

An Introduction to MATHEMATICAL COMPUTATION USING MAPLE V

by

Ronald S. Harichandran
Professor and Chairperson

Department of Civil and Environmental Engineering
Michigan State University
East Lansing, MI 48824-1226

Phone: (517) 355-5107
FAX: (517) 432-1827
E-Mail: harichan@cee.msu.edu

Acknowledgements

Mr. Scott Stowitts, a graduate student in civil and environmental engineering, assisted in the preparation of this guide while being supported by the DeVlieg Fellowship in 1993. The DeVlieg Fellowship is administered by the Division of Engineering Computer Services at Michigan State University.

September 1999



Table of Contents

1.	Starting Maple on Sun Workstations Supported by DECS	1
2.	Basic Features	1
	2.1 Interface	1
	2.2 Maple Objects	2
3.	Algebraic Manipulations.....	4
4.	Numerical Evaluation	5
5.	Solving Equations	6
6.	Calculus.....	6
7.	2-D and 3-D Graphs.....	8
8.	Solution of Differential Equations.....	9
9.	Transforms	10
10.	Linear Algebra	10
11.	Linear Programming	13
12.	Statistics	13
13.	The Maple Programming Language	15
14.	Reading and Writing Files, and Printing.....	16
15.	Useful Miscellaneous Commands.....	17
16.	Initialization File and Customizing User Interface.....	18
	References.....	18

Maple V is a powerful computer-based tool for performing symbolic, numerical and graphical mathematical computation. It is available on a wide variety of platforms including IBM mainframes, UNIX workstations, PC's running MS-Windows and Macintosh's. Student editions of Maple for PC's are available for a reasonable cost at most bookstores. This is a short guide to help you get started with Maple V Release 3 or 4 for engineering applications. Maple has extensive on-line help that can be used to find out about its numerous commands and features. In addition, tutorial guides, books and manuals are also available (Char et al. 1992, Char et al. 1991, Abell and Braselton 1994a, Abell and Braselton 1994b, Abell and Braselton 1994c).

1. Starting Maple on Sun Workstations Supported by DECS

To start Maple V type the following after logging in:

1. `set path=($path /opt/maple/bin)`
2. `xmaple`

You may wish to place the first command at the bottom of your `.cshrc` file so that the required directory is automatically included in your path every time you log in.

2. Basic Features

2.1 Interface

- Help:** On-line help can be obtained conveniently using the *Help Browser* option from the *Help* menu located at the top right corner of the Maple window. Detailed help on function `fname` can also be obtained using the Maple command `?fname`.
- Prompt:** Maple commands are entered at the `>` prompt. On color terminals the typed commands appear in red.
- Fonts:** Fonts displayed on the screen and used when printing can be changed using the *Format/Styles* menu option.
- Cut & Paste:** Parts of previously typed commands (red text on color terminals) may be cut by clicking and dragging with the left mouse button, placing the cursor at the bottom line by clicking the left button next to the prompt, and pressing the middle mouse button to paste the text. Extended regions may be copied to the clipboard and pasted using the *Edit* menu.
- Scrolling:** The scroll bar on the right side of windows can be used to scroll back to review previous work.
- Terminators:** **Each command must be terminated with a semi-colon (;) or a colon (:).** The semi-colon echoes the typed line, while the colon suppresses the echo.
- Case:** Maple is case sensitive. `xyz` is different from `XYZ`.
- History:** The previous three commands may be recalled by typing `"`, `"` or `" "`. For a longer history mechanism use the `history` command.
- Saving:** Select *Save* or *Save As...* from the *File* menu. For a new worksheet, or when the *Save As...* command is selected, enter the file name in the *Filename* box or double click on a file appearing in the list of files. Previously saved file can be retrieved using the *Open...* command from the *File* menu.

Printing: Select *Print...* from the *File* menu, and on the pop-up form, click on the *Print Command* radio button, and type “lp -dprinter_name” in the *Printer Command* box (where *printer_name* should be replaced by the name of a printer). Other print options can also be selected on the form.

2.2 Maple Objects

Operators: Arithmetic operators are +, -, * and / for addition, subtraction, multiplication and division, ** or ^ for exponentiation and ! for factorial. Boolean operators that return True or False are: <, <=, >=, <> and = for less than, less than or equal, greater than or equal, not equal, and equal; and or, not and and.

Constants: Constants consist of integers, floating-point numbers, rational numbers (e.g., -1/3, 5/7), binary constants (true and false), π (Pi), e (exp(1)), ∞ (infinity), $\sqrt{-1}$ (I or sqrt(-1)), Euler’s constant (gamma), etc.

Expressions: Any algebraic expression. e.g., $2*x + y$. Greek symbols represented by their full name (e.g., omega for ω and Omega for Ω) are echoed back in the Greek form in the X-Windows interface.

Equations: An equation must contain an = sign. e.g., $2*x + y = 3$.

Evaluation: Normally Maple manipulates expressions symbolically. If numeric evaluation is required, the following functions can be used:

`evalf(expr, n)` evaluates the expression using floating point arithmetic to n digits (n is optional).

`evalc(expr)` evaluates the expression using complex arithmetic.

Variables: Long expressions, numerical results, etc. can be assigned to variables using *variable := expression*;

e.g., `exp1 := 2*x + y`;

Note that `exp1` is a Maple variable, while `x` and `y` are algebraic variables used to construct the expression.

Functions: The built-in mathematical functions are: `exp(x)`, `ln(x)` or `log(x)`, `log10(x)` for logarithm to the base 10, `log[b](x)` for logarithm to the base b , `sqrt(x)`, `abs(x)`, `min(...)`, `max(...)`, `iquo(...)` for truncating division, `irem(...)` for integer remainder, `round(x)` to round to nearest integer, `trunc(x)` to truncate, `frac(x)` for fractional part, `a mod b` for modular arithmetic, `gcd(...)` for greatest common divisor, `lcm(...)` for least common multiple, `signum(x)` for the sign function, `sin(x)`, `cos(x)`, `tan(x)`, `sec(x)`, `csc(x)`, `cot(x)`, `sinh(x)`, `cosh(x)`, `tanh(x)`, `sech(x)`, `csch(x)`, and `coth(x)` for the trigonometric and hyperbolic functions, `arcsin(x)`, `arccos(x)`, etc., for the inverse trigonometric and hyperbolic functions, `GAMMA(x)` for the Gamma function, `binomial(n,m)` for the binomial coefficient, `Beta(x,y)` for the Beta function, `erf(x)` for the error function, `Dirac(t)` for the Dirac delta function, `Heaviside(t)` for the unit step function, and so on.

