# Mathematica Primer

Version 1.0
29 September 1992

Richard M. Murray

Division of Engineering and Applied Science
California Institute of Technology
Pasadena, California 91125
murray@design.caltech.edu

# 1. Mathematica Primer

Mathematica is a comprehensive software system for mathematical computation. Mathematica can be used to perform numerical calculations (including more than 400 functions), symbolic computations (polynomial factorization, power series expansion, etc.) and graphics (both two and three dimensional). The general reference for Mathematica is the book *Mathematica: A System for Doing Mathematics by Computer* by Stephen Wolfram.

## 1.1 Starting up Mathematica

Mathematica 2.0 is available on the following machines:

- CCO sun cluster: punisher, sandman; remote access via X11
- CCO NeXT machines: most machines, notebook interface
- CCO Macintoshes: all machines, notebook interface
- Research machines: avalon and robby (limited processes)

See the CCO Reference Guide for up to date information on which versions of Mathematica are available and on which machines.

To start Mathematica on Unix machines, use the command `math`. The response should look something like:

```
Mathematica 2.0 for SPARC
Copyright 1988-91 Wolfram Research, Inc.
 -- X11 windows graphics initialized --

In[1]:=
```

The prompt `In[1]:` is Mathematica's way of asking for input. All input lines are numbered for later recall. Output from Mathematica is preceded by the string `Out[N]:`, where `N` is the output line. For example, we can ask Mathematica to add two numbers and print the result:

```
In[1]:= 6+7

Out[1]= 13

In[2]:=
```

Mathematica responds by printing the output and requesting a new line of input.

To exit from Mathematica, type `Quit`.

## 1.2 Numerical calculations

Numbers in mathematica can be either integers or floating point numbers. An integer is represented as a sequence of digits without a decimal point. All integer calculations are kept as integer calculations for as long as possible (this includes ratios of integers). Floating point numbers are specified by using a decimal point:

```
17.
17.0
```

Exponents are specified by using standard mathematical operations; Fortran and C floating point formats are not supported:

```
2 * 10^17
9.8 * 10^5
```

Arbitrary precision arithmetic is supported, but machine precision floating point is used by default to increase performance.

All of the standard arithmetic functions are supported by mathematica:

| | | | | | |
|---|---|---|---|---|---|
| addition | + | division | / | grouping | () |
| subtraction | - | exponentiation | ^ | | |
| negation | - | multiplication | * | | |

An important difference between Mathematica and most symbolic and programming languages is that two expressions separated by white space are *multiplied*. Thus `a b` is the same as `a * b`. This mimics the usual written notation, but can be quite confusing for novices. This partially explains why Mathematica does not use Fortran or C notation for floating point numbers. Mathematica interprets `10e-5` *symbolically*

```
In[2]:= 10e-5

Out[2]= -5 + 10 e
```

Notice that multiplication was implied even though no spaces were explicitly present.

Mathematica also provides over 400 mathematical functions. The arguments to a Mathematica function are enclosed in square brackets. All mathematica functions begin with capital letters

```
Sqrt[x] calculate the square root of x
Sin[x], Cos[x] trigonometric functions
```

To find out what a function does, type `?fcn`. Wildcards are allowed, in case you don't know quite what you are looking for:

```
ArcCos  ArcCot  ArcCsc  ArcSec  ArcSech ArcSin  ArcSinh ArcTan
ArcTanh ArcCosh ArcCoth ArcCsch

In[3]:= ?ArcCsch
ArcCsch[z] gives the inverse hyperbolic cosecant of the
```

```
complex number z.
```

Mathematica also supports the use of symbolic constants. The following constants are available:

```
Pi pi, approx 3.1415
E e, approx 2.72
I imaginary unit Sqrt[-1]
Infinity positive infinity
ComplexInfinity infinite magnitude, undetermined phase
```

Constants are left in symbolic form for as long as possible (this preserves precision). For example, `2 + Pi` evaluates to `2 + Pi`. To force a conversion to floating point numbers, use the function `N`: `N[x]` converts `x` to a numeric expression (if possible). An optional argument allows the precision of the floating point number to be specified:

```
In[3]:= 2 + Pi

Out[3]= 2 + Pi

In[4]:= N[2 + Pi]

Out[4]= 5.14159

In[5]:= N[2 + Pi, 40]

Out[5]= 5.141592653589793238462643383279502884197
```

## 1.3 Building up calculations

Mathematica has a history mechanism to allow previous results to be recalled and reused:

```
%       evaluates to the previous output
%n      returns Out[n]
```

This mechanisms can be used to continue calculations from one line to the next.

In addition to the history mechanism, variables can be used to store the results of calculations. The syntax for variable assignment is

```
variable = expression
```

This causes Mathematica to evaluate the expression to the right of the equals sign and assign the resulting value to the variable. Once a variable is assign a value, it can be used in expressions and will evaluate to its value:

```
In[6]:= x = 5

Out[6]= 5
```

```
In[7]:= x + 9

Out[7]= 14
```

Variable names can be any combination of letters and digits, as long as they start with a letter. To suppress the output of a variable assignment, terminate the line with a semicolon. Assigning a variable the value . clears the value of the variable.

