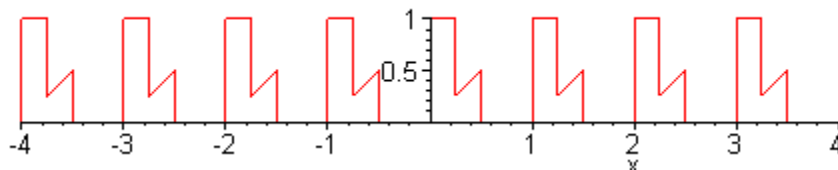
```
plot(f(x), x=-1..1, scaling=constrained, numpoints=100);
```



It is really like a little "case" statement. The rightmost argument is the default if  $x$  is in none of the ranges which you specify, which in our example is when  $x < 0$  or  $x > 1/2$ . You can symbolically or numerically integrate a piecewise function using the usual `int` command.

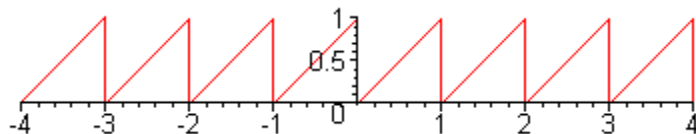
It is sometimes useful to construct a custom function like the above and then make it be a **periodic function** across the real axis. That can be done like so:

```
plot(f(x-floor(x)), x=-4..4, numpoints=2000, scaling=constrained);
```



where the argument of  $f(x)$  maps integer ranges of the real axis to the interval  $(0,1)$ ,

```
plot(x-floor(x), x=-4..4, scaling=constrained, numpoints=2000);
```



## Integral and Series Transforms

Maple is aware of the following integral transforms, which require **with(inttrans)**:

```
addtable fourier      fouriercos fouriersin hankel  
hilbert  invfourier invhilbert invlaplace invmellin  
laplace  mellin      savetable
```

Calls are provided for Fourier, Hilbert, Mellin and Laplace transforms and their inverses. The Fourier Sine, Fourier Cosine and Hankel transforms are their own inverses. There are also routines for the **Z transform** (ztrans, invztrans) and **FFT** (FFT, iFFT). The add and save table items allow the user to enlarge the internal transform lookup tables. Here is an example of a Laplace Transform, where one should note carefully the ordering of the last two arguments:

```
with(inttrans):  
laplace(exp(a*t), t, s);
```

$$\frac{1}{s-a}$$

```
invlaplace(1/(s-a), s, t);
```

$$e^{(a t)}$$

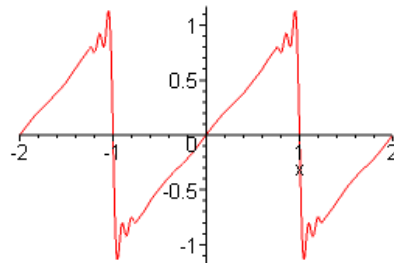
Series transforms like the Fourier Series Transform are done "manually" in Maple using an integral to compute coefficients and then the sum function to add up some number of terms. For  $f(x) = x$  on the interval  $(-1,1)$  for example we can write

```
bn := int(x*sin(n*Pi*x), x = -1..1);
```

$$b_n = -2 \frac{-\sin(n\pi) + n\pi \cos(n\pi)}{n^2 \pi^2}$$

```
f20 := sum(bn*sin(n*Pi*x), n = 1..20);
```

```
plot(f20, x=-2..2);
```



## dchange    doing PDE work with coordinate transformations

The dchange package does many things, here is a simple useful example. First, one can manually compute the following facts concerning how  $\partial_x$  and  $\partial_y$  are expressed in polar coordinates,

$$\begin{aligned}\partial_x &= \cos\theta \partial_r - (\sin\theta/r)\partial_\theta \\ \partial_y &= \sin\theta \partial_r + (\cos\theta/r)\partial_\theta\end{aligned}$$

Here is how to make Maple obtain this result:

```
with(PDEtools,dchange);

tr := {x = r*cos(theta), y=r*sin(theta)};
tr := {x = r cos(theta), y = r sin(theta)}
dfdx := dchange(tr,diff(f(x,y),x)): expand(%);
dfdy := dchange(tr,diff(f(x,y),y)): expand(%);
```

$$\cos(\theta) \left( \frac{\partial}{\partial r} f(r, \theta) \right) - \frac{\sin(\theta)}{r} \left( \frac{\partial}{\partial \theta} f(r, \theta) \right)$$
$$\sin(\theta) \left( \frac{\partial}{\partial r} f(r, \theta) \right) + \frac{\cos(\theta)}{r} \left( \frac{\partial}{\partial \theta} f(r, \theta) \right)$$

The transformation of interest (in this case, Cartesian to polar coordinates) is first installed in a set {...} called tr (for transformation). The dchange command is told about this transformation, and is given some partial differential expression as its second argument, and that expression is converted to a partial differential expression in the transformed coordinates. As one does by hand, Maple just uses the chain rule to obtain the result.

## series    series expansions about a point

A few examples show the basic idea. First argument is function to expand, second the point about which to expand, third is number of terms.

```
f := sqrt(1+x+x^2);
series(f,x=0,3); # first 3 terms about x = 0
series(f,x=1,3); # first 3 terms about x = 1
series(f,x=infinity,3); # first 3 terms about x = 1
```

$$f := \sqrt{1+x+x^2}$$
$$1 + \frac{1}{2}x + \frac{3}{8}x^2 + O(x^3)$$
$$\sqrt{3} + \frac{1}{2}\sqrt{3}(x-1) + \frac{1}{24}\sqrt{3}(x-1)^2 + O((x-1)^3)$$
$$x + \frac{1}{2} + \frac{3}{8x} - \frac{3}{16}\frac{1}{x^2} + O\left(\frac{1}{x^3}\right)$$

---

## Differential Operators: curl diverge grad laplacian vector Laplacian

For some reason Maple V decided to use "diverge" instead of "div" for divergence. Perhaps div suggests "divide" though div is not a reserved word. First, here are some 3D Cartesian examples:

```
> restart: with(linalg):
Warning, new definition for norm
Warning, new definition for trace
> E := [x^2*y,cos(x*z),z*x*y]; # specify a vector function
      E := [x^2 y, cos(x z), z x y]
> r := [x,y,z]; # names of coordinates
      r := [x, y, z]
> curl(E,r);
      [x z + sin(x z) x, -z y, -sin(x z) z - x^2]
> diverge(E,r);
      3 x y
> f := 1/(x^2+y^2+z^2); #specify a scalar function
      f :=  $\frac{1}{x^2 + y^2 + z^2}$ 
> grad(f,r);
       $\left[ -2 \frac{x}{(x^2 + y^2 + z^2)^2}, -2 \frac{y}{(x^2 + y^2 + z^2)^2}, -2 \frac{z}{(x^2 + y^2 + z^2)^2} \right]$ 
> laplacian(f,r):simplify(%);
       $2 \frac{1}{(x^2 + y^2 + z^2)^2}$ 
> diverge(grad(f,r),r): simplify(%);
       $2 \frac{1}{(x^2 + y^2 + z^2)^2}$ 
```

But Maple can do all these things in any many different coordinate systems, not just Cartesians, and not just the 11 classical systems like spherical and cylindrical and ellipsoidal. Here is the list:

At present, Maple supports the following coordinate systems:

In three dimensions - bipolarcylindrical, bispherical, cardioidal, cardioidecylindrical, casscylindrical, confocalellip, confocalparab, conical, cylindrical, elleycylindrical, ellipsoidal, hypercylindrical, invcasscylindrical, invellcylindrical, invoblspheroidal, invproospheroidal, logcoshcylindrical, logcylindrical, maxwellcylindrical, oblatespheroidal, paraboloidal, paraboloidal2, paracylindrical, prolatespheroidal, rectangular, rosecylindrical, sixsphere, spherical, tangencylindrical, tangentsphere, and toroidal.

In two dimensions - bipolar, cardioid, cassinian, cartesian, elliptic, hyperbolic, invcassinian, invelliptic, logarithmic, logcosh, maxwell, parabolic, polar, rose, and tangent.

And here is a simple example showing how the coordinate system is specified:

```
f := -cos(theta)*exp(I*phi)/r; # scalar function
```

$$f = -\frac{\cos(\theta) e^{(I\phi)}}{r}$$

```
c := [r, theta, phi];
```

$$c := [r, \theta, \phi]$$

```
grad(f,c, coords=spherical);
```

$$\left[ \frac{\cos(\theta) e^{(I\phi)}}{r^2}, \frac{\sin(\theta) e^{(I\phi)}}{r^2}, -\frac{I \cos(\theta) e^{(I\phi)}}{r^2 \sin(\theta)} \right]$$

Maple *assumes* a specific ordering of the curvilinear coordinates, which in the above example is  $r, \theta, \phi$ . The Maple orderings are stated in the Help system for each coordinate system, along with the equations which define the coordinate system. All systems assume the order  $[u, v, w]$ , and then for example,

spherical:

$$x = u \cdot \cos(v) \cdot \sin(w)$$

$$y = u \cdot \sin(v) \cdot \sin(w)$$

$$z = u \cdot \cos(w)$$

from which one can see that  $u, v, w = r, \theta, \phi$  in the usual notation.

If the above list of coordinate systems is not enough, you can add your own user coordinate system using the `adccoords` command (which of course is how Maple constructed the above list).

If you just want to *see* what a certain differential operator looks like in a certain coordinate system, follow this example which uses an unspecified scalar function  $g(r, \theta, \phi)$  and vector function  $\mathbf{T}(r, \theta, \phi)$  :

```
c := [r, theta, phi];
```

$$c := [r, \theta, \phi]$$

```
laplacian(g(r, theta, phi), c, coords=spherical): expand(%, r);
```

$$2 \frac{\partial}{\partial r} \frac{g(r, \theta, \phi)}{r} + \left( \frac{\partial^2}{\partial r^2} g(r, \theta, \phi) \right) + \frac{\cos(\theta) \left( \frac{\partial}{\partial \theta} g(r, \theta, \phi) \right)}{r^2 \sin(\theta)} + \frac{\frac{\partial^2}{\partial \theta^2} g(r, \theta, \phi)}{r^2} + \frac{\frac{\partial^2}{\partial \phi^2} g(r, \theta, \phi)}{r^2 \sin(\theta)^2}$$

```
T := [Tr(r, theta, phi), Tth(r, theta, phi), Tph(r, theta, phi)];
```

$$T := [Tr(r, \theta, \phi), Tth(r, \theta, \phi), Tph(r, \theta, \phi)]$$

```
diverge(T, c, coords=spherical): expand(%, r);
```

$$\left( \frac{\partial}{\partial r} Tr(r, \theta, \phi) \right) + 2 \frac{Tr(r, \theta, \phi)}{r} + \frac{\frac{\partial}{\partial \theta} Tth(r, \theta, \phi)}{r} + \frac{Tth(r, \theta, \phi) \cos(\theta)}{r \sin(\theta)} + \frac{\frac{\partial}{\partial \phi} Tph(r, \theta, \phi)}{r \sin(\theta)}$$

```
c1 := curl(T, c, coords=spherical):
```

```
c2 := [expand(c1[1], r), expand(c1[2], r), expand(c1[3], r)];
```

$$c_2 := \left[ \frac{\cos(\theta) T_{ph}(r, \theta, \phi)}{r \sin(\theta)} + \frac{\frac{\partial}{\partial \theta} T_{ph}(r, \theta, \phi)}{r} - \frac{\frac{\partial}{\partial \phi} T_{th}(r, \theta, \phi)}{r \sin(\theta)}, \right. \\ \left. \frac{\frac{\partial}{\partial \phi} T_r(r, \theta, \phi)}{r \sin(\theta)} - \frac{T_{ph}(r, \theta, \phi)}{r} - \left( \frac{\partial}{\partial r} T_{ph}(r, \theta, \phi) \right), \right. \\ \left. \frac{T_{th}(r, \theta, \phi)}{r} + \left( \frac{\partial}{\partial r} T_{th}(r, \theta, \phi) \right) - \frac{\frac{\partial}{\partial \theta} T_r(r, \theta, \phi)}{r} \right]$$

where in the last command we have expanded each component on r to get a conventional result.

The **vector Laplacian** can be obtained from ( $\star$  is Moon and Spencer notation)

$$\star \mathbf{B} \equiv \text{grad}(\text{diverge } \mathbf{B}) - \text{curl}(\text{curl } \mathbf{B}) = " \nabla^2 \mathbf{B} "$$

Recall that  $[\star \mathbf{B}]_i = \nabla^2 [B_i]$  only in Cartesian coordinates, where  $\nabla^2$  is the scalar Laplacian.

The reader interested in learning how Maple computes all these curvilinear coordinate expressions will find the answer in our document "Tensor Analysis and Curvilinear Coordinates".

---

## MATRIX/VECTOR RELATED MATH FUNCTIONS/OPERATIONS

---

The batting order for this section is this: we first mention briefly the notion of a **table**, and then talk about the **arrays**. The 1D and 2D array structures are also known as **Maple vectors** and **Maple matrices**. Then finally toward the end we discuss the slightly different vector and matrix notions which are part of the linear algebra library (package) called **linalg**.

---

### table

A table is a one-dimensional list of items each of which has a key. Here is the example Maple gives,

```
> F := table([sin=cos,cos=-sin]);
F = table([
  cos = -sin
  sin = cos
])
> F[sin];
cos
```

A table is used *to look things up*, so here you are perhaps looking up the derivatives of two trig functions, at least in terms of the function name. Accessing a table entry uses square brackets around the key.

If you don't specify a key, it assumes the keys are 1,2,... (such a key is called "an index")

```
> F := table([cos,-sin]);
F := table([
  1 = cos
  2 = -sin
])
```

Notice that the table command allows you to fill in some entries right in the command itself (an initialization list), although you can also add things at a later time,

```
> F[3] := tan;

> print(F);
table([
  1 = cos
  2 = -sin
  3 = tan
])
```

---

## array

Here is our first example which shows a 1 dimensional array which gets initialized in the array command

```
P := array(1..2, [a,b]);  
                                     P := [a, b]  
print(P);  
                                     [a, b]  
P[2];  
                                     b  
type(P, 'list'); type(P, 'table'); type(P, 'array');  
type(P, 'vector'); type(P, 'matrix');  
  
                                     false  
                                     true  
                                     true  
                                     true  
                                     false
```

Maple recognizes P as a table, an array, and a vector, but it is not a matrix. It is also not a "list", but it looks *exactly* like a list when it is displayed as by the print command above. Accessing an element again is with square brackets.

If one says

```
col := [red,blue,green];
```

then col is an array and by default it has legal indices 1,2,3 and col[1] = red. If some other indexing choice is desired, it must be explicitly stated, such as

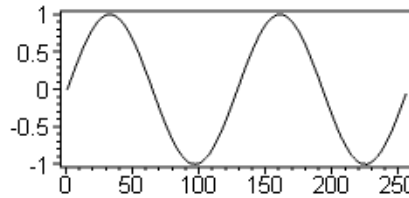
```
col := array(0..2,[red,blue,green]);
```

and then col[0] = red. Any integers are allowed for array indices.