User-defined functions can be specified using a special syntax or as Maple procedures as shown in Table 1. Procedures are written using the Maple programming language (see Section 13).

- Sequences:** A simple ordering of Maple objects separated by commas.
 e.g., `a, b, c` is a sequence.
`s := seq(x^2, x=0..5)` assigns the sequence 0,1,2,3,4,5 to `s`.
 The third to the fifth elements of `s` may be extracted with `s[3..5]`.
- Sets:** A set is a sequence surrounded by braces `{ }`. e.g., `{a, b, c}`.
 The operations `union`, `intersect` and `minus` can be used to with sets.
 e.g., `set1 := {a, b}`:
`set2 := {a, d, e}`:
`set1 union set2` would yield `{b, a, d, e}`.
 The operand of a set may be extracted as a sequence with `op(range, set)`.
 e.g., `op(2..3, set2)` would yield the sequence `d, e`.
 The number of operands in a set may be obtained with `nops(set)`.
 e.g., `nops(set3)` would yield 3.
- Lists:** A list is a sequence surrounded by square brackets `[]`. e.g., `[a, b, c]`.
 The elements of a list may be extracted as a sequence with `list[i..j]`.
 e.g., `L := [17, f, ln(y), [1,9]]`:
`L[3]` would return `ln(y)`. `L[4]` would return the list `[1,9]`. `L[2..3]` would return the sequence `f, ln(y)`.
 The command `map(f, L)` can be used to apply the function `f` on every element of the list, set, array, vector or matrix `L`.
- Arrays:** An array is a collections of objects referenced by integer subscripts. The subscripts can contain negative integers and zero.
 e.g., `a := array(1..2, 1..2)` defines a 2×2 array.
`a[1,1]:=1: a[1,2]:=2:` assigns values to the array elements.
`print(a)` would print the entire array on the screen.
 Arrays can be defined using the indexing functions `sparse`, `symmetric`, `antisymmetric` or `identity` by the command `array(indexing function, ...)`. Sparse arrays are assumed to have zero elements unless specified otherwise, `symmetric` and `antisymmetric`.

TABLE 1 DEFINING FUNCTIONS

Type	Special Syntax	Procedure
1-D	<code>f := x -> x^2 - a:</code>	<code>f := proc(x) x^2 - a end:</code>
2-D	<code>f := (x,y) -> x*y-1:</code>	<code>f := proc(x,y) x*y-1 end:</code>
1-D Piecewise	<code>readlib (piecewise):</code> <code>f := x -> piecewise (x>=0,</code> <code>exp(-a*x), 0)</code>	<code>f := proc(x) if x>=0 then</code> <code>exp(-a*x) else 0 fi: end:</code>
2-D Piecewise	<code>readlib (piecewise):</code> <code>f := (x,y) -> piecewise</code> <code>(x*y>=0, x^2 + y^2, 0)^a</code>	<code>f := proc(x,y) if x>=0 and</code> <code>y>=0 then x^2 + y^2</code> <code>else 0 fi: end:</code>

^aNote that the conditions used must be pure relations and cannot involve boolean constructs such as “`x>=0` and `y>=0`”.

metric arrays have those properties enforced on their elements, and identity arrays have unit diagonal elements with all other elements being zero.

The `indices(array)` function lists the indices that correspond to non-zero elements, and the `entries(array)` function lists the non-zero entries.

Vectors: Vectors are one-dimensional arrays of the form `array(1..n)`.

Matrices: Matrices are two-dimensional arrays of the form `array(1..n, 1..m)`.

Strings: Procedure names, file names, etc. that are strings must be enclosed by back-quotes when used as arguments to commands.

3. Algebraic Manipulations

<code>collect(expr, var) ;</code> <code>collect(expr, var, name) ;</code>	Collect coefficients of like powers of <i>var</i> . The second form allows terms containing <i>name</i> (e.g., <code>sin(x)</code>) to be collected.
<code>coeff(expr, var) ;</code> <code>coeff(expr, var^n) ;</code>	Extracts the coefficient of var^n in <i>expr</i> . In the first form <i>n</i> defaults to 1. Use <code>collect</code> before using <code>coeff</code> .
<code>sort(expr) ;</code>	Sorts the polynomial or list <i>expr</i> into ascending order.
<code>expand(expr) ;</code> <code>expand(expr, expr₁, ..., expr_n) ;</code>	Expand expression by multiplying it out. The optional arguments <i>expr₁</i> , ..., <i>expr_n</i> are used to prevent particular sub-expressions from being expanded. Functions such as <code>sin(2*x)</code> are also expanded.
<code>combine(expr) ;</code> <code>combine(expr, name) ;</code>	Combine terms in sums, products and powers into a single term. Values for <i>name</i> should be one of <code>exp</code> , <code>ln</code> , <code>power</code> , <code>trig</code> or <code>Psi</code> . For many functions, the transformations applied by <code>combine</code> are the inverse of the transformations that are applied by <code>expand</code> .
<code>simplify(expr) ;</code> <code>simplify(expr, rule) ;</code>	Simplify <i>expr</i> . The type of simplification can be controlled by specifying <i>rule</i> (commonly <code>power</code> , <code>radical</code> , <code>RootOf</code> , <code>sqrt</code> or <code>trig</code>).
<code>factor(expr) ;</code> <code>factor(expr, field) ;</code>	Factor a polynomial over the rational numbers. The second form allows factoring over other algebraic fields (e.g., <code>factor(x^2+4,I)</code> yields $(x + 2 I)(x - 2 I)$).
<code>subs(var = replacement, expr) ;</code> <code>subs(var₁ = replacement₁, ..., var_n = replacement_n, expr) ;</code>	Substitute all occurrences of <i>var</i> in <i>expr</i> by the specified replacement expression. The second form is used to substitute for several variables.
<code>assign(var = expr) ;</code> <code>assign(var₁ = expr₁, ..., var_n = expr_n) ;</code>	Globally assign <i>expr</i> to the variable <i>var</i> . The second form is used to assign values to many variables. All other previously defined objects containing the variables are updated using the assigned values. To un-assign the value assigned to a variable <i>x</i> (i.e., to let the variable become an unknown again) enter <code>x := 'x'</code> .