Lists, vectors, matrices, and arrays are created in Mathematica using curly braces. A list is just an ordered collection of elements. The elements may themselves be lists or any other expression. This is used to create matrices and more complicated arrays:

```
{a, b, c}                 a list of depth 1
{{a, b, c}, {d, e, f}}    a list of depth 2 (matrix)
```

Matrices are entered by *rows*. The function `MatrixForm` can be used to print out a matrix in row/column format:

```
In[8]:= MatrixForm[{{a,b,c}, {d,e,f}}]

Out[8]//MatrixForm= a   b   c

                    d   e   f
```

To access the items of a list, double square braces are used (remember that single square braces were used for functions).

```
list[[i]]              ith element of a list
list[[i, j]]           ith row, jth column
list[[i]][[j]]         another way to get ith row, jth column
```

Mathematica does not require any structure within lists. The elements of a list can be any expression, including other lists of different lengths. Of course, in order to perform certain operations (such as matrix multiplication), the list must have a certain structure.

## 1.4 Operations on lists

Most Mathematica functions operate on the individual elements of a list. This property is called `Listable`.

```
Sqrt[{a1, a2}]         returns {Sqrt[a1], Sqrt[a2]}
{a1, a2} + {b1, b2}    returns {a1 + b1, a2 + b2}
a * {a1, a2}           returns {a * b1, a * b2}
{a1, a2} * {b1, b2}    returns {{a1 b1, a1 b2}, {a2 b1, a2 b2}}
```

Note that multiplication between lists gives the *outer* product.

Matrix and vector multiplication is implemented using the dot product operator, represent by a period. The dot product between two vectors gives the usual inner product. The dot product between a matrix and a vector is used for multiplying a vector times a matrix. Similarly, the dot product between two matrices corresponds to the usual matrix product. The dot product returns an error if the operands do not have the proper dimensions.

```
vec1 . vec2 returns a scalar
mat1 . vec1 returns a vector (list)
mat1 . mat2 returns a matrix
```

## 1.5 Symbolic mathematics

The main power of mathematica is its ability to perform symbolic calculations. Variables which do not have values assigned to them can be manipulated symbolically and later evaluated if desired. The following sequences illustrates some of these capabilities:

```
In[6]:= f = (x+y)^3

                 3
Out[6]= (x + y)

In[7]:= x = 5

Out[7]= 5

In[8]:= Factor[f]

                 3
Out[8]= (5 + y)
```

Notice that `f` replaces `x` with its current value but leaves y unevaluated.

Mathematica can solve symbolic equations and return a solution in terms of one or more variables. For example, the quadratic formula, according to Mathematica is given by

```
In[15]:= Solve[x^2 + b x + c == 0, x]

                             2                            2
                -b + Sqrt[b   - 4 c]         -b - Sqrt[b   - 4 c]
Out[15]= {{x -> -------------------}, {x -> -------------------}}
                         2                            2
```

Mathematica can also perform symbolic differentiation and integration.

```
D[f, x]                 partial derivative of f with respect to x
Integrate[f, x]         indirect integral of f with respect to x
```

Example:

```
D[x^2 + x, x]            returns    2x + 1
D[Sin[x], x]             returns    Cos[x]
D[{x^2 + x, Sin[x]}, x]  returns    {2x + 1, Cos[x]}
```

## 1.6 Simplifying expressions

There are several functions available in mathematica for simplifying symbolic expressions:

```
Expand[expr]    expand all products     (x+2)^2 --> x^2 + 4x + 4
Factor[expr]    factor products         x^2 + 4x + 4 --> (x+2)^2
Simplify[expr]  reduce to simplest possible form
Together[expr]  put everything over a common denominator
```

Trigonometric expansions are performed by default in the `Simplify[]` function, but not by other simplification functions. To enable trigonometric simplification, use the `Trig->True` option:

```
Expand[expr, Trig->True]
```

## 1.7 Defining functions

Mathematica allows you to extend the functions which are available by defining your own. A function declaration in Mathematica is represented as a substitution pattern. A simple function has the form

```
name[arg1_, arg2_] := expr;
```

After the function is defined, whenever `name[x,y]` is encounted in an input expression, `expr` will be substituted with the values of `x` and `y` replacing occurrences of `arg1` and `arg2`. For example, the following function adds two expression together:

```
add[x_, y_] := x+y; define a new function
add[5, a] returns a+5
```

Note that the underscores are used only on the left hand side of a function declaration. This identifies the pattern for the function. Much more complex patterns can be specified (see the Mathematica book).

Functions which evaluate a series of statements can also be specified by using the `Module` statement:

```
name[args...] :=
  Module[
    {localvar1, localvar2, ...},
    statement1;
    :
    statementn
  ];
```

Mathematica evaluates the expressions `statement1`, ..., `statementn` sequentially and returns the value of the *last* expression. Note that there is no semicolon after the last expression. The variables `localvar1`, ..., `localvari`, can be used within the body of the Module with disturbing their previous value (if any).