Here is a little code showing how such a 1D array could be used



```
[> N := 256: k := 2: set some parameters
[> x := array(1..N): allocate the empty vector and give it a name
[> for n to N do x[n] := evalf(sin(2*Pi*k*(n-1)/N)) od: load it up with do loop
[> with(plots): listplot(x, axes=boxed); display with "listplot"
```



Now let's try a 2 dimensional array

```
P := array(1..2, 1..2, [[a,b], [c,d]]);
```

$$P := \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

```
P[1,2];
```

*b*

```
Q := evalm(P &* P);
```

$$Q := \begin{bmatrix} a^2+bc & ab+bd \\ ca+dc & bc+d^2 \end{bmatrix}$$

```
Q; print(Q);
```

$$Q$$

$$\begin{bmatrix} a^2+bc & ab+bd \\ ca+dc & bc+d^2 \end{bmatrix}$$

```
type(P, 'table'); type(P, 'array'); type(P, 'vector'); type(P, 'matrix');
```

*true*

*true*

*false*

*true*

```
evalm(3 * P);
```

$$\begin{bmatrix} 3a & 3b \\ 3c & 3d \end{bmatrix}$$

Notice first how the data gets initialized: it is a list of two items, each of which is a list of two items. The second line shows element access, while the third shows Maple can do matrix multiplication on such 2D arrays using the evalm command and using the special symbol &\*. The last line shows that the symbol \* is reserved to multiply a matrix by a constant. Maple recognizes the array P as a table, array, AND a matrix.

Next, we might as well throw in a 3 dimensional array:

```

> P := array(1..2,1..2,1..2,[ [[a,b],[c,d]], [[e,f],[g,h]] ] );
P := array(1..2, 1..2, 1..2, [
  (1, 1, 1) = a
  (1, 1, 2) = b
  (1, 2, 1) = c
  (1, 2, 2) = d
  (2, 1, 1) = e
  (2, 1, 2) = f
  (2, 2, 1) = g
  (2, 2, 2) = h
])
> P[1,2,2];
                                     d
> type(P, 'table'); type(P, 'array'); type(P, 'vector'); type(P, 'matrix');
                                     true
                                     true
                                     false
                                     false

```

The data initializer list is now a list of two items, each being a list of two items, each of those being a list of two items. Maple recognizes this thing as a table and an array, but not a matrix or vector.

I presume Maple can handle as many array dimensions as the user needs.

The elements of an array need not be the same type of object. Consider,

```

a := array(1..3):
a[1] := "joe":
a[2] := 3.6:
a[3] := [1,2,3]:
print(a);
                                     ["joe", 3.6, [1, 2, 3]]

```

Note: Reference to an element of an array requires that all array indices be specified.

### How to work with a long list of vectors

Imagine you have 1000 points of the form (x,y,z) and you want to store them in a reasonable manner so you can work with them as vectors.

1. One way is to have this list of vectors be a 2D array. This method is extremely inconvenient because everything then has to be done with individual vector components, since all array indices must be specified.

```

> a := array(1..101,1..3);
                                     a := array(1 .. 101, 1 .. 3, [ ])
> a[2];
Error, array defined with 2 indices, used with 1 indices
> a[2,1];
                                     a2,1
> a[2,1] := 0;
                                     a2,1 := 0
> a[2,1];
                                     0
> a[2,1] := 4: a[2,2] := 7: a[2,3] := 11:

```

You would like to say  $a[2] := [4,7,11]$  for the last line, but  $a[2]$  is illegal.

2. An alternative is to store the points as a sequence of vectors, where one gets the possible benefit of initializing the vectors to desired values :

```

> a := seq(vector([i,i,i]),i=1..4);
>
                                     a := [1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]
> a[2];
                                     [2, 2, 2]
> a[2] := [4,7,11];
Error, cannot assign to an expression sequence

```

The problem here is that, although  $a[2]$  is understood by Maple, it cannot be assigned.

3. Next, we can slightly upgrade the above idea by making  $a$  be a *list* instead of a sequence.

```

> a := [seq(vector([i,i,i]),i=1..4)];
>
                                     a := [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
> a[2] := [4,7,11];
                                     a2 := [4, 7, 11]
> b := [seq(vector([i,i,i]),i=1..101)]:
> b[2] := [4,7,11];
Error, assigning to a long list, please use arrays

```

This approach works, but the Catch 22 is that it does not work if the sequence has more than 100 elements!

4. Here we store the points as an array of lists so each vector is a list of three elements:

```

a := array(1..1000);
                                a := array(1..1000, [ ])
for i from 1 to 1000 do a[i] := [i,i,i]; od:
a[2];
                                [2, 2, 2]
a[2] := [4,7,11];
                                a2 := [4, 7, 11]
a[2];
                                [4, 7, 11]
a[3];
                                [3, 3, 3]
a[2] + a[3];
                                [7, 10, 14]
type(a, array);
                                true
type(a[2], list); type(a[2], vector);
                                true
                                false

```

Although array elements can all be different types, we have in the second command initialized each element of the array *a* to be a list of three elements which we think of as a vector, though in Maple the type is list, not vector. So storing a large number of 3D vectors as an array of lists seems to solve the problem. This avoids having a "long list" which Maple complained about in item 3 above.

As shown above, Maple will add the elements of two lists of the same size. Maple will *not* do component-wise addition of two items which are of different type, such as adding a list to an array,

```

A := array(1..3):
A[1] := 1: A[2] := 2: A[3] := 3:
A;
                                A
print(A);
                                [1, 2, 3]
B := [4, 5, 6];
                                B := [4, 5, 6]
print(B);
                                [4, 5, 6]
A + B;
                                A + [4, 5, 6]
print(A+B);
                                A + [4, 5, 6]
type(A, array): type(B, list);
                                true
                                true

```

Since A is an array, its elements must be accessed by an index. When A and B are "printed", they both look like lists, but only B is a list and A is an array. If in the above code one were to omit the declaration of A as an array, then A is interpreted as a table, and again the addition fails.

**matrix**    **"matrix vectors"**            **evalm**

Here is the same matrix built three different ways:

```
Q := array(1..2,1..3, [[a,b,c], [d,e,f]]);
```

$$Q = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
Q := matrix(2,3, [a,b,c,d,e,f]);
```

$$Q = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
Q := matrix(2,3, [[a,b,c], [d,e,f]]);
```

$$Q = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
type(Q, 'matrix'); type(Q, 'array');
```

*true*

*true*

**Key Fact:** There is no distinction between a matrix and a two-dimensional array.

### Matrix Vectors.

If a matrix has one row or one column, we shall call it a "matrix vector" (a `linalg` vector) which is just a special case of a matrix. As we shall see below, there is another animal called a "Maple vector" which is not the same thing. The keyword "vector" in Maple always refers to this Maple vector object. So consider

```

R := matrix(1,2,[a,b]);
C := matrix(2,1,[c,d]);

type(R,'vector'); type(C,'vector');

type(R,'array'); type(C,'array');

type(R,'matrix'); type(C,'matrix');

evalm(R &* C);

evalm(C &* R);

```

$$R := [a \quad b]$$

$$C := \begin{bmatrix} c \\ d \end{bmatrix}$$

*false*  
*false*  
*true*  
*true*  
*true*  
*true*  
  
 $[a \ c + b \ d]$   
  
 $\begin{bmatrix} a \ c & c \ b \\ d \ a & b \ d \end{bmatrix}$

Notice that, in the matrix  $R = [a \ b]$  above, there is no comma separating the elements. The last two lines show the expected results of contracting a row and a column vector with the row vector first on the left, and then on the right. The first is the dot product of the vectors, the second is a new matrix. The operator `evalm` is discussed just below.

## Math with Matrices

Conforming matrices can be added or subtracted with the usual `+` and `-` operators. The `*` operator is used to multiply a matrix by a scalar, while the operator `&*` is used to multiply two conforming matrices. Square matrices can be raised to a power with the usual `^` operator, as in  $A^3$ .

## Evalm

Recall the way the operator `evalf` is used to evaluate an expression to a floating point result:

```

a := sin(2); evalf(%);

```

$$a := \sin(2)$$

$$.9092974268$$

In a similar fashion, the operator **evalm** (evaluate matrix) is required to evaluate a matrix expression to get a result:

```
Q := matrix(2,3,[a,b,c,d,e,f]);
```

$$Q = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
S := matrix(3,2,[a,b,c,d,e,f]);
```

$$S = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

```
Q &* S;
```

$$Q \&* S$$

```
evalm(%);
```

$$\begin{bmatrix} a^2+bc+ce & ab+bd+cf \\ da+ce+fe & bd+ed+f^2 \end{bmatrix}$$

Matrices which do not "conform" properly generate an error message upon evaluation, as one would expect:

```
> Q &* Q;evalm(%);
```

$$Q \&* Q$$

```
Error, (in linalg[multiply]) non matching dimensions for vector/matrix product
```

```
> Q + S;evalm(%);
```

$$Q + S$$

```
Error, (in linalg[matadd]) matrix dimensions incompatible
```

### The zero matrix and the identity matrix in evaluation

In matrix evaluation, any matrix filled with zeros can be represented by the symbol 0.

Any identity matrix (diagonal of 1's) can be represented by the four-characters &\*().

```
evalm(Q + 0);
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
evalm(Q &* &*());
```

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

```
evalm(0 &* Q);
```

0

In the first line we add our 0 matrix causing no change. In the second, we right-multiply our matrix shown by an implied 3x3 identity matrix to get back the original matrix. In the last line we left-multiply the matrix Q by an implied 2x2 matrix of 0's and the result is a 3x2 matrix of zeros represented just as 0. One

might wonder why one would do any of the three operations. We shall see a reason below in our "unexpected behavior" subsection.

### Stand-alone zero matrix and identity matrix.

Here are fast ways to create these matrices,

```
matrix(3,3,[0$3^2]);          array(1..3,1..3,identity);
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where recall that 0\$3 generates 0,0,0, \$ being the repeat operator.

**Creating a matrix from a function f(i,j).** First, an example from the linalg[matrix] info page,

```
f := (i,j) -> i*j^2;
```

$$f = (i,j) \rightarrow ij^2$$

```
T := matrix(3,3,f);
```

$$T := \begin{bmatrix} 1 & 4 & 9 \\ 2 & 8 & 18 \\ 3 & 12 & 27 \end{bmatrix}$$

This very simple example verifies that, for a matrix constructed in this manner, the matrix displayed is  $T_{ij}$ , which is to say, i is the row index, as you would expect from the notation.

Here is another example showing the construction of a matrix  $T_{up}$  which is the transformation matrix which connects Cartesian to Spherical coordinates. The main point of this example is to show how easy it is in Maple to build an important matrix.

```
q[1] := r;
q[2] := theta;
q[3] := phi;
x[1] := r*sin(theta)*cos(phi);
x[2] := r*sin(theta)*sin(phi);
x[3] := r*cos(theta);
Tf := (i,j) -> diff(x[j],q[i]);
```

Note that: (1) i is the row index. (2) This is the "up-tilt" matrix.

$$Tf = (i,j) \rightarrow \frac{\partial}{\partial q_i} x_j$$

```
Tup := matrix(3,3,Tf);
```

$$T_{up} = \begin{bmatrix} \sin(\theta) \cos(\phi) & \sin(\theta) \sin(\phi) & \cos(\theta) \\ r \cos(\theta) \cos(\phi) & r \cos(\theta) \sin(\phi) & -r \sin(\theta) \\ -r \sin(\theta) \sin(\phi) & r \sin(\theta) \cos(\phi) & 0 \end{bmatrix}$$



The rows of  $T_{up}$  are the so-called tangent base vectors. Using the final transpose routine discussed below, one can then compute the metric tensor  $g_{up} = T_{up} T_{up}^T$  for spherical coordinates, which comes out being

$$g := \begin{bmatrix} 1 & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 - r^2 \cos^2(\theta) \end{bmatrix}$$

### Creating a matrix from a function f(i,j) using inert operators

In this example, 2x2 matrices U and D (D was unprotected then unassigned!) and column vector h were predefined. The 2x2 matrix P is constructed where each entry is a double sum involving a derivative. The variables  $q_{1,2}$  have been preset to r and  $\theta$ . The inert forms Sum and Diff have been used to cause deferral of their execution. If one uses sum and diff instead, matrix P comes out all zeros. The double for loop then causes each symbolic entry of matrix P to be evaluated using the value(..) command, at which time Diff and Sum are executed. The results are put into matrix X which is then displayed. The value command cannot simply be applied to matrix P. (See item below concerning when elements of matrices are evaluated.)

```
> P_ := (i,j) -> (1/h[i])*(1/h[j])* Sum( Sum(U[i,d]* Diff(D[k,d],q[j]) *h[k]*v[k] ,d=1..2), k=1..2);
```

$$P_{i,j} = \frac{\sum_{k=1}^2 \left( \sum_{d=1}^2 U_{i,d} \left( \frac{\partial}{\partial q_j} D_{k,d} \right) h_k v_k \right)}{h_i h_j}$$

```
> P := matrix(2,2,P_);
```

$$P = \begin{bmatrix} \frac{\sum_{k=1}^2 \left( \sum_{d=1}^2 U_{1,d} \left( \frac{\partial}{\partial r} D_{k,d} \right) h_k v_k \right)}{r} & \frac{\sum_{k=1}^2 \left( \sum_{d=1}^2 U_{1,d} \left( \frac{\partial}{\partial \theta} D_{k,d} \right) h_k v_k \right)}{r} \\ \frac{\sum_{k=1}^2 \left( \sum_{d=1}^2 U_{2,d} \left( \frac{\partial}{\partial r} D_{k,d} \right) h_k v_k \right)}{r} & \frac{\sum_{k=1}^2 \left( \sum_{d=1}^2 U_{2,d} \left( \frac{\partial}{\partial \theta} D_{k,d} \right) h_k v_k \right)}{r^2} \end{bmatrix}$$

```
> X :=matrix(2,2,[1,1,1,1]);
> for n from 1 to 2 do
  for m from 1 to 2 do
    X[n,m] := simplify(value(P[n,m]));
  od;
od;
> evalm(X);
```

$$\begin{bmatrix} 0 & -\frac{v_2}{r} \\ -\frac{v_2}{r} & \frac{v_1}{r} \end{bmatrix}$$

**Empty matrix.** The call matrix(m,n) creates an m by n matrix with unspecified elements.

**Displaying a Matrix: Unexpected Behaviors.** A matrix is a storage bin for expressions. The matrix elements are expressions just as are the non-matrix variables of Maple. Let's define the following matrix:

```
> restart;
> with(linalg):
Warning, new definition for norm
Warning, new definition for trace
> V := matrix([ [k, -3*k], [-k, 2*k] ]);
      V := 
$$\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$$

> V;
      V
> V[1,2];
      -3k
> print(V);
      
$$\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$$

> eval(V);
      
$$\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$$

```

The first perhaps unexpected behavior is that after you have created matrix V, when you type "V; " you might expect to see the matrix you just made, but all you see is "V", as if V had never been defined to be anything at all. But the next line V[1,2] shows that Maple really does know what V is, and confirms that the first index is the row index. If you want to "see" the matrix, use either print or eval as shown.

Now, suppose we set variable k = 4. Look what happens

```
k := 4;
      k := 4
eval(V);
      
$$\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$$

evalm(V);
      
$$\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$$

V[1,2];
      -12
print(V);
      
$$\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$$

```

So the second unexpected behavior is that, after we set k = 4, we expect that when we display the matrix whose elements are functions of k, those expressions will be evaluated with k = 4, but all we see is the original matrix unevaluated. Compare this to what happens at the top level in Maple