`convert(expr, form, optargs);`

Convert expressions from one form to another. Valid forms include: `parfrac` with `optargs` set to `var` for a partial fraction expansion about the main variable `var`; `fraction` to express a floating point number as a whole fraction, `polar` for polar form of a complex expression, and so on. The type of conversion usually depends on the form of `expr`. Look under the *Library./convert.* topic in the *Help Browser* for a detailed list.

`lhs(expr); rhs(expr);`

Returns the left-hand side and right-hand side of the equation or inequality `expr`.

`numer(expr); denom(expr);`

Numerator and denominator of `expr`.

`Re(expr); Im(expr);`
`conjugate(expr);`

Real and imaginary parts, and the conjugate, of the complex-valued `expr`.

`evalc(expr);`

Attempts to split complex-valued `expr` into its real and imaginary parts expressing the result as `expr_1 + I*expr_2`. Unknown variables are assumed to be real-valued. This command is often useful for expressing complex exponentials in terms of sine and cosine functions.

`map(f, expr);`

`map(f, expr, arg2, arg3, ..., argn);`

Applies the function `f` to each element of `expr`, where `expr` may be any structure such as a sequence, list, set, matrix, etc. Some functions do not work with certain types of arguments. For example, the `simplify` command works with a single expression, or a list of expressions, but does not work with a matrix. The command `map(simplify, mat)`, simplifies each element of the matrix, `mat`.

For Maple commands that need arguments, first a user-defined function must be created and then used with the `map` command. For example, the `subs` command does not work on matrices, and a user-defined function must first be created and then used with `map`, as shown below, to perform substitution within matrices:

```
mysubs := proc(x,y) subs(y,x) end;  
map(mysubs, mat, {var1 = replacement1, ..., varn = replacementn});
```

4. Numerical Evaluation

`Digits := n;`

Sets the precision for numerical evaluations. The default is `Digits := 10`.

`evalf(expr);`
`evalf(expr, digits);`

Evaluates `expr` using floating-point calculations. In the second form, `digits` temporarily overrides the precision set by the variable `Digits`.

5. Solving Equations

```
solve(eqn) ;  
solve(eqn, var) ;  
solve(eqn_set, var_set) ;
```

The first form solves an equation in one variable (e.g., `solve(3*x + 4 = 5*x)`). The second form solves for *var* in an equation containing several variables (e.g., `solve(ln(x^2-1)=a, x)`). If multiple solutions exist, then a solution sequence is returned. The third form solves a set of equations for a set of variables (e.g., `solve({a=b+2, b-a=t}, {a,b})`) and returns a solution set, or a sequence of solution sets if multiple solutions exist. Solutions may sometimes contain the expression `RootsOf(polynomial)` to signify that the solutions are the roots of the indicated polynomial.

```
allvalues(expr) ;
```

Computes all possible values of solutions returned with the expression `RootsOf`. Since `RootsOf` may contain several roots of a polynomial, each root is substituted into the solutions to obtain all possible solutions.

```
fsolve(eqn) ;  
fsolve(eqn, var) ;  
fsolve(eqn_set, var_set) ;  
fsolve(eqn_set, var_set, options) ;
```

Numerical solution of equations. Equations that cannot be solved exactly using `solve` may be able to be solved numerically using `fsolve`. When a solution cannot be found, the input expression is returned, but there could still be a possible solution (specifying a range may help). Valid *options* are: `complex` (e.g., complex roots of polynomial); `a..b` to limit search to open interval (e.g., `{x=a..b, y=c..d, ...}`); `maxsols=n` to find at most *n* roots; and `fulldigits` to evaluate roots to the precision specified by the variable `Digits`.

6. Calculus

```
limit(expr, var = a) ;  
limit(expr, var = a, option) ;
```

The first form computes the bi-directional limit of *expr* as *var* approaches the number *a*, except when *a* is infinity or -infinity in which case the appropriate directional limit is computed. The second form is more general and a valid *option* is one of `left`, `right`, `complex` or `real` (the last option also computes the bi-directional limit). If a limit is not found the input expression is returned.

```
series(expr, var, n) ;  
series(expr, var=a, n) ;
```

Series expansion of *expr* up to order *n*. In the first form the expansion is about the origin, while in the second form it is about the point *a*. If a Taylor series is specifically desired, then `series` may be replaced by `taylor`.

```
mtaylor(expr, [var1, ..., varn], n) ;  
mtaylor(expr, [var1=a1, ..., varn=an], n) ;
```

Multi-dimensional Taylor series of *expr* up to order *n*. In the first form the expansion is about the origin, while in the second form it is about the point (*a*₁, ..., *a*_{*n*}).

```
sum(expr, index = low..high);
sum(expr, index);
```

The first form is used for a definite summation (e.g., `sum(i,i=1..1000)`); or symbolic sum (e.g., `sum(i^2 i=1..n)`). The second form is used for an indefinite summation.

```
product(expr, index = low..high);
product(expr, index);
```

Similar to the `sum` command, except that the product is computed.

```
readlib(minimize):
minimize(expr);
minimize(expr, var1, ..., varn);
maximize(expr);
maximize(expr, var1, ..., varn);
```

Unconstrained minimization and maximization of `expr`. The third and fifth forms are used to minimize and maximize multi-dimensional functions with respect to the variables `var1, ..., varn`. The `readlib(minimize)` command must precede the first call to `minimize` or `maximize` in the current session.

```
readlib(singular):
singular(expr);
singular(expr, {var1, ..., varn});
```

Finds singularities of `expr`. The `readlib(singular)` command must precede the first call to `singular` in the current session. In the last form, `expr` is assumed to be a function of `var1, ..., varn`.

```
diff(expr, var1, ..., varn);
```

Partial derivative of `expr` with respect to `var1, ..., varn`. Specify variables repeatedly for higher-order ordinary derivatives (e.g., `diff(sin(x), x, x)` evaluates $d^2(\sin x)/dx^2$). The syntax `diff(expr, var1$n)` or `diff(expr, var1$n1, var2$n2, ...)` can be used for higher-order derivatives, where `var1$n` represents n repetitions of `var`.

```
int(expr, var);
int(expr, var = low..high);
int(expr, var = low..high, option);
```

Indefinite or definite integral of `expr` with respect to `var`. The values `low` and `high` specify the range for definite integration. If the integral cannot be evaluated in closed-form then it is displayed unevaluated, and if desired numerical integration or series expansion may then be performed using `evalf` or `series`. For definite integration the existence of discontinuities are checked by default, and the integral is evaluated as the sum of independent integrals, each of which has no discontinuity. Specifying `continuous` for `option` disables this check. To compute the Cauchy Principal Value of an integral containing a discontinuous integrand specify `option` to be `CauchyPrincipalValue`. For double and triple integrations use nested constructions of `int` (e.g., `int(int(x^2*y^2, x), y)` for $\iint x^2y^2 dx dy$).

```
Diff(...);
Int (...);
```

These are the inert forms of `diff` and `int`, and have arguments identical to those. They return unevaluated, simply producing mathematically formatted display, and are useful for visually checking complicated expressions, before actually evaluating them using `diff` or `int`.

```
assume(var1 > 0, var2 < 0, ...);
```

Assume that `var1` is positive, `var2` is negative, etc. Some integrals that cannot be evaluated for general values of its sec-

```
readlib(residue);
residue(expr, var = a);
```

ondary variables, may be able to be evaluated if the range of the variables is known (e.g., $\int_0^\infty e^{-at} \ln t \, dt$ is evaluated only if it is assumed that $a > 0$).

Algebraic residue of $expr$ for the chosen variable, var , around the point a . The `readlib(residue)` command must precede the first call to `residue` in the current session.

7. 2-D and 3-D Graphs

```
plot(expr, var_1=a..b, options);
plot(expr, var_1=a..b, c..d,
      options);
plot({expr_1, ..., expr_n}, var_1=a..b,
      options);
plot({expr_1, ..., expr_n}, var_1=a..b,
      c..d, options);
```

Normal 2-D graphs. In the first and third forms, the y-axis is scaled automatically, while in the second and fourth forms it goes from c to d . The third and fourth forms (with a set of expressions) is used to plot several graphs on the same plot. The most common *option* is `title='plot title'` to write a plot title at the top of the plot.

A series of data points can be plotted by specifying a list $[x_1, y_1, \dots, x_n, y_n]$ in place of $expr$.

In the X-Windows interface, several plot features can be controlled using a pop-up menu activated by clicking the right mouse button in the plot window.

```
plot([x-expr, y-expr, param=a..b],
      options);
plot([x-expr, y-expr, param=a..b],
      c..d, e..f, options);
plot([x-expr_1, y-expr_1,
      param_1=a_1..b_1], ..., [x-expr_n,
      y-expr_n, param_n=a_n..b_n],
      options);
```

2-D parametrized plots. In the first form both axes are scaled automatically while in the second form the X-axis goes from c to d and the Y-axis from e to f . The third form is used for a multiple parametrized plot. The most common options are to write plot titles and X and Y labels as described above for regular 2-D plots. e.g., `plot([sin(t), cos(t), t=0..2*Pi])`; plots a circle with unit radius centered at the origin.

```
plot([r-expr, theta-expr, theta=a..b],
      coords=polar, options);
plot([r-expr_1, theta-expr_1, theta=a_1..b_1],
      ..., [r-expr_n, theta-expr_n,
      theta=a_n..b_n], coords=polar,
      options);
```

2-D parametrized polar plots. The first form is used for a single plot and the second form for a multiple plot. The most common *options* are to write a plot title and X and Y labels as described previously for regular 2-D plots. e.g., `plot([1, x, x=0..2*Pi], coords=polar)`; plots a circle with unit radius centered at the origin. Type `?polarplot` in Maple for help on other ways of obtaining polar plots.

```

plot3d(expr, var1=a..b,
        var2=c..d, options);
plot3d({expr1, ..., exprn},
        var1=a..b, var2=c..d, options);
plot3d([x-expr, y-expr, z-expr,
        param1=a..b, param2=c..d],
        options);
plot3d([x-expr1, y-expr1, z-expr1,
        param11=a1..b1,
        param21=c1..d1], ..., [x-exprn,
        y-exprn, z-exprn, param1n=an..bn,
        param2n=cn..dn], options);

```

3-D surface plots. The first two forms are used for single and multiple surfaces in Cartesian coordinates. The third and fourth forms are used for parametric plots in which expressions within each list are defined in terms of two parameters. Common *options* are `title='plot title'` to write a plot title, `labels=['xlab', 'ylab', 'zlab']` to write axes labels, `grid=[m,n]` to generate the surface using an $m \times n$ grid of equally spaced values (default is 25×25) and `coords=spherical` or `cylindrical` for plots in these other coordinates.

In the X-Windows interface, several features of the plot can be controlled using a pop-up menu activated by clicking the right mouse button in the plot window. Dragging the mouse while the left mouse button is depressed causes the view angle to change. Select *Redraw* from the pop-up menu to display the plot using the new plot angle. The plot style can be changed using the *Style* menu (default is *Hidden Line* removal), surface colors can be changed using the *Color* menu (default is *XYZ* which changes the colors continuously in all three directions), axes styles can be changed with the *Axes* menu (default is *None*), and the projection styles can be changed with the *Projection* menu (default is *No Perspective* and *Unconstrained*). Experiment with these options to get the plot you like.

Type `?sphereplot`, `?cylinderplot`, `?spacecurve`, `?tubeplot` or `?pointplot` in Maple for help on other ways to obtain spherical and cylindrical plots, and for help on other types of 3-D plots.

8. Solution of Differential Equations

Maple is able to find closed-form solutions to many differential equations. The solution is returned either as an equation in $y(x)$ and x (or whatever variables were specified) or in parametric form $[x=f(_T), y(x)=g(_T)]$ where $_T$ is the parameter. Any arbitrary constants are represented as $_C1$, $_C2$, ..., $_Cn$. Derivatives in the equations can be specified using either the `diff` or `D` operators. Initial or boundary conditions often need to be specified. Conditions involving derivatives must be specified using the `D` operator. Examples of differential equations without and with conditions are shown in Table 2.