```

· restart;
· f := -3*k;
· f;
· k := 4;
· f;
· k := 5;
· f;
· unassign('k');
· f;

```

$f := -3k$   
 $-3k$   
 $k := 4$   
 $-12$   
 $k := 5$   
 $-15$   
 $-3k$

The object "f" really is the expression  $-3*k$  all the time. If we set  $k$  to some number, that fact is not changed. The difference is that when we type "f", we see the expression "evaluated", whereas in the matrix case we don't see it evaluated.

If you want to force Maple to evaluate the elements of a matrix so you can see them that way, you can do this: (we continue the code shown at the left above)

```

evalm( V &* &*() );
print(V);

```

$\begin{bmatrix} 4 & -12 \\ -4 & 8 \end{bmatrix}$   
 $\begin{bmatrix} k & -3k \\ -k & 2k \end{bmatrix}$

We multiply the matrix  $V$  by the unit matrix  $\&*()$ , and make it evaluate the elements. The original matrix  $V$  of expressions is still as it was.

## Maple vectors

### convert

We have seen above how matrices which have one row or one column may be thought of as vectors. In Maple, a "vector" is an object distinct from such matrices and as such is usually called "a Maple vector". Here is how you make a Maple vector: ( a Maple vector is what is associated with the keyword 'vector')

```

P := vector([e,f]);
print(P);
P[2];
type(P, 'list');type(P, 'matrix');
type(P, 'array');type(P, 'vector');

```

$$P := [e, f]$$

$$[e, f]$$

$$f$$

$$false$$

$$false$$

$$true$$

$$true$$

**Key Fact:** There is no distinction between a Maple vector and a one-dimensional array.

The object which appears as the argument of the vector command above is the list [a,b]. But the vector this produces, which appears as [a,b], is NOT a list, although it looks like a list. Only the type command knows! A Maple vector has the appearance of a list and is therefore always displayed as a horizontal object with comma separated elements, never as a vertical matrix "column vector", for example. The Maple vector can act as either a column vector or a row vector as required. Continuing the code from above, we find:

```

Q := matrix(2,2,[a,b,c,d]);
V := evalm(P &* Q);
V := evalm(Q &* P);
type(V, 'list');type(V, 'matrix');
type(V, 'array');type(V, 'vector');

```

$$Q = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$V := [ea+fc, eb+fd]$$

$$V := [ea+bf, ce+fd]$$

$$false$$

$$false$$

$$true$$

$$true$$

One might wonder what happens if you try to contract two Maple vectors together with evalm:

```

R := vector([g,h]);
evalm(R &* P);
evalm(R &* R);

```

$$R = [g, h]$$

$$ge + hf$$

$$\text{op}(g^2, h^2)$$

It works OK as long as you don't use the same vector. Evidently Maple has some difficulty casting the a Maple vector as both a row and column vector at the same time.

### Mixing Maple vectors with matrix vectors

It is possible to mix "Maple vectors" with "matrix vectors" in some situations, but one is never quite sure what the result will be. Consider this code, where R is a Maple vector, while S and T are "matrix vectors",

```

R := vector([g,h]);
S := matrix(2,1,[a,b]);
T := matrix(1,2,[c,d]);
M := evalm(R &* S);
type(M, 'matrix');type(M, 'list');type(M, 'vector');
N := evalm(S+T);
error, (in linalg/matadd) matrix dimensions incompatible
N := evalm(R + T);
error, (in linalg/matadd) matrix dimensions incompatible
N := evalm(R + S);

```

$$R = [g, h]$$

$$S = \begin{bmatrix} a \\ b \end{bmatrix}$$

$$T = [c \quad d]$$

$$M = [ga + hb]$$

*false*  
*false*  
*true*

$$N = \begin{bmatrix} g+a \\ h+b \end{bmatrix}$$

*true*  
*false*  
*false*

Our attempted dot product M produces a vector of one element which is this dot product. We then try three different additions. We expect the first two fail since we add a column matrix vector to a row matrix vector. The next two show that Maple prefers to think of the Maple vector as a column vector rather than a row vector. The successful sum is a matrix column vector.

Luckily, it is possible to convert back and forth between the two kinds of vectors:

```

· convert(R, matrix);
                                 $\begin{bmatrix} g \\ h \end{bmatrix}$ 
· convert(S, vector);
                                 $[a, b]$ 
· convert(T, vector);
                                 $[c, d]$ 

```

Going from a Maple vector to a matrix vector, we get a column vector, confirming the conjecture just made above. Either matrix vector converts into the same Maple vector object.

**Displaying a Maple vector: Unexpected Behaviors.** In the matrix section above, we discussed certain unexpected behaviors of Maple in regard to displaying matrices. That entire discussion applies as well to Maple vectors. For example, here we compute the gradient of a scalar function, which is a Maple vector,

```

> restart;
> with(linalg);
Warning, new definition for norm
Warning, new definition for trace
> g := grad(x^2 + 3*y^3 - 2*z, [x, y, z]);
                                 $g := [2x, 9y^2, -2]$ 
> g;
                                 $g$ 
> eval(g);
                                 $[2x, 9y^2, -2]$ 
> x := 2;
                                 $x := 2$ 
> eval(g);
                                 $[2x, 9y^2, -2]$ 
> g[1];
                                 $4$ 
> evalm(&*( ) &* g );
                                 $[4, 9y^2, -2]$ 
> evalm(g &* &*());
Error, (in linalg[multiply]) non matching dimensions for
vector/matrix product

```

The last two lines again confirm the fact that Maple thinks of a Maple vector as a column vector, so it is the first line where we left-multiply by the identity matrix `&*( )` which lets us see our Maple vector in its evaluated form.

## Preliminary comments on the linalg package

Above we discussed arrays and in particular the 2 and 1 dimensional arrays which are also known as matrices and Maple vectors. Maple provides some limited ability to fiddle with these objects, but not much.

The linalg "package" (an external library loaded by saying `with(linalg)`) contains a large number of routines which enhance Maple's ability to do things with matrices and Maple vectors. Here is a list of the commands.

<a href="#">GramSchmidt</a>	<a href="#">JordanBlock</a>	<a href="#">LUdecomp</a>	<a href="#">QRdecomp</a>	<a href="#">addcol</a>
<a href="#">addrow</a>	<a href="#">adjoint</a>	<a href="#">angle</a>	<a href="#">augment</a>	<a href="#">backsub</a>
<a href="#">band</a>	<a href="#">basis</a>	<a href="#">bezout</a>	<a href="#">blockmatrix</a>	<a href="#">charmat</a>
<a href="#">charpoly</a>	<a href="#">cholesky</a>	<a href="#">col</a>	<a href="#">coldim</a>	<a href="#">colspace</a>
<a href="#">colspan</a>	<a href="#">companion</a>	<a href="#">cond</a>	<a href="#">copyinto</a>	<a href="#">crossprod</a>
<a href="#">curl</a>	<a href="#">definite</a>	<a href="#">delcols</a>	<a href="#">delrows</a>	<a href="#">det</a>
<a href="#">diag</a>	<a href="#">diverge</a>	<a href="#">dotprod</a>	<a href="#">eigenvalues</a>	<a href="#">eigenvectors</a>
<a href="#">entermatrix</a>	<a href="#">equal</a>	<a href="#">exponential</a>	<a href="#">extend</a>	<a href="#">ffgausselim</a>
<a href="#">fibonacci</a>	<a href="#">forwardsub</a>	<a href="#">frobenius</a>	<a href="#">gausselim</a>	<a href="#">gaussjord</a>
<a href="#">geneqns</a>	<a href="#">genmatrix</a>	<a href="#">grad</a>	<a href="#">hadamard</a>	<a href="#">hermite</a>
<a href="#">hessian</a>	<a href="#">hilbert</a>	<a href="#">htranspose</a>	<a href="#">ihermite</a>	<a href="#">indexfunc</a>
<a href="#">innerprod</a>	<a href="#">intbasis</a>	<a href="#">inverse</a>	<a href="#">ismith</a>	<a href="#">issimilar</a>
<a href="#">iszero</a>	<a href="#">jacobian</a>	<a href="#">jordan</a>	<a href="#">kernel</a>	<a href="#">laplacian</a>
<a href="#">leastsqrs</a>	<a href="#">linsolve</a>	<a href="#">matadd</a>	<a href="#">matrix</a>	<a href="#">minor</a>
<a href="#">minpoly</a>	<a href="#">mulcol</a>	<a href="#">multiply</a>	<a href="#">norm</a>	<a href="#">normalize</a>
<a href="#">orthog</a>	<a href="#">permanent</a>	<a href="#">pivot</a>	<a href="#">potential</a>	<a href="#">randmatrix</a>
<a href="#">randvector</a>	<a href="#">rank</a>	<a href="#">references</a>	<a href="#">row</a>	<a href="#">rowdim</a>
<a href="#">rowspan</a>	<a href="#">rowspan</a>	<a href="#">scalarmul</a>	<a href="#">singularvals</a>	<a href="#">smith</a>
<a href="#">stackmatrix</a>	<a href="#">submatrix</a>	<a href="#">subvector</a>	<a href="#">sumbasis</a>	<a href="#">swapcol</a>
<a href="#">swaprow</a>	<a href="#">sylveste</a>	<a href="#">toeplitz</a>	<a href="#">trace</a>	<a href="#">transpose</a>
<a href="#">vandermonde</a>	<a href="#">vecpotent</a>	<a href="#">vectdim</a>	<a href="#">vector</a>	<a href="#">wronskian</a>

---

## grad transpose

This linalg routine computes the gradient of a function and returns the result as a Maple vector.

```
with(linalg):
arning, new definition for norm
arning, new definition for trace
g := grad(x^2 + 3*y^3 - 2*z, [x,y,x]);
                                     g=[2 x, 9 y^2, 2 x]
type(g, 'table');type(g, 'array'); type(g, 'vector'); type(g, 'matrix');
                                     true
                                     true
                                     true
                                     false
```

As noted above, the result could be converted to a "matrix vector" of either type as follows:

```
G := convert(g, matrix);
                                     -
                                     G :=  $\begin{bmatrix} 2x \\ 9y^2 \\ 2x \end{bmatrix}$ 
type(G, 'table');type(G, 'array'); type(G, 'vector'); type(G, 'matrix');
                                     true
                                     true
                                     false
                                     true

GT := transpose(G);
                                     GT=[2 x 9 y^2 2 x]
```

In general of course transpose returns the transpose of any matrix.



---

**row col transpose inverse det diag ....**

Here we simply illustrate the above list of commands:

```
> with(linalg):  
Warning, new definition for norm  
Warning, new definition for trace  
> A := matrix([[a,b],[c,d]]);  
A :=  $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$   
> row(A,1); type(%,'vector');  
[a,b]  
true  
> col(A,1);type(%,'vector');  
[a,c]  
true  
> AT := transpose(A);  
AT :=  $\begin{bmatrix} a & c \\ b & d \end{bmatrix}$   
> AI := inverse(A);  
AI :=  $\begin{bmatrix} \frac{d}{ad-bc} & -\frac{b}{ad-bc} \\ -\frac{c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}$   
> evalm(A^2 + AT);  
>  
 $\begin{bmatrix} a^2+bc+a & ab+bd+c \\ ca+dc+b & bc+d^2+d \end{bmatrix}$   
> det(A);  
ad-bc  
> diag(a,b,c);  
 $\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$ 
```

## Speed of matrix inversion.

Here is an inversion example where P are Legendre polynomials [need with(orthopoly) ]

```
> GG := (n,m) -> (2*m-1)^(1/2)*int(P(m-1,1-2*x)*P(n-1,x),x=0..1);
```

$$GG := (n, m) \rightarrow \sqrt{2m-1} \int_0^1 P(m-1, 1-2x) P(n-1, x) dx$$

```
> G := matrix(13,13,GG);
```

G =

$$\begin{bmatrix} 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \\ \left[ \frac{1}{2}, -\frac{1}{6}\sqrt{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \\ \left[ 0, -\frac{1}{4}\sqrt{3}, \frac{1}{20}\sqrt{5}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \\ \left[ -\frac{1}{8}, -\frac{1}{8}\sqrt{3}, \frac{1}{8}\sqrt{5}, -\frac{1}{56}\sqrt{7}, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \end{bmatrix}$$

```
> GI := inverse(G);
```

GI =

$$\begin{bmatrix} 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \\ \left[ \sqrt{3}, -2\sqrt{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \\ \left[ 3\sqrt{5}, -6\sqrt{5}, 4\sqrt{5}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \\ \left[ 11\sqrt{7}, -24\sqrt{7}, 20\sqrt{7}, -8\sqrt{7}, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \end{bmatrix}$$

where I have shown only the first four rows of the matrix so produced (and its inverse)

Inverting the 13x13 matrix as above takes about 14 seconds on my PC (Athlon 1600). If I try doing a 20x20 inversion, it takes more than 30 minutes (maybe a lot more). I think the main reason is that this inversion is being performed in a "symbolic" fashion rather than a "numeric" fashion. This is indicated by the "exact" numbers once sees in both G and GI above, such as  $11\sqrt{7}$ . If we apply evalf on the initial matrix, then the 20x20 inversion is instantaneous. Here is that code:

```
> G := evalf(matrix(20,20,GG));
```

G =

$$\begin{bmatrix} 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \\ \left[ .5000000000, -.2886751347, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \\ \left[ 0, -.4330127020, .1118033989, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \\ \left[ -.1250000000, -.2165063510, .2795084973, -.04724555913, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \right] \end{bmatrix}$$

```
> GI := inverse(G);|
GI:=
[1.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.,0.]
[1.732050804, -3.464101600, -.5734194478 10-8, .8665052543 10-8, -.9715994820 10-8, .7365303046 10-8,
-.1683999639 10-8, .1335806672 10-8, -.9208183973 10-9, .1670785577 10-8, -.1259638923 10-8, .1972822049 10-9,
-.2271624786 10-11, .5057444415 10-11, -.9354873543 10-12, .6364836359 10-13, .1225634440 10-12, .3665653462 10-14
, .2228291631 10-15, -.3625849288 10-16]
```

where here I show the first four rows of G but only the first two rows of GI. If we increase Digits from the default 10 to 20, the above operations are nearly as fast. Note that numerical calculations are always done in *software* floating point [ unless one somehow uses the evalhf hardware FP call, which does not work in the above case ]. Digits is just a variable which can be set to any positive integer, see elsewhere.

### linear algebra using matrices      eigenvectors    eigenvalues    linsolve

Suppose you want the **eigenvalues** and **eigenvectors** of a matrix. Easy as pie:

```
A := matrix(3,3, [1,-3,3,3,-5,3,6,-6,4]);
```

$$A := \begin{bmatrix} 1 & -3 & 3 \\ 3 & -5 & 3 \\ 6 & -6 & 4 \end{bmatrix}$$

```
e := eigenvalues(A);
```

$$e = 4, -2, -2$$

```
v := [eigenvectors(A)];
```

$$v = [[-2, 2, \{[1, 1, 0], [-1, 0, 1]\}], [4, 1, \{[1, 1, 2]\}]]$$

See the list discussion above for how to access elements of this nested list result! Actually, the list of eigenvectors is a "set", so be careful. The eigenvectors are not normalized.

Suppose you want to **solve**  $Ax = b$  for the vector x, given A and b. Here it is:

```
A := matrix( [[1,2], [1,3]] ):
b := vector( [1,-2] ):
linsolve(A, b);
```

$$[7, -3]$$

Notice that b is a Maple vector, and the result is a Maple vector. Similarly, you can solve  $AX=B$  for the matrix X. If the solution has a free parameter, Maple makes up a name for it, like t1.

---

## PLOTTING

---

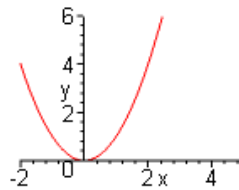
Maple is very heavy duty in this area (so is MATLAB). It has an underlying system of building a display list and then processing the items on that list for display, the way any good CG system works. However, there are many canned plotting routines, below are some examples.

Sometimes plots come out very small. You can crudely scale them up using the menu item **View/Zoom** factor which also has keyboard shortcuts like C-2 for 100%. This scales everything in the worksheet. Or just drag a corner of the plot to make it larger. Most of these plots require `with(plots)` to be executed before the plot call is used.

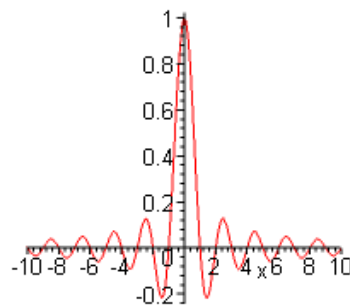
The 3D plots can all be rotated in real time using the mouse. Different kinds of plots have different options, which are always referenced at the bottom of the plot help window.

**plot** is the most basic 2D routine, you can specify the two ranges (hor and vert) or leave either be some default. (  $\text{sinc}(x) = \sin(x)/x$  )

```
> plot(x^2, x=-2..5, y=0..6);  
>
```

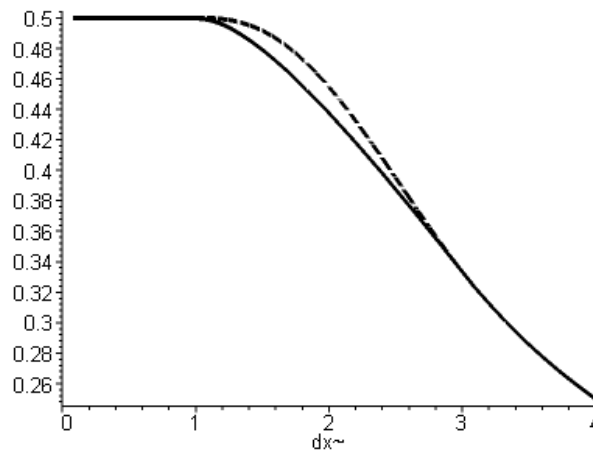


```
> plot(sinc(x), x);
```



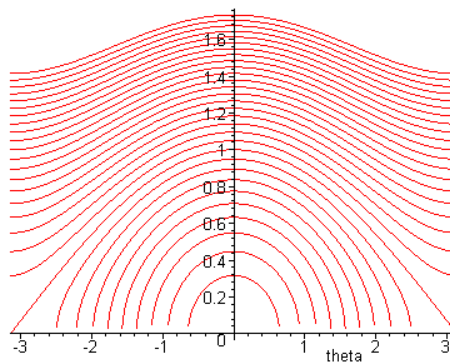
Here is an example showing how to do **multiple plots** and adjust appearances (j and gcos are function names, dx is the variable of the two functions):

```
plot([j, gcos],dx=0..4, color=black, thickness=3, linestyle=[1,3]);
```



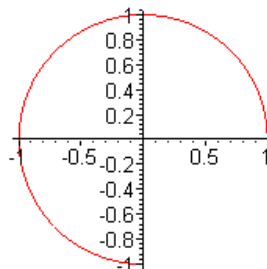
Here we use this same multiple plots idea to show a piece of the phase space of a pendulum. Note use of the seq statement mentioned above to create a "list" [...] of functions to plot. By default, Maple tries to make all the functions be different colors, so we put a stop to that. The plots were a little ragged until the number of points for the plot was raised to 1000

```
> plot([seq((E/10 - sin(theta/2)^2)^(1/2), E = 1..30)], theta = -Pi..Pi, color = [seq(red, E = 1..30)], numpoints=1000);
```



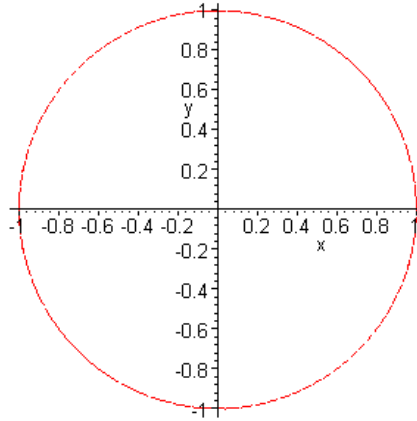
A **2D parametric plot** is done this way:

```
x := cos(t):
y := sin(t):
plot([x,y,t=0..1.5*Pi], scaling=CONSTRAINED );
```



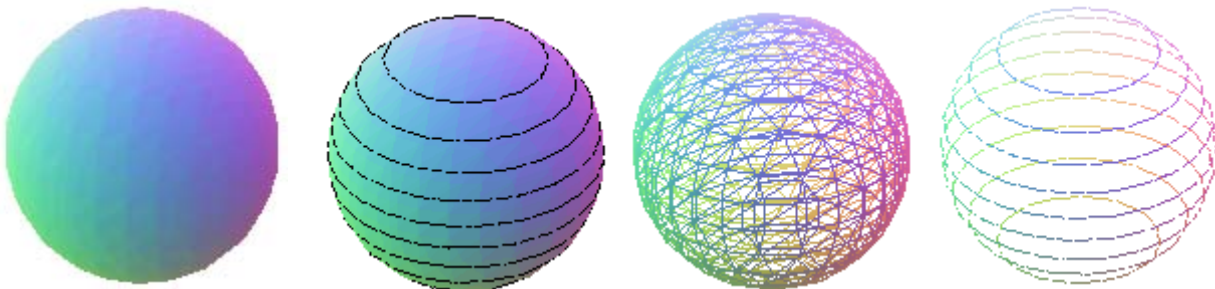
A **2D implicit plot** scans the parameter space and paints a dot at points where the stated equation is true within some graphic epsilon. For example (see below for a 3D implicit plot example)

```
with(plots):  
implicitplot(x^2 + y^2 = 1,x=-1..1,y=-1..1, scaling=CONSTRAINED);
```



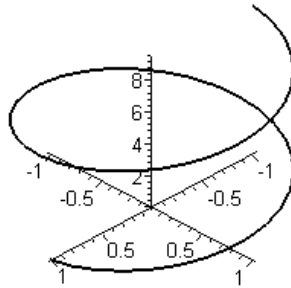
A **3D implicit plot** does the same thing with a 3D space scan, and constructs a 3D surface according to the selected options :

```
implicitplot3d(x^2+y^2+z^2 = 1, x=-1..1,y=-1..1,a=-1..1,scaling=CONSTRAINED);  
implicitplot3d(x^2 + y^2 + z^2= 1,x=-1..1,y=-1..1,z=-1..1, scaling=CONSTRAINED);
```



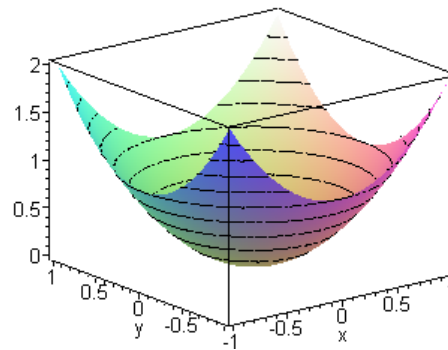
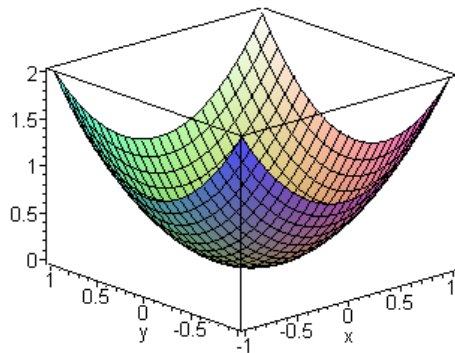
**spacecurve:** A 3D parametric plot is done with a different function call,

```
x := cos(t): y := sin(t): z := t:
with(plots):
spacecurve([x,y,z],t=0..2*Pi,axes=NORMAL,thickness=2,color=BLACK);
```



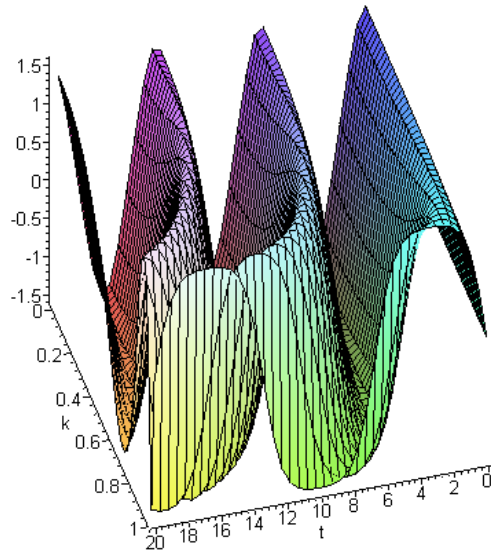
**plot3d**( $x^2 + y^2$ ,  $x=-1..1, y=-1..1$ ) gives you a wonderful 3D plot of this surface, and you can grab the surface and rotate it "in real time" as you like (as with all 3D plots). The default mesh is 20x20 and color is used to make the display clear. Here is a double parabola whose horizontal slices are circles.

```
plot3d(x^2 + y^2, x=-1..1,y=-1..1, axes=BOXED);
plot3d(x^2 + y^2, x=-1..1,y=-1..1, axes=BOXED, style=PATCHCONTOUR);
```



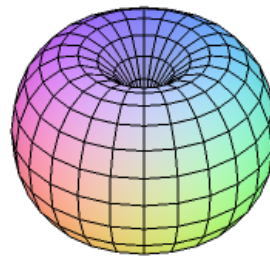
Here is another example which shows how a pendulum starts out doing sine waves for small oscillations, (in the back at  $k=0$ ) but as you increase the starting angle ( $k = \sin^2 \theta_0/2$ ), they distort into Jacobi functions  $sn(x,k)$  and the period increases. ( $\theta_0 = \text{Pi}/2$ )

```
> plot3d(2*arcsin( sin(theta0/2) * JacobiSN(t,k)), t = 0..20, k=0..0.99, grid = [40,40]);
```



To plot in **spherical coordinates**, the angles can have arbitrary names, but the azimuth must appear first:

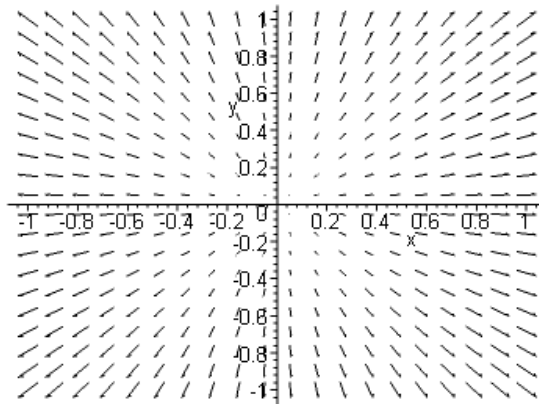
```
plot3d(sin(theta)^2, phi = 0..2*Pi, theta = 0..Pi, coords=spherical);
```



**gradplot** gives an array of little arrows, showing you the size and direction of the gradient at each point in the range you specify.

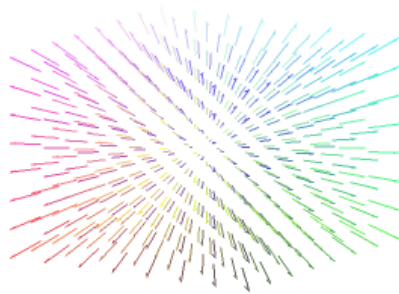


```
> with(plots): gradplot(x^2 + y^2, x=-1..1, y=-1..1);
>
```



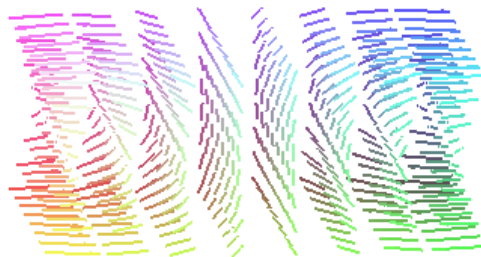
**gradplot3d** is the 3D version of the same thing, here is an example

```
> gradplot3d(x^2 + y^2 + z^2, x=-1..1, y=-1..1, z=-1..1);
```



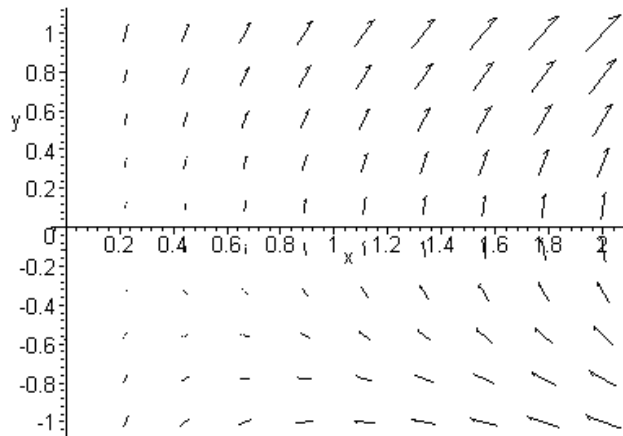
**fieldplot3D** can be used to plot a vector field in 3D space.