The notation `diff(y(x), x$2)` is equivalent to `diff(y(x), x, x)` and `(D@@2)(y)(t)` is equivalent to `D(D(y))(t)`.

The `dsolve` command for the solution of differential equations takes one of the following forms

```

dsolve(deqn, var, option);
dsolve({deqn1, ..., deqnn, cond1, ..., condm}, {var1, ..., varn}, option);

```

TABLE 2 EXAMPLES OF SPECIFYING DIFFERENTIAL EQUATIONS

Differential Equation	Conditions
$\text{diff}(y(x), x) - y(x) = 1$	None
$\text{diff}(v(t), t) + 2*t = 0$	$v(1)=5$
$\text{diff}(y(t), t) + 5*\text{diff}(y(t), t) + 6*y(t) = 0$	$y(0)=0, D(y)(0)=1$
$(D@@2)(y)(t) + 5*D(y)(t) + 6*y(t) = 0$	$y(0)=0, D(y)(0)=1$

The variables var, var_1, \dots, var_n are specified as functions of the independent variables (e.g., $\{y(x), z(x)\}$). The *option* may be `explicit`, `laplace`, `series` or `numeric`. If the differential equation contains the `Dirac` or `Heaviside` functions, then `laplace` must be used. The initial conditions must be specified at $x=0$ if the `laplace` or `series` option is used.

9. Transforms

`laplace(expr, t, s):`
`invlaplace(expr, s, t);`

Laplace transforms and its inverse. The forward transform transforms from t to s , and the inverse transform transforms from s to t .

`ztrasn(expr, n, z):`
`invztrans(expr, z, n);`

Z transform and its inverse. The forward transform transforms from n to z and the inverse transform transforms from z to n .

`readlib(fourier):`
`fourier(expr, t, ω):`
`invfourier(expr, ω, t);`

Fourier transforms and its inverse. The `readlib(fourier)` command must precede the first call in the current session to `fourier` or `invfourier`. The forward transform transforms from t to ω and the inverse transform transforms from ω to t . Note that the $\frac{1}{2\pi}$ factor occurs in the inverse transform.

`readlib(FFT):`
`FFT(m, x, y):`
`iFFT(m, x, y);`

Fast Fourier transform and its inverse. The `readlib(FFT)` command must precede the first call in the current session to `FFT` or `iFFT`. Both x and y are real-valued arrays of length 2^m that contain the real and imaginary parts of a complex array. The transforms are performed in place, and the result is returned in x and y .

10. Linear Algebra

Maple allows some matrix operations to be performed in the standard package. These and more involved operations can be performed using the `linalg` package. In the following summary, commands that do not need the `linalg` library are marked with †.

`with(linalg):`

Load the linear algebra package. This must precede any command requiring this package.

```
vec:=vector([expr1, ..., exprn]);
vec:=vector(n, [expr1, ...,
    exprn]);
vec:=vector(n, f);
vec:=vector(n);
```

Define *vec* to be a vector. In the first two forms, the elements of the vector are assigned explicitly. In the third form, the elements are computed using the function *f* as $f(1), \dots, f(n)$. In the last form, the elements are undefined. Elements of a vector can be accessed with *vec*[*i*].

```
vec:=array(1..n, [expr1, ...,
    exprn]);†
vec:=array(1..n);†
```

Alternative way of defining vectors without the `linalg` package.

```
norm(vec, 2);
```

The length (Euclidean norm) of *vec*.

```
dotprod(vec1, vec2);
crossprod(vec1 &* vec2);
```

Dot and cross products of vectors.

```
grad(expr, vec);
```

Gradient of *expr* with respect to the vector or list of variables *vec*.

```
curl([expr1, expr2, expr3], [var1,
    var2, var3]);
```

Curl of the three-dimensional function $f = [expr_1, expr_2, expr_3]$ with respect to the three variables.

```
diverge([expr1, ..., exprn], [var1,
    ..., varn]);
```

Divergence of a vector function $f = [expr_1, \dots, expr_n]$ with respect to its variables.

```
potential([expr1, ..., exprn],
    [var1, ..., varn], 'pot');
```

Determines whether the vector field $f = [expr_1, \dots, expr_n]$ is derivable from a scalar potential $V = [var_1, \dots, var_n]$ such that $\nabla V = f$. If this is possible, then the value `true` is returned, and the potential is assigned to the variable *pot*.

If a potential does not exist then `false` is returned.

```
vecpotent([expr1, expr2, expr3],
    [var1, var2, var3], 'pot');
```

Determines whether the vector function $f = [expr_1, expr_2, expr_3]$ is derivable from a vector potential $V = [var_1, var_2, var_3]$ such that $\nabla \times V = f$. If this is possible, then the value `true` is returned, and the potential is assigned to the variable *pot*. If a potential does not exist then `false` is returned.

```
mat:=matrix(L);
mat:=matrix(m, n, L);
mat:=matrix(m, n, f);
mat:=matrix(m, n);
```

Define *mat* to be a matrix. *L* is a list of lists or vectors of the form $[[expr_{11}, \dots, expr_{1n}], \dots, [expr_{m1}, \dots, expr_{mm}]]$. In the second form, *L* may consist of a single list of *mn* elements. In the third form, the elements are computed using the two-dimensional function *f* as $[[f(1, 1), \dots, f(1, n)], \dots, [f(m, 1), \dots, f(m, n)]]$. In the last form, the elements are left undefined. Elements of a matrix can be accessed with *mat*[*i*, *j*].

```
mat:=array(1..m, 1..n, L);†
mat:=array(1..m, 1..n);†
```

Alternative way of defining matrices. *L* is a list of lists as in the `matrix` command.

```
mat:=diag(var1, ..., varn);
mat:=diag(mat1, ..., matn);
```

The first form defines *mat* to be a diagonal matrix with the diagonal elements being var_1, \dots, var_n . The second form defines *mat* to be a block diagonal matrix with mat_1, \dots, mat_n being the matrix blocks.

<code>vec[i] := expr ; †</code>	Individually assign values to elements of an array or matrix.
<code>mat[i,j] := expr ; †</code>	
<code>extend(mat, m, n) ;</code>	Enlarges <i>mat</i> by adding <i>m</i> additional rows and <i>n</i> additional columns. In the second form, all new elements are initialized to <i>expr</i> .
<code>extend(mat, m, n, expr) ;</code>	
<code>augment(mat₁, ..., mat_n) ;</code>	Defines a new matrix that is obtained by stacking <i>mat</i> ₁ through <i>mat</i> _n (which must all have the same number of rows) side by side.
<code>row(mat, i) ; col(mat, i) ;</code>	The first form extracts the <i>i</i> th row (or column) of <i>mat</i> as a vector. The second form extracts rows (or columns) <i>i</i> through <i>j</i> as a sequence of vectors.
<code>row(mat, i..j) ; col(mat, i..j) ;</code>	
<code>submatrix(mat, i..j, k..l) ;</code>	The first form extracts rows <i>i</i> through <i>j</i> and columns <i>k</i> through <i>l</i> from <i>mat</i> as a submatrix. The second form assigns the (<i>k,l</i>)th element of the submatrix to be the (<i>i_k,j_l</i>)th element of <i>mat</i> .
<code>submatrix(mat, [i₁, ..., i_n], [j₁, ..., j_m]) ;</code>	
<code>copyinto(mat₁, mat₂, i, j) ;</code>	Copies <i>mat</i> ₁ into the larger <i>mat</i> ₂ starting at index position (<i>i,j</i>) of <i>mat</i> ₂ . New matrices can be created from existing ones by using <code>extend</code> , <code>submatrix</code> and <code>copyinto</code> .
<code>matadd(var₁, var₂) ;</code>	Matrix or vector addition.
<code>evalm(var₁ + var₂) ; †</code>	
<code>multiply(mat₁, mat₂) ;</code>	Matrix-matrix or matrix-vector multiplication. When performing matrix arithmetic using <code>evalm</code> , the special notation <code>&*()</code> can be used to signify an identity matrix of appropriate order (e.g., <code>evalm(A - lambda*&*())</code> computes $\mathbf{A} - \lambda\mathbf{I}$.)
<code>evalm(mat₁ &* mat₂) ; †</code>	
<code>scalarmul(mat, expr) ;</code>	Scalar multiplication of matrix or vector <i>mat</i> .
<code>evalm(expr*mat) ; †</code>	
<code>mulcol(mat, i, expr) ;</code>	Multiply column or row <i>i</i> with <i>expr</i> .
<code>mulrow(mat, i, expr) ; †</code>	
<code>evalm(mat^n) ;</code>	The <i>n</i> th power of a matrix.
<code>inverse(mat) ;</code>	Inverse of a matrix.
<code>evalm(1/mat) ; †</code>	
<code>evalm(mat^(-1)) ; †</code>	
<code>adjoint(mat) ; det(mat) ;</code>	Adjoint, determinant, trace, transpose and hermitian transpose of a matrix.
<code>trace(mat) ; transpose(mat) ;</code>	
<code>htranspose(mat) ;</code>	
<code>linsolve(A, b) ;</code>	Solve the matrix equation $\mathbf{Ax} = \mathbf{b}$.
<code>leastqsrs(A, b) ;</code>	Solve an over-determined set of equations $\mathbf{Ax} = \mathbf{b}$ in the least-squares sense (i.e., such that $\ \mathbf{Ax} - \mathbf{b}\ $ is minimized.
<code>gausselim(mat) ;</code>	Gaussian elimination.

<code>rank(mat);</code>	Rank of a square matrix.
<code>norm(mat, normname);</code>	Matrix or vector norm. The default norm if <i>normname</i> is not specified is the infinity norm. Typically <i>normname</i> is 1, 2, 'infinity' or 'frobenius', but for vectors it may also be any positive integer. For the length of a vector <i>normname</i> should be 2.
<code>cond(mat);</code>	Condition number of a square matrix ($\ \mathbf{A}\ \ \mathbf{A}^{-1}\ $).
<code>singularvals(mat);</code>	Singular values of a matrix.
<code>rref(mat);</code>	Reduced row echelon form of a matrix.
<code>gaussjord(mat);</code>	
<code>eigenvals(mat);</code>	The first form finds the eigenvalues of <i>mat</i> . The second form finds the eigenvalues of the generalized problem $\mathbf{Ax} = \lambda\mathbf{Bx}$.
<code>eigenvals(A, B);</code>	
<code>eigenvects(mat);</code>	Eigenvectors. Eigenvectors of the generalized problem cannot be obtained directly.
<code>charmat(A);</code>	Finds the characteristic matrix of \mathbf{A} (i.e., $\lambda\mathbf{I} - \mathbf{A}$), and the characteristic polynomial $ \lambda\mathbf{I} - \mathbf{A} $.
<code>charpoly(A);</code>	
<code>exponential(A, t);</code>	Matrix exponential $e^{\mathbf{A}t}$.
<code>hessian(expr, vec);</code>	Hessian matrix of <i>expr</i> with respect to vector or list <i>vec</i> .

11. Linear Programming

<code>with(simplex):</code>	Load the Simplex algorithm library. This must precede any command requiring this library.
<code>feasible({constraint₁, ..., constraint_n}, option);</code>	Determine if the set of linear constraints can yield a feasible solution. Returns <code>true</code> if feasible and <code>false</code> if infeasible. Constraints are specified using <code><=</code> , <code>>=</code> or <code>=</code> . <i>Option</i> may be specified as <code>NONNEGATIVE</code> to ensure that all the solution variables are nonnegative, or alternatively, these non-negativity constraints may be listed in the set of constraints.
<code>maximize(expr, {constraint₁, ..., constraint_n}, option);</code>	Maximize the linear objective function <i>expr</i> subject to the linear constraints. <i>Option</i> is same as for <code>feasible</code> .
<code>minimize(expr, {constraint₁, ..., constraint_n}, option);</code>	Minimize the linear objective function <i>expr</i> subject to the linear constraints. <i>Option</i> is same as for <code>feasible</code> .
<code>standardize({constraint₁, ..., constraint_n}, option);</code>	Standardizes all the constraints to have the inequality <code><=</code> .