```
with(plots):
u1 := x3*(x1-1):
u2 := -0.3*x2*x3:
u3 := (1/2)*x1*(2-x1) + (1/2)*0.3*(x2^2-x3^2):
fieldplot3d([u1,u2,u3],x1=0..2, x2=-1..1, x3=-1..1,thickness=2);
```



**fieldplot** plots a 2D vector field in 2D space:

```
fieldplot([x*y,x+y],x=0..2, y=-1..1,grid=[10,10]);
```

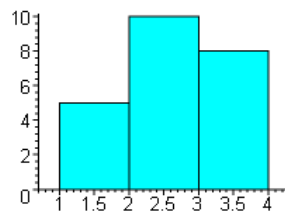


There are many, many more plotting methods, I will add them here when I use them some day.

**logplot** puts log only on the vertical axis  
**loglogplot** puts log on both axes  
**semilogplot** puts log only on the horizontal axis

**histogram** is ugly, in order to get a bar chart you have to put in the width of each bar as the first argument in a Weight function:

```
> with(stats):  
with(stats[statplots]):  
data1:=[ Weight(1..2, 5), Weight(2..3, 10), Weight(3..4, 8)]:  
histogram(data1, color=cyan);
```

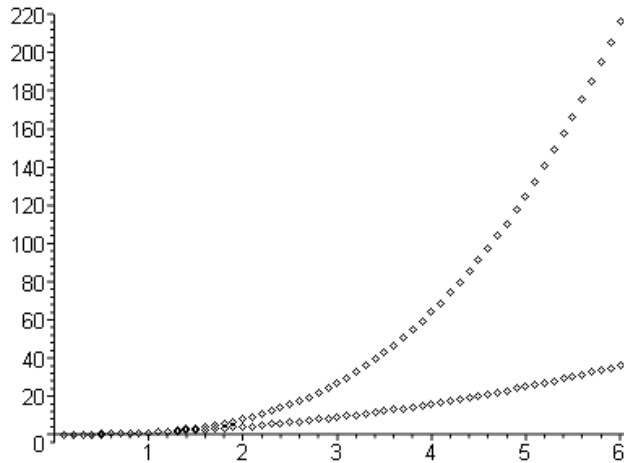


**pointplot** requires a list or set of x,y coordinates. It makes what Excel calls a "scatter plot". Here is an example that illustrates several things at once:

```

restart;
with(plots):
p := seq([i/10, (i/10)^2], i=1..60):
q := seq([i/10, (i/10)^3], i=1..60):
pointplot({p,q});

```



First we generate two sequences  $p$  and  $q$ , each of which is intended to be a separate "curve" on our graph. These curves are  $x^2$  and  $x^3$ . The coordinate ordering is  $[x,y]$  as you would expect. Inside the `pointplot` call, we first concatenate  $p$  and  $q$  into a single sequence, then we turn that sequence into a set (`pointplot` needs a single set or a list as argument) and we then get our scatter plot. Another technique illustrated is scaling of the coordinates like  $i/10$ . Clearly the index  $i$  can take only integer values, so by scaling in this way we can get more points on the curves.

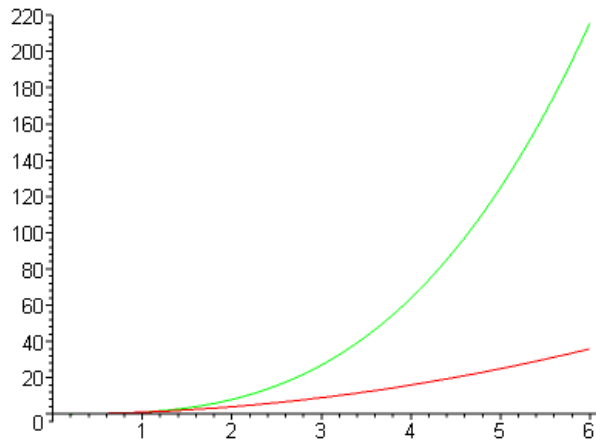
You can add the option `style=LINE` and it will connect each adjacent pair of points with a line instead of putting down markers, and example being `pointplot([q], style=LINE)`. In this case, you really want to have a list, not a set, since it might treat the points of a set in some random order, then you get a mess! However, if we try this option in our example above, it will connect the last point of the first curve to the first of the second, adding a spurious line to our nice graph.

The `pointplot` call cannot plot multiple curves except in the scatter sense above. The argument must be a single list of points. If you want to have multiple curves from multiple lists of points, you can do it this way, using the regular `plot` call which knows about point lists,

```

p := seq([i/10, (i/10)^2], i=1..60):
q := seq([i/10, (i/10)^3], i=1..60):
plot({ [q], [p] });

```

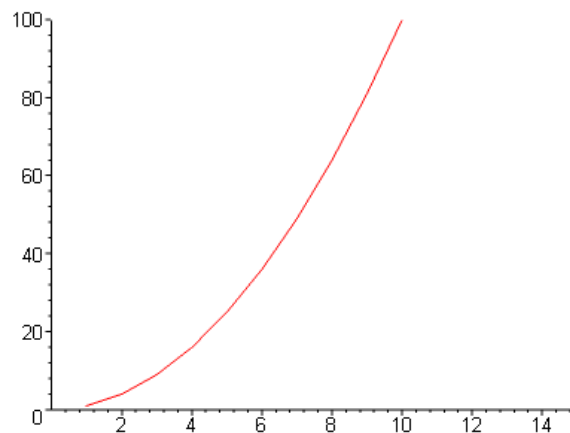


**listplot** is similar to **pointplot**, but your list contains only y coordinates for your points, and the x coordinates are assumed to be incrementing integers starting with 1. The points are not marked, but are connected by lines. Here is a simple example

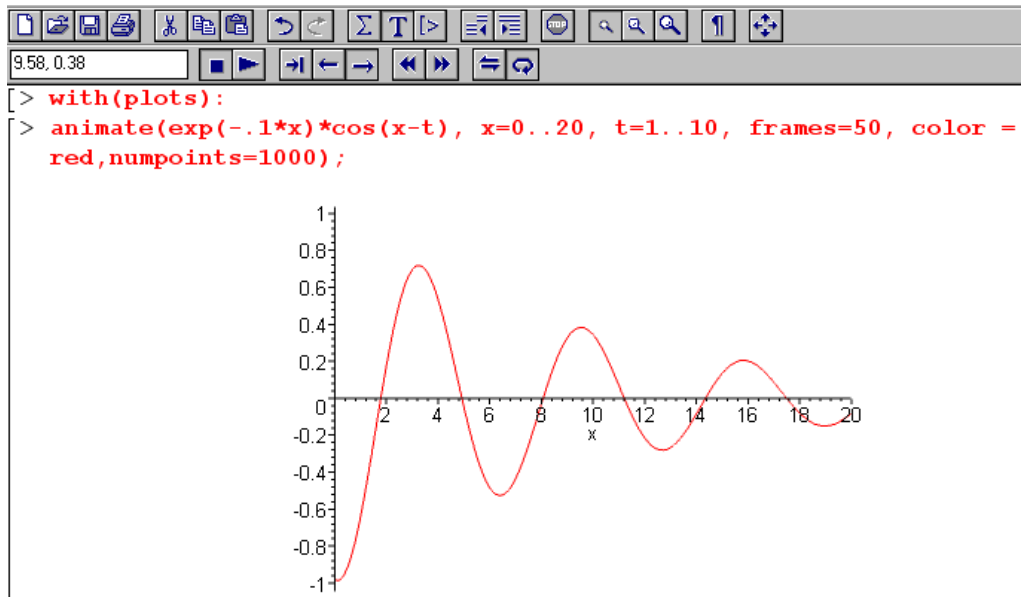
```

listplot([seq(i^2, i=1..10)], color=red, view=[0..15, 0..100]);

```



**animate** creates an animated 2D plot. First, you execute the **animate** call to build the display database which might take a while, after which the first frame only is displayed. If you then click in the plot, some "media player" icons appear and you "play" the animation using the usual play icon, and another icon sets the animation to loop forever. Multiple animations can be run at once by starting them one at a time and making each one loop. It is all very excellent. Here is an example with the plot selected and the player icons showing.

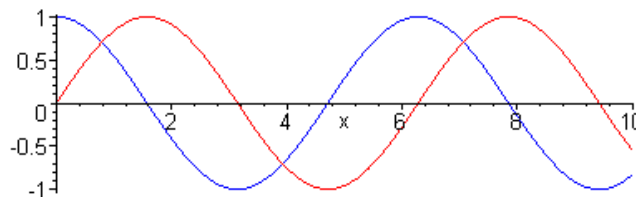


If you click on a point in the graph area, the (x,y) coordinates of that point show up in the white window. Here I have just clicked on the top of the second hump.

### Overlaying multiple plots: lists, PLOT, display() and gridlines

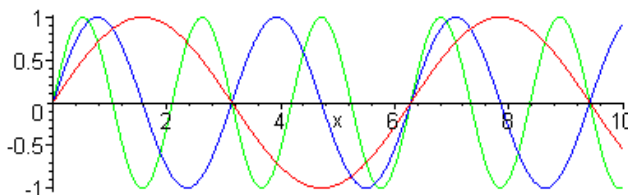
1. To overlay multiple plots where it is OK for all plots to have the same plot options (such as thickness), you can arrange to have the first argument be a *list* of functions like this (see earlier on Maple lists):

```
plot([sin(x),cos(x)], x = 0..10, color = [red,blue]);
```



or like this

```
plot([seq(sin(N*x), N=1..3)], x = 0..10, color = [red,blue,green]);
```



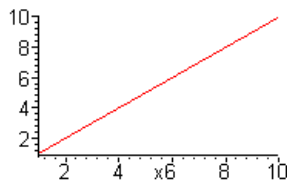
We now build up to a second method of overlaying plots which is more general.

2. When assigned to a plot command, a variable is a "display list" containing displayable data of some sort. This display list in Maple is called a PLOT structure. Once you have a display list, you can plot it using the display command. For example:

```

> p1 := plot(x, x = 1..10, color = red, numpoints = 10);
p1 = PLOT(CURVES([[1., 1.], [1.523131700000000, 1.523131700000000],
[2.046263400000000, 2.046263400000000], [2.501437495000000, 2.501437495000000],
[2.956611590000000, 2.956611590000000], [3.468502465000000, 3.468502465000000],
[3.980393340000000, 3.980393340000000], [4.495677360000000, 4.495677360000000],
[5.010961380000000, 5.010961380000000], [5.523796345000000, 5.523796345000000],
[6.036631310000000, 6.036631310000000], [6.512094970000000, 6.512094970000000],
[6.987558630000000, 6.987558630000000], [7.479873710000000, 7.479873710000000],
[7.972188790000000, 7.972188790000000], [8.481341750000000, 8.481341750000000],
[8.990494710000000, 8.990494710000000], [9.495247355000000, 9.495247355000000], [10., 10.]]),
COLOUR(RGB, 1.00000000, 0, 0), AXESLABELS("x", ), VIEW(1..10., DEFAULT))
> display(p1);

```

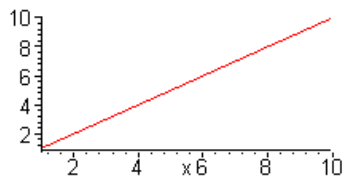


This produces the same graph as the plot command all by itself with no assignment,

```

plot(x, x = 1..10, color = red, numpoints = 10);

```



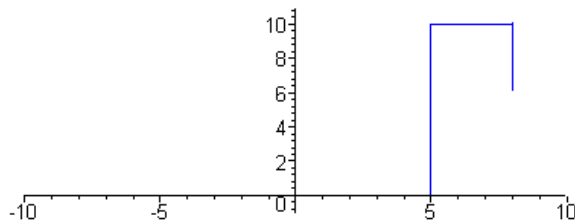
Comment: The plot command takes numpoints as a suggestion only. In the example above it actually made 17 points. For a curved plot, there will likely be a large number of points. For that reason it is good to end a `p1 := plot..` command with a colon to prevent display of perhaps 100 or more points.

3. You can manually create a display list using the PLOT structure directly and then display it,

```

p2 := PLOT(CURVES([[5, 0], [5, 10], [8, 10], [8, 6]]), COLOR(RGB, 0, 0, 1));
p2 = PLOT(CURVES([[5, 0], [5, 10], [8, 10], [8, 6]]), COLOR(RGB, 0, 0, 1))
display(p2, view = [-10..10, -1..11]);

```

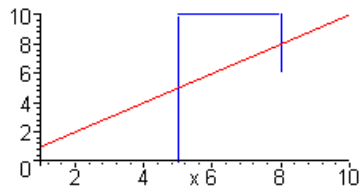


4. Although the two display lists `p1` and `p2` were generated by different methods, they can be displayed at the same time to overlay the two plots :

```

p1 := plot(x, x = 1..10, color = red, numpoints = 10):
p2 := PLOT(CURVES([[5, 0], [5, 10], [8,10],[8,6]]), COLOR(0,0,1)):
display(p1,p2);

```



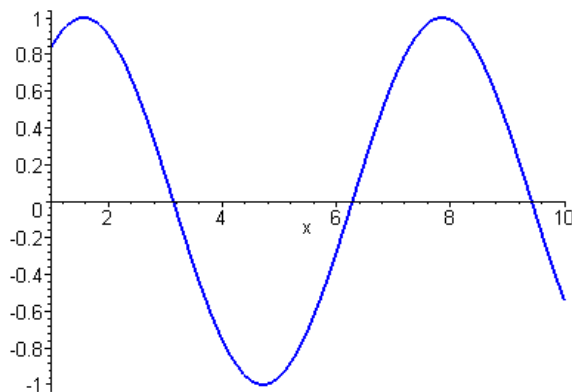
Note that the command "plot" and the structure "PLOT" are different object. You can overlay any number of plots this way using the display command, which itself has various options.

5. Application: Here we plot a sine curve to which we add vertical and horizontal grid lines. Current Maple has easy ways to do this, but in Maple V you had to roll your own. This serves as a good illustration and shows how highly customized graphs can be made. A display command is added on the right just so we can see what we have.

```

restart: with(plots):
p1 := plot(sin(x), x = 1..10, color=blue, thickness = 2):display(p1);

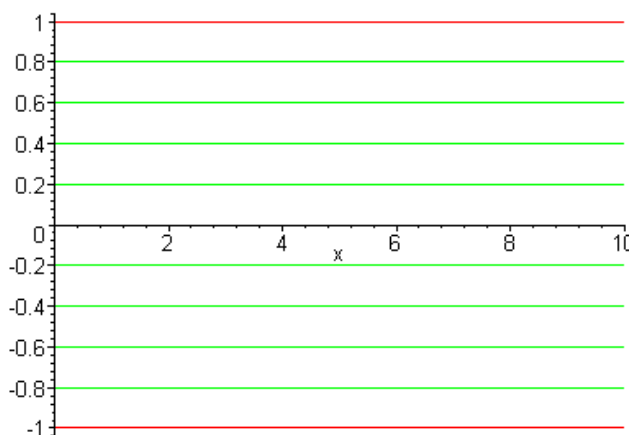
```



```

p2 := plot([seq(N/5,N=-5..5)],x=0..10, color = [red, green$4]):display(p2);

```

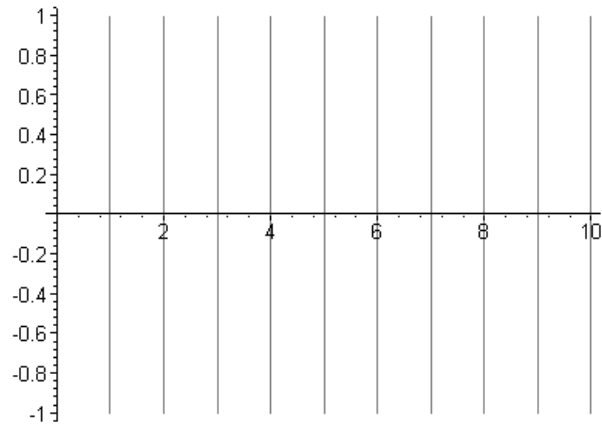


Note here the use of the \$ operator mentioned earlier to repeat the green color 4 times. After that, the color sequence is repeated as needed. Making vertical lines cannot be done by the above method, but it can be done this way

```
with(plots):p3 := PLOT(seq(CURVES([[N,-1],[N,1]]),N=0..10),COLOR(RGB,.5,.5,.5)):display(p3);
```

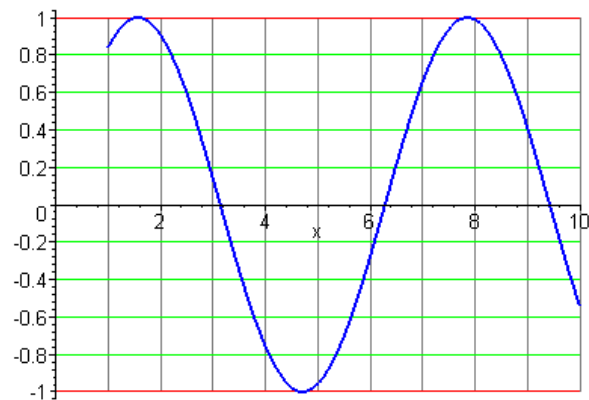
```
with(plots):
```

```
p3 := PLOT(seq(CURVES([ [N, -1] , [N, 1] ]), N=0..10), COLOR( RGB , . 5 , . 5 , . 5 )) : display(p3) ;
```



Then the final graph is this,

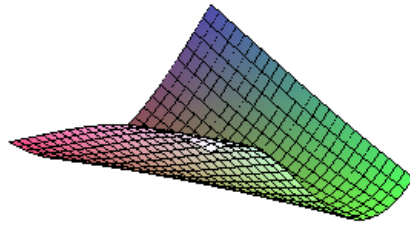
```
display(p1,p2,p3) ;
```



6. This concept can be extended to 3D plots as well.



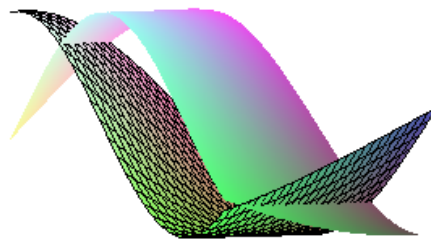
```
restart;with(plots):  
p1 := plot3d( sin(x*y), x=1..2, y = 3..4):display(p1);
```



```
p2 := plot3d( cos(x*y), x=1..2, y = 3..4, style=PATCHNOGRID):display(p2);
```



```
display(p1,p2);
```



Maple is pretty smart about this stuff. Notice the hidden surface removal .

---

## PROCEDURES AND PROGRAMMING

---

### Procedures and Programming      Algol

The Help system has a man page on procedures, but it fails to say what kind of statements are allowed in the body of a procedure, and it gives no examples whatsoever (but they are on other help pages). I presume that ANY statements are allowed inside. See "hypergeometric" section below for an example!

The syntax reminds me of **Algol68** as I learned it circa 1968. Several things make me think this. First, if and do statements end with fi and od, which was done in Algol68. Second, assignments are done with := instead of just =, another Algol thing. Statements end in semicolons. Apart from these similarities, Maple was I think done from scratch in 1985. It combines both numeric and symbolic manipulation, perhaps the best of all worlds so you can avoid LISP :-). Maple was written in Maple mostly, always a good sign.

Here again is the above list of reserved words which suggests what programming constructs are available in Maple. The main acts are if, for and do, see code sample below.