12. Statistics

Maple's statistical library typically operates on "statistical matrices." A statistical matrix is simply a matrix whose columns consist of observations of a variable, and whose first row contains the name of the variables (keys). The matrix

$$\begin{bmatrix} x & y & z \\ 1.2 & 5.1 & 3.1 \\ 2.2 & 4.4 & 1.1 \end{bmatrix}$$

is an example of a statistical matrix in which x , y and z are keys, and each of these have two observations. In the following summary, *smat* is used to signify a statistical matrix.

<code>with(stats):</code>	Load the statistics library. This must precede any command requiring this library.
<code>putkey(mat, [key₁, ..., key_n]);</code>	Converts the standard matrix <i>mat</i> to a statistical matrix by inserting a row at the beginning containing the keys.
<code>getkey(smat);</code>	Returns a list $[key_1, \dots, key_n]$ which is the key of the statistical matrix <i>smat</i> .
<code>addrecord(smat, [a₁, ..., a_n]);</code>	Adds a row of data values to the statistical matrix <i>smat</i> .
<code>evalstat(smat, newkey₁=expr₁, ..., newkey_n=expr_n);</code>	Computes new columns of data in <i>smat</i> corresponding to the specified new keys using the expressions specified in terms of existing keys.
<code>average(x);</code>	Average of data. In the first form, x may be a list, array or matrix, and for matrices, the average of each column is computed. The second form uses a simple sequence of data values. In the third form, the average of the columns associated with the specified keys are computed.
<code>average(x₁, ..., x_n);</code>	
<code>average(smat, key₁, ..., key_n);</code>	
<code>median</code>	Median, standard deviation and variance of data. These commands may take one of the three forms shown for <code>average</code> .
<code>sdev</code>	
<code>variance</code>	
<code>correlation(smat, key₁, key₂);</code>	The first form computes the correlation coefficient for the data in the columns of <i>smat</i> associated with the keys key_1 and key_2 . The second form uses data from two lists.
<code>correlation([x₁, ..., x_n], [y₁, ..., y_n]);</code>	
<code>covariance(smat, key₁, key₂);</code>	Similar to <code>correlation</code> , except that the covariance is computed.
<code>covariance([x₁, ..., x_n], [y₁, ..., y_n]);</code>	
<code>linregress(smat, ykey = xkey);</code>	Returns the list $[a, b]$, in which a is the intercept and b the slope of the linear regression equation $y = a + bx$. In the first form the data is given in the statistical matrix <i>smat</i> , and <i>ykey</i> and <i>xkey</i> are the keys for the dependent and independent variables, respectively. In the second form the data is specified in the two lists.
<code>linregress([y₁, ..., y_n], [x₁, ..., x_n]);</code>	
<code>multregress(smat, ykey = [xkey₁, ..., xkey_n], const);</code>	Returns a list of coefficients $[a_0, \dots, a_n]$ of the multiple linear regression equation $y = a_0 + a_1x_1 + \dots + a_nx_n$. <i>ykey</i> , <i>xkey₁</i> , ..., <i>xkey_n</i> are keys of the statistical matrix <i>smat</i> . The second form omits the constant a_0 in the regression equation.
<code>multregress(smat, key₁ = [key₂, ..., key_n]);</code>	

```
regression(smat, ykey = f(xkey1,  
..., xkeyn)) ;
```

Fits a generalized regression equation $y = f(x_1, \dots, x_n)$ that is linear in the unknown coefficients (e.g., $y = a_0 + a_1x_1^2 + a_2x_2^3$). The equation is specified using the keys of the statistical matrix *smat* and coefficients with arbitrary names.

```
statplot(smat, ykey*xkey) ;  
statplot(smat, ykey = f(xkey)) ;
```

Plots 2-D data and specified equation on top of each other. The first form is used to specify the X and Y axes. The second form is typically used after `regression` (which assigns values to the coefficients in the equation $f(xkey)$) to obtain the plot.

```
ChiSquare(F, v) ;  
Fdist(F, v1, v2) ;  
StudentsT(F, v) ;
```

Returns the value *x* that has the specified distribution value *F*. The remaining parameters are those of the distribution. The results of these commands are equivalent to the tables found in most statistics books.

```
Exponential( $\lambda$ , x) ;
```

Returns the value of the exponential distributions with parameter λ evaluated at *x* (i.e., $\lambda e^{-\lambda x}$).

```
Ftest(x, v1, v2) ;
```

Computes $1 - F_{v_1, v_2}(x)$, where *F* is the F distribution.

```
N(x, m,  $\sigma^2$ )  
Q(x) ;
```

Areas under the Normal distribution. The first form computes the area under $N(m, \sigma^2)$ to the left of *x*. The second form computes the area under $N(0,1)$ to the right of *x*.

```
RandBeta(a, b) ;  
RandChiSquare(v) ;  
RandExponential( $\lambda$ ) ;  
RandFdist(v1, v2) ;  
RandGamma( $\lambda$ ) ;  
RandNormal(m,  $\sigma$ ) ;  
RandPoisson( $\lambda$ ) ;  
RandStudentsT(v) ;  
RandUniform(a..b) ;
```

Generates random numbers from the specified distributions. The arguments for each command are the parameters of the distribution. In all commands an optional last argument *n* may be specified signifying the number of digits required in the generated random number (e.g., `RandNormal(0, 1, 8)`).

13. The Maple Programming Language

Maple's programming language has features that are useful when performing repetitive calculations, conditional execution, defining functions and subroutines, etc. Some of the more commonly used features of this language are summarized here.

```
while condition  
do  
    statement sequence  
od ;
```

The *statement sequence* is executed while *condition* is true. The statements may be written all in one line or split over several lines for clarity. e.g., The statements `i:=0: while i <= 10 do i^2 i:=i+2; od; compute i^2 for $i = 0, 2, 4, \dots, 10$.`

```
for i from start by change to finish  
do  
    statement sequence  
od ;
```

The *statement sequence* is executed for a sequence of values of *i*. e.g., The statements `for i from 0 by 2 to 10 do i^2 od; compute i^2 for $i = 0, 2, 4, \dots, 10$.`

```

if condition then
    statement sequence 1
else
    statement sequence 2
fi;

```

Conditional execution with if-then-else blocks. The statements `if $x \leq 2$ then x^2 else x^4` computes x^2 if $x \leq 2$ and x^4 otherwise.

```

if condition 1 then
    statement sequence 1
elif condition 2 then
    statement sequence 2
    :
    :
elif condition n then
    statement sequence n
else
    default statement sequence
fi;

```

Complex conditional evaluation according to one of several conditions. The statements `if $x \leq 2$ then x^2 elif $x <= 4$ x^3 else x^4` computes x^2 if $x \leq 2$, x^3 if $2 < x \leq 4$, and x^4 otherwise.

```

name := proc(parameter sequence)
    local variable sequence;
    statement sequence
end;

```

Defines a Maple procedure (subroutine) named *name*. When local variables are defined, they are treated as local to the procedure and different from any global variables of the same name. e.g., The statements `max2 := proc (a, b) if a < b then b else a fi end;` defines a procedure to compute the maximum value of two arguments and `max2(4, 5);` returns 5. Maple, however, has an internal function `max` that can be used for this purpose.

14. Reading and Writing Files, and Printing

When working with large amounts of numeric data, it is often necessary to read/write data from/to files. When working on large amounts of symbolic manipulation, it is often desirable to save the current state or intermediate variables to a file for continued manipulation at a later time. In order to communicate with others it may be useful to save a transcript of the current session. Symbolic results obtained through Maple may need to be coded into a Fortran or C program. Maple commands are available to facilitate such tasks.

```
readdata('fname', type, n);
```

Reads data from the file *fname*. The optional argument *type* may be `integer` or `float` (real). The integer *n* specifies how many columns are to be read from each line of the file (default is 1). The column of data in each line of the file must be separated by a space or tab. Each line from the file is read as a list, and the entire data is a list of these lists. An array can be easily defined using data in a file by

```
array(readdata('fname', n));
```

```

save 'fname';
save var1, ..., varn, 'fname';
read 'fname';

```

The first form saves the state of the current Maple session to *fname*, while the second form saves only the named variables. The `read` command is used to read the saved file into Maple. If *fname* ends with `.m` then the file is saved in Maple internal format, otherwise Maple language format is used.

In the X-Windows interface, the current session can be saved in an internal format with the *File/Save* or *File/Save As* menu options (a `.mws` extension should normally be used), and in text format with the *File/Export as Text* menu option. Saved sessions may be read with the *File/Open* menu option.

```
fortran(expr, option);
fortran(expr, filename='fname',
      option);
```

The first form writes the expression in FORTRAN 77 format to the screen, while the second form writes it to the file *fname*. If *expr* is a list, then one line of code is written for each item in the list. The *option* may be `optimized`, in which case several lines of code may be written for each expression in order to minimize the number of arithmetic operations in the generated code.

```
readlib(C):
C(expr, option);
C(expr, filename='fname',
  option);
```

These commands are used to write *expr* in C format. The `readlib(C):` command must precede the first `C` command in the current session. As with the `fortran` command, *option* may be `optimized`. If it is necessary to use both the `C` command and a variable named `C` then the command should be renamed with, say, `Ctrans:=readlib(C):`.

Files saved in text format can be printed on any line printer. In the X-Windows interface the current work sheet can be saved in PostScript format using the *File/Print* menu option. The file can be sent to a PostScript printer using the UNIX `lp` command.

15. Useful Miscellaneous Commands

```
alias(eqn1, ..., eqnn);
```

Assign short aliases to commonly used expressions or long function names. For example, after `alias(J=BesselJ);` `J(0,-x)` is equivalent to `BesselJ(0,-x)`. Maple also translates any result involving `BesselJ` to `J`. To un-assign an alias use `alias(var=var)`. For example, `I` has been defined in Maple to be an alias for `sqrt(-1)`, so in order to define `j` as the unit imaginary number use `alias(I=I, j=sqrt(-1))`.

```
readlib(history):
history():
```

Automatically assigns each result to the global variables `01`, `02`, etc. Previous results can be recalled using the variable names. This allows for a more comprehensive recall mechanism than the standard `"`, `"`, and `"` notations.

```
readlib(showtime):
showtime():
```

Similar to the `history` command, except that in addition to assigning each result to a variable the time taken to compute each result is also displayed.

```
anames();
```

Displays a list of defined variables and procedure names.

```
assigned(name);
```

Returns the value `true` if the variable or procedure *name* has been assigned a value, otherwise returns `false`.

```
interp([x1, ..., xn], [y1, ..., yn],
      var):
```

Computes the polynomial of order $n-1$ in *var* that passes through the points $(x_1, y_1), \dots, (x_n, y_n)$.

16. Initialization File and Customizing User Interface

Commonly used libraries/commands can be automatically loaded/executed by placing them in a personal initialization file. The initialization file is named

- `.mapleinit` in your home directory for Unix systems
- `MAPLE.INI` in your current directory for DOS systems
- `MapleInit` in your System folder for Macintosh systems.

It is advisable to end all statements in the initialization file with the colon so that they are not echoed back at startup. It is convenient to place the following commands in the initialization file if the linear algebra package is used often:

```
with (linalg):  
mysubs := proc(x,y) subs(y,x) end:  
myint := proc(x,y) int(x,y) end:  
mydiff := proc(x,y) diff(x,y) end:
```

The procedures `mysubs`, `myint` and `mydiff` can be used with the `map` command to perform substitutions, integrations and differentiations of elements of matrices.

References

- Abell, M. L., and Braselton, J. P. (1994a). *Differential equations with Maple V*. AP Professional, Cambridge, MA.
- Abell, M. L., and Braselton, J. P. (1994b). *Maple V by example*. AP Professional, Cambridge, MA.
- Abell, M. L., and Braselton, J. P. (1994c). *The Maple V Handbook*. AP Professional, Cambridge, MA.
- Char, B. W., Geddes, K. O., et al. (1991). *Maple V library reference manual*. Springer-Verlag, New York.
- Char, B. W., Geddes, K. O., et al. (1992). *First leaves: A tutorial introduction to Maple V*. Springer-Verlag, New York.