```
and by do done elif
else end fi for from
if in intersect local minus
mod not od option options
or proc quit read save
stop then to union while
```

---

### The for/while/do statements      for do od from to by while

Summary: For or while can each be combined with do, or all three can appear together. The loop index need not be an integer. Use the print statement to figure out what a loop does. These "iteration statements" can be used at top level in Maple or in a procedure.

We shall now consider a series of examples, each of which is a tiny case study. Consider:

```

n := 14;
for n from 1 to 3 do
  m := n+2;
  a[n] := m^3;
od;

```

```

n := 14
m := 3
a1 := 27
m := 4
a2 := 64
m := 5
a3 := 125

```

```

m;
n;

```

Notice there can be multiple statements inside the do...od bracketing. Also, all variables are global in nature, as shown in our extra statements. The n value got incremented to 4, was then out of range so the loop exited. The last time the do code ran was with n = 3 so m is left at 5. There are no "local variables" here as there are in a procedure (see below).

Here is a for statement with more bells and whistles, and this line shows how you can deduce what the loop really looks like in terms of where the test is done by using print(n) :

```

for n from 10 to 2 by -3 do print(n) od;

```

```

10
7
4

```

Things like the 10 and 2 and -3 can all be **expressions**, though we don't show that in our examples. Another option is to add a **while** (something) clause as follows:

```

for n from 1 to 10 while n < 4 do print(n) od;

```

```

1
2
3

```

By default, the range for n starts at 1 and goes up by 1, so you could do this

```

for n while n<4 do print(n); od;

```

```

1
2
3

```

but not recommended by me. Here is a "while loop" all on its own, without a "for"

```

while abs(a)<2 do a := rand(-5..5)(); od;

```

which produces in "a" a random integer as one of these values: ±3,±4,±5.

This loop computes the sum of squares of integers,

```
s := 0; for n from 1 to 4 do s := s + n^2 od;
s := 0
30
```

By ending with a colon, we suppress intermediate output lines. Here is a simpler alternative to the above:

```
add(n^2, n=1..4);
30
```

Many Maple commands have implicit "for loops" like the above.

The loop index does not have to be an integer,

```
for n from 0.2 to 1.0 by 0.2 do print(n) od;
.2
.4
.6
.8
1.0
```

This pair of examples shows the only other "for" syntax

```
> joe := 10,20,30,40,50,60;
joe := 10, 20, 30, 40, 50, 60
> for n in joe while n < 45 do print(n) od;
10
20
30
40
> for n in joe[2..6] while n < 45 do print(n) od;
20
30
40
```

Here joe is a sequence, and n gets set to each element of the sequence from left to right. In the second example, we select a piece of the full joe sequence and do the same thing, with different output.

You can **break** out of a loop like this

```
for n from 1 to 4 do
  print(n, "before");
  if (n>2) then break fi;
  print(n, "after");
od;

1, "before"
1, "after"
2, "before"
2, "after"
3, "before"
```

Notice the use of humanized print statements for quick debugging of loop code, as in any language. If you have nested do loops, **break** jumps out of all of them (I think), while **next** only breaks from the loop in which it appears and continues in the next higher loop. Experiment to verify!

**Nested loops** are allowed to any level, but if you want to see the execution of statements beyond the first level, you must adjust the printlevel variable (which defaults to 1). In this example, white space emulates "real code" (using shift-enter to get each new line):

```
[> printlevel := 2:
[> for n from 1 to 2 do
  for m from 1 to 2 do
    n^2*m^3;
  od;
od;

1
8
4
32
```

For more detail, look in the Help system under Programming/Flow Control/Iteration or Looping.

---

### The if statement                    if then and or else elif fi

First, the simplest possible syntax, the code is I think self explanatory.

```
a := 4: b := 7:
if (a>b) then print(a) fi;
if (a<b) then print(a) fi;
```

4

As a next example consider this code

```
[> a := 4: b := 7: c := 10:
> if ((a<b and c>b) or (c<a)) then (a-c)^3 else (b-c) fi;
-216
> %;
-216
> d := if ((a<b and c>b) or (c<a)) then (a-c)^3 else (b-c) fi;
reserved word `if` unexpected
> if ((a<b and c>b) or (c<a)) then d := (a-c)^3 else d :=(b-c) fi;
d := -216
```

The first if puts -216 into the "last thing computed basket" called %, the ditto operator. The if command does not return anything (just like the for command), so the second if statement fails. This is then rectified by the third if statement.

Here is the usual "else if" construction:

```
[> if (a>b) then print(a,b,"first line")
elif (b>c) then print(b,c,"second line")
elif (a>c) then print(a,c,"third line")
else print(a,b,c, "fourth line") fi;
4, 7, 10, "fourth line"
```

See Programming/Flow Control/ if .

Here is a rather startling result:

```
> restart;
> if (Pi < 10) then a := 1 else a := 2 fi;
Error, cannot evaluate boolean
>
> if (evalf(Pi) < 10) then a := 1 else a := 2 fi;
a := 1
```

Maple sees the Boolean expression  $\text{Pi} - 10$ , prints it as  $\pi - 10$ , but this does not become a number until it is evaluated. Thus, Maple cannot determine that  $(\text{Pi} < 10)$  is true, same is for  $(\text{bob} < 10)$ .

The operators used in Maple boolean expressions are :  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $=$ ,  $<>$ , and, not, or .

**Example 1:**

```

[ > cube := proc(x)
  x^3;
  end;
                                     cube := proc(x) x^3 end
[ > a := cube(4);
                                     a := 64
[ > a;
                                     64

```

Remember how you use the shift-enter to get multiple lines in a command!

This shows how things start with `proc(argument list)` and end with "end". In this case we pass the procedure a single argument and it computes the cube of that argument. Since this is the last expression to be computed, this is what gets returned when the call returns, so `a` ends up being 64 which is  $4^3$ . If you want to return something other than the last calculated item, use the command **RETURN(x)** where `x` is the quantity you want to be returned.

This procedure is what other languages might call a "function" of a single variable.

With our understanding of white space, we know we could have written this thing more compactly on one line, for example

```

[ > quad := proc(x) x^4; end;
                                     quad := proc(x) x^4 end
[ > quad(2);
                                     16

```

The semicolon just before the end is not required, but here we keep it anyway.

**Example 2:**

```

[ > f := proc(x) a := x+2; a^2 end;
Warning, `a` is implicitly declared local
                                     f := proc(x) local a; a := x + 2; a^2 end
[ > f(3);
                                     25

```

This procedure contains two statements instead of one, the main new feature. As a command line thing, this entire procedure definition is a single statement, but the *body* of the procedure contains two statements. The second feature is that we have used what we intend to be a temporary variable `a`, but in our bad form we have not made this clear to Maple so it tells us its interpretation. Better code:

```
f := proc(x) local a; a := x+2; a^2 end;
f:=proc(x) local a; a := x + 2; a^2 end
```

where "local" is a keyword having just this meaning in a procedure. There is another keyword "global" that means the variable so specified is one that exists at the top level, outside the procedure, and we certainly would not want to do that here.

### Example 3:

```
> f := proc(x,y)
  local a,b;
  a := x^2 + y;
  b := x - y^2;
  if (a < b) then a else b fi;
end;
>
f:=proc(x,y) local a,b; a := x^2 + y; b := x - y^2; if a < b then a else b fi end
> f(2,3);
-7
```

(1) Now we have two local variables, and more code lines including an "if" statement. (2) The parens around the Boolean test expression( here  $a < b$ ) are not required as they are in other languages. (3) The function as usual returns the last thing evaluated, and that is going to be either a or b from the if statement. (4) Notice that we don't see any display of a or b from statements inside the procedure which we know ran when we did f(2,3), even though they end with semicolons. This is the way things are with procedures. If you WANT to see something, you have to throw in a print statement:

```
> f := proc(x,y)
  local a,b;
  a := x^2 + y;
  b := x - y^2;
  print(a,b);
  if (a < b) then a else b fi;
end:
>
> f(2,3);
7,-7
-7
```

Notice that by ending the procedure definition with a colon, we have stopped it from echoing its interpretation of the procedure in blue.



#### Example 4: Showing a mysterious problem and how to fix it: single quotes and deferral

Consider:

```
> sin1 := proc(x)
    if (x<0) then sin(x) else sin(x) fi;
end;
>
sin1 = proc(x) if x < 0 then sin(x) else sin(x) fi end
> sin1(.2);
.1986693308
> type(sin1(x), 'vector');
Error, (in sin1) cannot evaluate boolean
> plot(sin1(x), x=0..2);
Error, (in sin1) cannot evaluate boolean
```

Here we have what most would agree is a fairly elementary procedure. It works, but when it is used in the type statement or the plot statement, it does not work, which is very annoying. Here is what is going on. The plot statement fails because it calls a type statement to sort of "check out" what kind of argument the first plot argument is. The type statement fails (as shown explicitly above) for the following reason: it tries to evaluate  $\text{sin1}(x)$ , but  $x$  is unassigned, so when it tests to see if  $x < 0$  (boolean true or false),  $x$  is not even a number so it spits out the error you see. The fix is to have the procedure detect the nature of  $x$  before it does anything else, and if it is not a number, then we want to return  $\text{sin1}(x)$  in good condition so it can be evaluated later. Consider the following strange looking code

```
restart;
mysin := 'sin(x)';
mysin;
x := 0.2;
mysin;
mysin;
```

*mysin* := sin(*x*)  
*x* := .2  
.1986693308

Here, the string  $\text{sin}(x)$  is installed into the variable  $\text{mysin}$ . During this installation,  $\text{sin}(x)$  is not evaluated, the evaluation is *deferred* till "the next time". At the time  $\text{mysin}$  is assigned,  $x$  is unassigned and not a number, but we don't get any complaint because of this deferral. We then assign  $x$  for the first time, and we do  $\text{mysin}$  and Maple sees  $\text{sin}(x)$  with  $x = .2$  and we get our proper answer. From Maple help on "uneval":

- After each evaluation, one level of single quotes is stripped off. So if  $a := 1$ ; then "a"; returns 'a', 'a'; returns a, and a; returns 1.

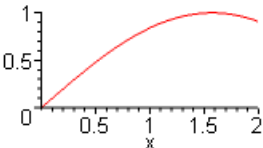
Hopefully this explains how Example 4 can be repaired. The repaired code is this:

```

> sin1 := proc(x)
  if type(x,numeric) then
    if (x<0) then sin(x) else sin(x) fi;
  else
    'sin1(x)';
  fi;
end;

sin1 := proc(x) if type(x, numeric) then if x < 0 then sin(x) else sin(x) fi else 'sin1(x)' fi end
> type('sin1(x)', 'numeric');
false
> plot(sin1(x), x=0..2);
>

```



All Maple built-in functions are coded this way and if you want your function to work *generally*, you should do the above. In practice, people probably don't bother to do that for little local procedures.

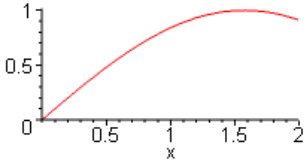
If we insist on the simple coding of our `sin1` routine, we can still make the typedef and plot routines work in the following manner:

```

> sin1 := proc(x)
  if (x<0) then sin(x) else sin(x) fi;
end;

sin1 := proc(x) if x < 0 then sin(x) else sin(x) fi end
> sin1(.2);
.1986693308
> type('sin1(x)', 'vector');
false
> plot('sin1(x)', x=0..2);
>

```



Notice now that the single quote are around `'sin(x)'` in the calls to `type` and `plot`, so the deferral is achieved at an earlier time. The first time `sin1(x)` is evaluated, it is not really evaluated. As noted above, the only downside to this method is that your `sin1(x)` routine is somehow "different" from all the other functions in Maple and needs this special treatment in certain places.

### Example 5: the permutation tensor

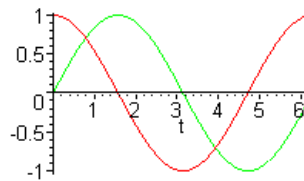
This code implements the permutation tensor in three dimensions and shows the multiple elif construction with a catch bag at the end for all unlisted cases. It also uses the protection mechanism described in the previous example, without which the sum shown at the end would come out being 0.

```
> eps := proc(a,b,c)
  if type(a,numeric) and type(b,numeric) and type(c,numeric) then
    if (a=1 and b=2 and c=3) then 1;
    elif (a=1 and b=3 and c=2) then -1;
    elif (a=2 and b=1 and c=3) then -1;
    elif (a=2 and b=3 and c=1) then 1;
    elif (a=3 and b=1 and c=2) then 1;
    elif (a=3 and b=2 and c=1) then -1;
    else 0;
    fi;
  else 'eps(a,b,c)';
  fi;
end:
> eps(2,1,3);
-1
> sum(sum(eps(a,b,3), a=1..3), b=1..1);
-1
```

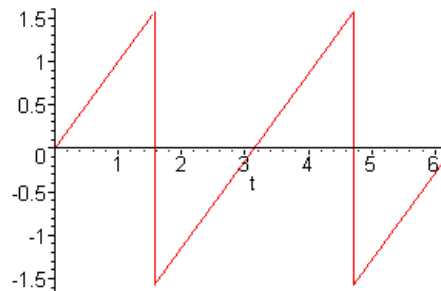
### Example 6: the arctan2Pi function

Suppose  $x$  and  $y$  are functions of  $t$  and you want to plot  $\varphi(t)$  where  $\tan\varphi = y/x$  and you want a result showing  $\varphi$  in the range  $(0,2\pi)$ . This does not work well with the Maple arctan function, as shown here

```
x := cos(t): y := sin(t):
plot([x,y], t = 0..2*Pi);
```



```
plot(arctan(y/x), t = 0..2*Pi);
```



One remedy is this custom arctangent procedure

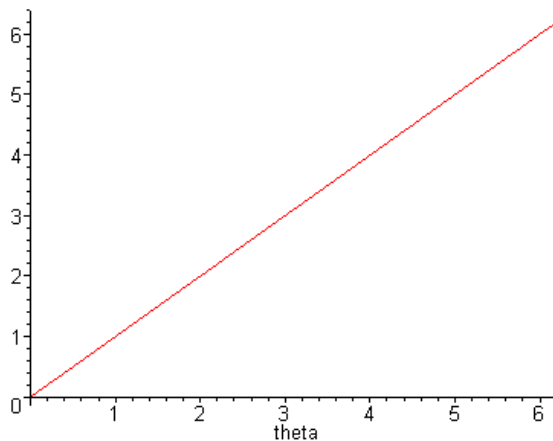
```

arctan2Pi := proc(x,y)
local q;
if type(x,numeric) and type(y,numeric) then
if x = 0 and y = 0 then print("arctan2Pi(0,0) error." ); RETURN(0) fi;
if x = 0 and y > 0 then RETURN(Pi/2) fi;
if x = 0 and y < 0 then RETURN(3*Pi/2) fi;
if x > 0 and y = 0 then RETURN(0) fi;
if x < 0 and y = 0 then RETURN(Pi) fi;
if x > 0 and y > 0 then q := 0 fi;
if x < 0 and y > 0 then q := Pi fi;
if x < 0 and y < 0 then q := Pi fi;
if x > 0 and y < 0 then q := 2*Pi fi;
RETURN(arctan(y/x)+q);
else
'arctan2Pi(x,y)';
fi;
end;

```

which among other things illustrates the fix for the problem in Example 4. We now get the desired result:

```
plot(arctan2Pi(x,y), theta=0..2*Pi);
```



### Example 7: One from the web

This is one brought in from the web (I lost the reference), just to show other people's style in writing procedures. It makes use of the limit and subs commands (see Help for limit). It also shows a nested do loop.

"The **hypergeom function** in Maple is the fully generalized one  ${}_pF_q$  where there are going to be  $p$  Pochhammer symbols in the top, and  $q$  in the bottom. We usually work with  ${}_2F_1$ . See page 556 of AS. In the Maple entry notation, you have `hypergeom([a1, a2 ... ], [b1, b2 ... ], z)` where the arguments are presented as usual as "lists". The  $p$  and  $q$  are determined by the length of these lists. "

```

=====
# Now for the associated Legendre function (of the first kind).
=====
#
# First, note that the above definition of assocLegendre1 gets into trouble

```

```

# when mu = 1 because the definition contains 1/GAMMA(1-mu).
# Moreover, whenever 1-mu is a negative integer we hit a singularity of GAMMA.
# In these cases, we understand the limit value is to be used.
#
# So redefine assocLegendre1 to use a limit value in those cases.
# Also, we can place the invocation of simplify into the definition.
assocLegendre1 := proc(nu,mu,z)
    local m,e;
    e := 1/GAMMA(1-m)*((z+1)/(z-1))^(m/2)*hypergeom([-nu, nu+1],[1-m],(1-z)/2);
    if type(mu-1, nonnegint) then
        simplify(limit(e, m=mu))
    else
        simplify(subs(m=mu, e))
    fi;
    factor(%);
end:
#
# Try it for some values of mu and nu.
#
#
for nu from 0 to 2 do
    for mu from nu by -1 to -nu do
        print('nu'=nu, 'mu'=mu);
        print(assocLegendre1(nu,mu,z));
        print(`=====`)
    od
od;

```

### Example 8: Looking at Maple's own code

Some code that comes with the Maple system can be viewed by the user. Suppose we want to see how Maple actually computes Legendre Q functions. The first step is to *use* the function in question, with the type of argument in question, and that causes the code to "load" in some internal storage bin. Once loaded, you can look at it with the **showstat** command. For example (notice that back tick marks)

```

> evalf(LegendreQ(1,2,0.2));
-2.083333333 - .7513644637 10-14 I
> showstat(`evalf/LegendreQ`);
`evalf/LegendreQ` := proc()
local v, u, z, v_is_int, u_is_int, v_is_half_int, u_is_half_int, s, d;
  1   if nargs < 2 or 3 < nargs then
  2     ERROR(`expecting 2 or 3 arguments, got ` .nargs)
    fi;
  3   v := evalf(args[1]);
  4   z := evalf(args[-1]);
  5   if nargs = 2 then
  6     if not type([v, z], `list(complex(numeric))`) then
  7       RETURN(`LegendreQ` (v, z))
    fi;
  8   u := 0
    else
  9     u := evalf(args[2]);
 10   if not type([v, u, z], `list(complex(numeric))`) then
 11     RETURN(`LegendreQ` (args))
    fi
  fi;

```

lines omitted here

```

33   if v_is_half_int then
34     d := Digits+5;
35     s := evalf(evalf(LegendreQ(v+10^(-d), u, z), Digits+8+ilog10(v)))
    elif u_is_int then
36     s := traperror(evalf(`LegendreQ/complex_int` (v, u, z)))
    else
37     s := traperror(evalf(`LegendreQ/complex_gen` (v, u, z)))
    fi
  fi;

```

I don't know how this all works, but you see here the start of a certain procedure whose job is to evaluate a Legendre  $Q_\nu^\mu(z)$  function to a floating point result. In the last lines above you see that it calls different lower level routines depending on whether  $\mu$  and  $\nu$  are integers or half integers, and whether  $z$  is complex or real, etc. We can then drill down to look at the next lower piece of code, for example if we force evaluation of a  $Q$  function with a "general complex  $z$ " argument. It has to load the following lower piece of code, so you can then look at it:

```

> evalf(LegendreQ(1.2,2.3,0.2 + 0.3*I));
-1.894082130 - 1.675867291 I
> showstat(`evalf/LegendreQ/complex_gen`);

`evalf/LegendreQ/complex_gen` := proc(v, u, z)
local oldDigits, mag_u, mag_v, mag_z, v_mags, u_mags, z_mags, predDigits, resDigits, lost, s;
  1   oldDigits := Digits;
  2   mag_v := ilog10(v);
  3   mag_u := ilog10(u);
  4   mag_z := ilog10(z);
  5   v_mags := max(readlib('cmagdiff')(v),0);
  6   u_mags := max(readlib('cmagdiff')(u),0);
  7   z_mags := max(readlib('cmagdiff')(z),0);
  8   predDigits := 2;
  9   Digits := Digits+max(0,mag_v,mag_u,mag_z);
 10   to 5 do
 11     Digits := Digits+predDigits+2;
 12     if abs(z-1) < 1.5 then

```

An interesting notion you see in practice here is dynamically messing around with the Digits parameter to cause computation to an appropriate accuracy.

There may be some easier way to look at Maple code, this is just something I stumbled onto while doing some debugging (where it has to show you the code as you step through it). The code provides many examples of professional Maple programming techniques.

The above code only shows how Maple evaluates  $Q_\nu^\mu(z)$  to a floating point value. In fact, Maple "knows about"  $Q_\nu^\mu(z)$  and similar functions at a slightly deeper level because it knows properties such as recursion relations. But it doesn't know everything:

```

diff(LegendreQ(nu, mu, z), z);

$$\frac{(\nu - \mu + 1) \text{LegendreQ}(\nu + 1, \mu, z) - (\nu + 1) z \text{LegendreQ}(\nu, \mu, z)}{z^2 - 1}$$

diff(LegendreQ(nu, mu, z), nu);

$$\frac{\partial}{\partial \nu} \text{LegendreQ}(\nu, \mu, z)$$


```

## Debugging with the Debugger

*I have written a whole separate document with a debugger tutorial, but am leaving this text here since it gives a little overview.*

There is no nice GUI debugger (Maple V R5 1997), but what they have seems to work OK.

1. First, put your code in a separate file and *save it frequently* because Maple likes to crash in debug.
2. Get a new window, enter a restart statement then paste in your code to test.

3. One way you get into the debugger is by including a statement of the form **DEBUG('message')** right in your code. This statement then acts as the breakpoint. The string "message" will appear right after the code stops, so you can know which breakpoint you hit.

Another way is to put the statement **stopat(joe)** at the end of your subroutine joe.

4. You should run a call to your routine, and it will then break. Then issue a **step** command so make it step down to next line of code, however fine. Then do a **list** command which simulates what a GUI debugger would show, you see 11 lines of code.

Then the trick is to simple re-execute your step and your list, one after another, and in this way you single-step through the code. Perhaps you can have a third command set up so you can look at local variables. Just type the name of the variable you want to see and a ;/ That is how it is done in Maple!

The calls **step** and **next** are the usual ones. Step takes you down all the way, whereas next does not delve into lower details. Use next on system calls.

See also **into**, and **outfrom** for getting out of a loop.

5. You have to do a "stop" (quit) command before you can do anything else in Maple.

Crash Rule: You cannot have "other stuff" in a worksheet window below where you are debugging, or you will get crashes.

---

### **print printf sprintf etc**

As noted above, inside a procedure each command does not generate output as it does when you execute it at top level with a semicolon (as opposed to a colon) termination character. So if you want a program to display stuff on the screen, you use print or printf.

### **print( a,b,c,...)**

This command prints out Maple objects (expressions) in fancy notation. An example (joe is an undefined variable)

```
g := x^3*sin(mu);  
  
g := cos(θ)3 sin(μ)  
print(g, 2*mu, joe, "there", sin(theta)^2);  
cos(θ)3 sin(μ), 2 μ, joe, "there", sin(θ)2
```

In the print command you separate items by commas, and those commas appear on the output. You cannot make these commas not appear. A string appears with its double quotes. The good news is that you get all the fancy printing, but the bad news is that you can't really do much else. You cannot prevent print from issuing a newline before it prints and issuing one after it prints, and centering the stuff it prints. For example



```
> for n from 1 to 1 do printf("joe "); printf("bob"); print(phi,theta); printf("tom"); od;
joe bob
                                      $\phi, \theta$ 
tom
```

## printf (...)

This is very close to the C language printf statement so you have the usual flexibility there. The bad news is that the output cannot contain Greek letters or fancy-printed expressions. You can however get expressions to print in the linear format using the %a descriptor

```
[> printf("F(%d,%d,theta) = %a ",5,3,g);
F(5,3,theta) = cos(theta)^3*sin(mu)
```

Within a program, printf does not add a newline unless you tell it (see help), so you can print several items on the same line, as shown in the previous example with joe and bob. Maple has all the other printf family commands like sprintf and fprintf, see the help.

---

---

## MISC TOPICS

---

### Spreadsheets

You insert one from the insert menu. These are similar to Excel but NOT the same.

To move around in the sheet, there are no scrollbars, you just push against the sides of the sheet.

To select the spreadsheet for changing size, click just outside the sheet's boundary!

To make cell references, you must do it by hand, cannot just click as in Excel, AND you have to use a tilde such as ~A1. These do work in a relative sense, and the \$ business does absolutes as usual.

In this example, the right column boxes contain Maple math code entries which call a procedure which generates symbolic expressions for some associated Legendre functions

$L$	$M$	$\frac{Y(L, M)}{e^{(i M \phi)}}$
0	0	$\frac{1}{2} \frac{1}{\sqrt{\pi}}$
1	1	$-\frac{1}{4} \frac{\sqrt{3} \sqrt{2} \sqrt{1-x^2}}{\sqrt{\pi}}$
1	0	$-\frac{1}{2} \frac{\sqrt{3} x}{\sqrt{\pi}}$
1	-1	$\frac{1}{4} \frac{\sqrt{6} \sqrt{1-x^2}}{\sqrt{\pi}}$

---

**Appendix 1: The New User's Tour** ( my early notes which replicate some of the above)

### Maple V Release 5

2.8.99

**New User's Tour.** Another top level item, note that they are alpha sorted. They claim first to have 2700 functions and 1 million users! That is pretty good for a product in the upper stratosphere. Tour has 12 topics, they suggest 2 hours to do everything.

**Topic 1** is "working through the tour". Here we see the **worksheet**, which seems to allow command lines mixed with comments, lots of comments, etc.

**Topic 2** is "the worksheet env". First item is the **execution group**, which is a set of sequential commands and comments and a result, all encompassed in a left bracket. Results are one of three things:

**numeric, symbolic, or graphic!** We see an example of a **symbolic spreadsheet**, which is an Excel style sheet filled with perhaps a table of integrals, as you might find in a book of integrals! Spiffy!

When you have a living command going, right-click to see list of options and suggestions, this is the **context menu**. Ie, it depends where you are as to what you see listed on this menu. I don't get the context items as they describe, but later we shall see why.

A **paragraph** means a word-processor type of text paragraph, but fancy equation display is built in. Perhaps this is a better place to write a math paper than in Word with Equation 3.0, we shall see. This is the end of Topic 2.

**Topic 3** is on **numeric computations**. The operator **%** is called the **ditto operator**, and causes the previous result to be placed as input to a subsequent operation. First examples are with **integer** math. For floating point, they point out that integers are maintained until the very end. Use **evalf()** to get a floating point number from something. Another evaluator called **value()** does something different, not quite sure of the difference yet. Value seems to get a more perfect symbolic result, rather than a FP result. An example is the sum of an infinite series which might include PI. You say **evalf(%, 50)** to get a result to 50 decimal places. We have now seen how to enter series, product and summations.

You say **convert(%,polar)** to convert a complex number like  $5+4*I$  to polar. Note that cap I is the imaginary I, and you must show multiplication with the **\***.

The expo "e" is represented as  $\exp(1.0)$ , but pi is Pi, the Gamma function is GAMMA.

**My right-click mechanism is definitely not working!!!** The end of this topic 3 confirms that notion that has been accumulating!

**Topic 4** is **algebraic computations**. You create an expression using the **:=** operator (what language was that, I forget: Algol? ). Then later you can process that expression. **expand()** does just what you expect, and **factor(%)** does the reverse. Function **simplify()** uses trig identities to declutter a messy expression, such as  $\sin^2 + \cos^2$  would become just 1. Function **normal()** removes common factors from a num/den expression. The operator **:=** is the assignment operator, assigning an expression to some variable name you make up. You maybe define A, and then later say **eval(A,x=1)**; Notice that **semicolon** seems to end any command (again, some language is the basis here, need some history soon). The **eval** function puts variables to selected values, the result is in general still symbolic, as opposed to evalf. Conversion is a large world. You can convert to sum of partial fractions (parfrac).

Here is how you define a symbolic function:  $f := x - x^2 + 1/2$ . Then you can put anything you want in later for x, such as  $f(2)$  or  $f(a+b)$ . The strangely named function **unapply** does the same thing.

Next, we used assignment to define an object that is an equation:  $\text{equation} := 2*x+1 = x$ . Then later you can use **solve(equation,{x})** to get all the roots, notice the funny brackets here. You can also evaluate both sides of an equation using **eval(equation, x=2)**, the sides will be the same if you picked a root.

Next, we can solve **systems of equations**, very nice indeed. Trig and abs values are allowed. Holy cow! It also does a **system of inequalities**.

**Topic 5** is about **graphics**. We see that the term **with(plot)** brings in a class of C++ routines which you can then use, perhaps this is standard C++ terminology, I forget. So we bring in **plots** and **plottools**. The opening example is a **stunning simple animation**, that consumes a lot of my CPU. Next is an inequality system visualization plot.

OK, very impressive, and this is where you need a modern day PC and not some old 486 that would slug along on these rotations. 2D and 3D plots, change lighting (not shown), rotate, etc.

**Topic 6 is calculus.** This tutorial, like all of the above, speaks clearly for itself. When you want to remember this, just run it again! Functions are **Int**, **Diff**, and **value**, and here you see value doing its symbolic evaluation of integrals. Interesting example of the  $1/x$  object having finite volume but infinite area, I don't think I ever was aware of that before. It knows elliptic integral function, and all the special functions I guess. It knows when integrals diverge. It knows limits from two directions, and knows when something is undefined (instead of just blowing up the program, the time-honored historical method). You can create piecewise functions and process them just as regular functions. So far, things have been functions of one variable. The function **series()** will do a series expansion of a function around some point, presumably just a Taylor expansion.

**Topic 7 is differential equations.** First example is solving a second order ODE. You use the operator **D** to specify differentiation, there is a way to set the initial conditions, and then you just tell it to find the solution that meets those conditions and bang. Multiple variables, etc, a whole world here.

**Topic 8 is linear algebra**, meaning vectors and matrices. We can have matrices filled with symbolics or numbers. Eigenvalues and eigenvectors of a matrix. Special matrices are known, but I don't know about them. A nice feature is that Maple will emit FORTRAN or C code which you can copy into some other program or hardware for fast computation. They show as an example code for the inversion of a little matrix, but claim this ability is everywhere, call it **code generation**.

**Topic 9 is statistics and finance.** The finance stuff is pretty boring, present value, amortization, etc. These are just simple formula things. In the stat area, we can do least squares fit to a set of data points. We can make a scatterplot from some data, and so on.

**Topic 10 is programming.** You can write your own procedures, and you can view the source on almost all the built-in Maple functions, this is the open-architecture that would allow someone else to duplicate Maple I suppose. The programming language is some kind of composite of things I have seen over the years.

DONE with the tour!

**About Maple.** Website is [www.maplesoft.com](http://www.maplesoft.com).

(old notes, see newer below!)

The Maple system was originally developed at the University of Waterloo by Keith Geddes and Gaston Gonnet. The company, Waterloo Maple Inc., was founded in 1988. In 1993, Waterloo Maple acquired Prescience Corporation of San Francisco, a leading developer of equation editing, symbolic mathematics, graphics, and animation software. Waterloo Maple Inc. has grown from a five-person company to a world-class corporation with over **75 employees**. Waterloo Maple products are now marketed internationally through hardware and software OEMs, distributors, dealers, direct sales, and publishing partners.

Website has various patches that can be downloaded, bug fixes, improvements, etc.

Since Maple V Release 5 (Nov 1997), Maple has gone through releases 6(2000), 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 and is now at 18 (2013). As with many software programs, they have a release roughly every year, to keep that revenue flowing in -- just think of Windows or Office. Nobody wants an "old version" of software. Maple has to compete against flashy competition like Mathematica and the pizzazz has to be maintained in this foot race.

Wiki has an interesting Maple page, from which I quote 5 paragraphs:

"In 1999, with the release of Maple 6, Maple included some of the [NAG Numerical Libraries](#),<sup>[2]</sup> and made improvements to arbitrary precision arithmetic.

Between the mid 1995 and 2005 Maple **lost significant market share** to competitors due to a weaker user interface.<sup>[5]</sup> In 2005, Maple 10 introduced a new "document mode", as part of the standard interface. The main feature of this mode is that math is entered using two dimensional input, so that it appears similar to formulae in a book. In 2008, Maple 12 added additional user interface features found in Mathematica, including special purpose style sheets, control of headers and footers, bracket matching, auto execution regions, command completion templates, syntax checking and auto-initialization regions. Additional features were added for making Maple easier to use as a MATLAB toolbox.<sup>[6]</sup>

Maplesoft sells student, personal, academic and professional editions of Maple, with a substantial difference in price (US\$99, \$239cdn[citation needed], US\$1245, and US\$2,275, respectively). Later student editions (from version 6 onwards) have not placed computational limitations, but rather come with less printed documentation when purchased as a physical product. There is no difference in the product when purchased as an electronic download. Since Maple 14, all versions include the Maple Toolbox for MATLAB®, which used to be available as a separate product available only for academic and professional users.

Single-user editions of Maple are locked to the hardware of the computer they run on. This means that Maple may refuse to start if certain parts of the computer's hardware are removed or replaced. In this case the customer support has to be called, in order to receive a new license file for the updated hardware."

In September 2009 Maple and Maplesoft were acquired by the Japanese software retailer Cybernet Systems. "

I have a lot of sympathy and respect for the Maple crew.

## Appendix 2: List of Functions in Maple

The following are the names of Maple's standard library functions (there are about 550). You can get to this list by doing a Help search on topic "index" and then click on any item to learn more. Notice that the list includes  $\text{Dirac}(x) = \delta(x)$  and  $\text{Dirac}(n,x) = \delta^{(n)}(x)$  and Maple knows about the basic rules of these distributional symbolic functions. In particular,  $\text{diff}(\text{Heaviside}(x),x) = \text{Dirac}(x)$ .

AFactor	AFactors	AiriAiZeros	AiriBiYZeros
AiryAi	AiryBi	AngerJ	Berlekamp
BesselI	BesselJ	BesselJZeros	BesselK

Bessely	BesselyZeros	Beta	C
CHFARRAY	Chi	Ci	CompSeq
Content	D	DESol	Det
Diff	Dirac	DistDeg	Divide
Ei	Eigenvals	EllipticCE	EllipticCK
EllipticCPi	EllipticE	EllipticF	EllipticK
EllipticModulus	EllipticNome	EllipticPi	Eval
Expand	Expand	FFT	Factor
Factors	FresnelC	FresnelS	FresnelF
Fresnelg	GAMMA	GaussAGM	Gaussejord
Gausselim	Gcd	Gcdex	HankelH1
HankelH2	Heaviside	Hermite	Im
Interp	Inverse	Irreduc	JacobiAM
JacobiCD	JacobiCN	JacobiCS	JacobiDC
JacobiDN	JacobiDS	JacobiNC	JacobiND
JacobiNS	JacobiSC	JacobiSD	JacobiSN
JacobiTheta1	JacobiTheta2	JacobiTheta3	JacobiTheta4
JacobiZeta	KelvinBei	KelvinBer	KelvinHei
KelvinHer	KelvinKei	KelvinKer	KummerM
KummerU	LambertW	Lcm	LegendreP
LegendreQ	LerchPhi	Li	Linsolve
LommelS1	LommelS2	MOLS	Maple_floats
MatlabMatrix	MeijerG	Normal	Normal
Nullspace	Power	Powmod	Prem
Primitive	Primpart	ProbSplit	Product
Psi	Quo	RESol	Randpoly
Randprime	Ratrecon	Re	Rem
Resultant	RootOf	Roots	SPrem
Searchtext	Shi	Si	Smith
Sqrfree	Ssi	StruveH	StruveL
Sum	Svd	TEXT	WeberE
WeierstrassP	WeierstrassPPrime	WeierstrassSigma	WeierstrassZeta
WhittakerM	WhittakerW	Zeta	abs
add	addcoords	addressof	algebraic
algsubs	alias	allvalues	anames
antisymm	applyop	applyrule	arccos
arccosh	arccot	arccoth	arccsc
arccsch	arcsec	arcsech	arcsin
arcsinh	arctan	arctanh	argument
array	assign	assigned	asspar
assume	asubs	asympt	attribute
bernstein	branches	bspline	cat
ceil	charfcn	chrem	close
close	coeff	coeffs	coeftayl
collect	combine	commutat	comparray
compiletable	compoly	conjugate	content
context	convergs	convert	coords
copy	cos	cosh	cost
cot	coth	csc	csch
csgn	currentdir	dawson	define
degree	denom	depends	diagonal
diff	diffop	dilog	dinterp
disassemble	discont	discrim	dismantle
divide	dsolve	eliminate	ellipsoid
elliptic_int	entries	eqn	erf
erfc	erfi	eulermac	eval
evala	evalapply	evalb	evalc
evalf	evalfint	evalgf	evalhf
evalm	evaln	evalr	evalrC
exp	expand	expandoff	expandon
extract	factor	factors	fclose
fdiscont	feof	fflush	filepos
fixdiv	float	floor	fnormal
fopen	forget	fortran	fprintf
frac	freeze	fremove	frontend

fscanf	fsolve	galois	gc
gcd	gcdex	genpoly	getenv
harmonic	has	hasfun	hasoption
hastype	heap	hfarray	history
hypergeom	iFFT	icontent	identity
igcd	igcdex	ilcm	ilog
ilog10	implicitdiff	indets	index
indexed	indices	inifcn	ininame
initialcondition	initialize	insert	int
intat	interface	interp	invfunc
invztrans	iostatus	iperfpow	iquo
iratecon	irem	iroot	irreduc
iscont	isdifferentiable	isolate	ispoly
isqrfree	isqrt	issqr	latex
lattice	lcm	lcoeff	leadterm
length	lexorder	lhs	limit
ln	lnGAMMA	log	log10
lprint	map	map2	match
matrix	max	maximize	maxnorm
maxorder	member	min	minimize
minpoly	modp	modp1	modp2
modpol	mods	msolve	mtaylor
mul	nextprime	nops	norm
normal	nprintf	numboccur	numer
odetest	op	open	optimize
order	parse	patmatch	pclose
pclose	pdsolve	pdetest	pdsolve
piecewise	plot	plot3d	plotsetup
pochhammer	pointto	poisson	polar
polylog	polynom	powmod	prem
prevprime	primpart	print	printf
procbody	procmake	product	proot
property	protect	psqrt	queue
quo	radnormal	radsimp	rand
randomize	randpoly	range	rationalize
ratrecon	readbytes	readdata	readlib
readline	readstat	realroot	recipoly
rem	remove	residue	resultant
rhs	root	roots	round
rsolve	savelib	scanf	searchtext
sec	sech	select	seq
series	setattrtribute	shake	showprofile
showtime	sign	signum	simplify
sin	singular	sinh	sinterp
smartplot3d	solve	solvefor	sort
sparse	spline	split	splits
sprem	sprintf	sqrtfree	sqrt
sscanf	ssystem	stack	string
sturm	sturmseq	subs	subsop
substring	sum	surd	symmdiff
symmetric	syntax	system	table
tan	tanh	testeq	testfloat
thaw	thiele	time	timelimit
translate	traperror	trigsubs	trunc
type	typematch	unames	unapply
unassign	unload	unprotect	userinfo
value	vector	verify	whattype
with	worksheet	writebytes	writedata
writeline	writestat	writeto	zip
ztrans			

---

## MATLAB linkage

This is a package within Maple that you invoke using `with(Matlab)`. It fails to run because it wants `libmx.dll` which does not seem to be in my installation. I now see that Maple is just providing a "link" to the Matlab package, if you happen to own that package.

- The functions available are:

```
chol   closelink defined det       dimensions
eig   evalM    fft     getvar   inv
lu    ode45    openlink qr      setup
setvar size    square  transpose
```

---

## Fast Fourier Transform

Put the real and imaginary parts of your starting vector into two arrays, call the FFT, and the result will come back in those same two arrays. There is not need to fret over the missing Matlab `fft`, the Maple one works fine and fast.

---

## Maple Hot Keys Summary for Windows

Function	Hot Keys
Bold Style Attribute	Ctrl + B
Close Current Worksheet	Ctrl + F4
Close Help Topic	Ctrl + F4
Copy Selection to Clipboard	Ctrl + C
Cut Selection to Clipboard	Ctrl + X
Delete Entire Object	Ctrl + Delete
Enclose Selection in Subsection	Ctrl + .
Exit Maple	Alt + F4
Find Next, Find Previous	Ctrl + F
Go to Beginning of Line	Home
Go to Bottom of Worksheet	Ctrl + End
Go to Closest Parent	Ctrl + UpArrow
Go to End of Line	End
Go to Top of Worksheet	Ctrl + Home
Hard New Line	Shift + Enter
Input Mode/Text Mode Toggle	F5
Insert Execution Group	
...After Cursor	Ctrl + J
...Before Cursor	Ctrl + K
Insert Paragraph	
...After	Shift + Ctrl + J
...Before	Shift + Ctrl + K
Italic Style Attribute	Ctrl + I
Maple Input Mode	Ctrl + M
Move To Next (Previous) Input	Tab (Shift + Tab)
New Worksheet	Ctrl + N
Open Edit Menu	Alt + E
Open File Menu	Alt + F
Open Format Menu	Alt + R
Open Help Menu	Alt + H
Open Insert Menu	Alt + I



Open Options Menu	Alt + O
Open View Menu	Alt + V
Open Window Menu	Alt + W
Open Worksheet	Ctrl + O
Paste Selection from Clipboard	Ctrl + V
Print Worksheet	Ctrl + P
Redraw Screen	Ctrl + L
Remove Section Enclosing Selection~ (Outdent)	Ctrl +
Save Worksheet	Ctrl + S
Select All	Ctrl + A
Show Region Ranges	F9
Show Section Ranges	Shift + F9
Execution Groups	
... Split	F3
... Join	F4
Sections	
... Split	Shift F3
... Join	Shift F4
Text Input Mode	Ctrl + T
Underline	Ctrl + U
Undo Last Cut	Ctrl + Z
Zoom Factor...50%	Ctrl + 1
Zoom Factor...100%	Ctrl + 2
Zoom Factor...150%	Ctrl + 3
Zoom Factor...200%	Ctrl + 4
Zoom Factor...300%	Ctrl + 5
Zoom Factor...400%	Ctrl + 6

---

sign	name	example	meaning
+	operator of addition	$a + b$	(a plus b)
—	operator of subtraction	$a - b$	(a minus b)
*	operator of multiplication	$a * b$	(a times b)
/	operator of division	$a / b$	(a over b)
^, **	operators of involution	$a ^ b$	(a to the b <sup>th</sup> power)
!	operator of factorial	5!	$1 \times 2 \times 3 \times 4 \times 5 = 120$
Pi	3.1415	Pi/2	1.570796327
pi <sup>i</sup>	$\pi$	pi/2	$\pi/2$
sqrt	square root of	sqrt (x)	square root of x
surd	operator of evolution	surd(x, n)	n <sup>th</sup> root of x, $\sqrt[n]{x}$
log	logarithm	log[b] (x)	$\log_b x$
ln	natural logarithm (of base 2.718)	ln (x)	natural logarithm of x
exp	the exponential function	exp (x)	$e^x$
exp(1)	the base of the natural logarithm	exp (1)	2.718281828
sin	trigonometric functions	sin (x)	sine of x
cos		cos (x)	cosine of x
tan		tan (x)	tangent of x
cot		cot (x)	cotangent of x
arcsin	inverse trigonometric functions	arcsin(x)	arcsine of x
arccos		arccos(x)	arccosine of x
arctan		arctan(x)	arctangent of x
arccot		arccot(x)	arccotangent of x