Internally, Mathematica treats a function definition as a rule. The name and argument list define a *pattern* which Mathematica uses when evaluating expressions. All expression which match the pattern are replaced by the function definition, with arguments substituted appropriately. Much more complex patterns can be used to define functions which operator differently on different types of arguments or to define functions which have optional arguments with default values. Mathematica even uses patterns to store values of symbols: when a symbol occurs in an expression, it is replaced by the value of the symbol.

Mathematica has two types of assignment operators which are used to define patterns. The immediate assignment operator, `=`, evaluates the right hand side of and expression and assigns the resulting value to the symbol or pattern on the left hand side. Most simple symbol assignments (such as `x = 5`) are done using immediate assignment. The delayed assignment operator, `:=`, does not evaluate the right hand side. This is appropriate for function definitions since we do not want to evaluate the body of a function until the arguments are available. Occasionally there are reasons to use delayed assignment for symbol definitions or immediate assignments for function definitions. If you can't think of any situations in which this might be helpful, then you probably can use the assignment operators in the usual fashion.

## 1.8 Control statements

Mathematica has several functions which control the flow of execution and allow iteration.

The `If` function allows conditional execution of statements:

```
If[condition, true-statement, false-statement, other-statement]
```

`true-statement` is executed if the condition evaluates to the symbol `True`, `false-statement` is executed if the condition evalues to the symbol `False`, and `other-statement` is executed in all other cases. Sequences of statements can be used in the place of a single statement by separating the sequence with semicolons.

Several operators and functions are available which return logical values which can be used as conditions:

```
a == b return True if a and b are identical
a > b a is numerically greater than b
expr1 && expr2 logical and
expr1 || expr2 logical or
IntegerQ[expr] return True if expr is an Integer
```

Expressions which can not be evaluated to True or False are left unevaluated. Mathematica is somewhat finicky about comparing floating point numbers. For example `Pi == 3.14` will return unevaluated, but `N[Pi] == 3.14` will return false.

Loops can be executed using the `Do` and `While` statements. The `Do` function executes a sequence of statements using an iterator variable to control the number of times that the loop is executed:

```
Do[statements, {i, imin, imax}]
```

If `i` occurs in `statements`, its current value is used.

The `Table` function can be used to create lists:

```
Table[expr, {i, imin, imax}]
```

`Table` returns a list with each element of the list evaluated for the current value of `i`. This function is a much more efficient method of creating an array of values than using a `Do` loop.

## 1.9 Programs

In addition to interactive operation, Mathematica commands can be executed out of a file. This allows you to place a sequence of commands in a file for later use and evaluation (very handy for homework sets). To read a set of commands from a file, use the `<<` operator:

```
<<file.m
```

By convention, files which contain Mathematica functions end in `.m`. Commands executed from a file are not echoed on the terminal, even if they do not end with a semicolon. The exception is the last command in the file, which is the "value" of the expression `<<file.m`. (On the NeXT and Mac, the notebook front end provides a somewhat different interface for including files.)

Several commands are useful when executing programs from a file:

```
Print[args...]  evaluate args and display on terminal
(* comments *) user comments
Return[] return command to the terminal
Exit[] Exit from Mathematica
```

## 1.10 Plotting

Mathematica has a large array of graphics functions, including two- and three-dimensional plots and animation. Options allow the graphs to be tailored to individual needs. Graphs can be displayed on the screen or saved in a file (as PostScript) and incorporated into documents.

Plotting a function in Mathematica is done use the `Plot` command. `Plot` accepts two arguments: a function to be plotted and a range for the independent variable:

```
Plot[Sin[t] Cos[10t], {t, -2Pi, 2Pi}]
```

This command will cause the plot to be displayed on the screen. On most systems, the plot remains on the screen until a mouse button is clicked inside the plot. Any function can be plotted, as long as it evaluates to a numerical value when the independent variable is replaced by a number.

The output from the `Plot` command is a graphics object. The graphics object is Mathematica's representation of the plot. Graphics objects can be redisplayed using the `Show` command, which takes a graphics object as its argument. To save a plot to a file, use the `Display` command:

```
Display["file.mps", Graphics]
```

The `Display` function will create a file which can be converted into the PostScript graphics language. To convert the file into PostScript, use the `psfix` command from the system command line. On unix systems, the appropriate command looks like:

```
psfix < file.mps > file.ps
```

The output from `psfix` is conforming PostScript and can printed on any PostScript printer or included in a document.

A variant of the `Plot` command is the `ParametricPlot` command. The parametric plot command displays the x and y coordinates of each point as a function of a third parameter.

```
x = Cos[t] + 2 Cos[2t]
y = Sin[t] + 2 Sin[2t]
ParametricPlot[{x,y}, {t, -Pi, Pi}]
```

In addition to the Plot and ParametericPlot commands, the following other plotting commands are available:

```
ContourPlot     planar contour plots
DensityPlot     shaded density plot
Plot3D          three-dimensional surface plot
ListPlot        plot data given as list of (x, y) coordinates
```

See the online documentation or the Mathematica reference manual for complete descriptions.

## 1.11 Example #1: Jacobian of a mapping

To illustrate the power of Mathematica in defining symbolic functions, we define a routine to calculate the Jacobian of a mapping. Given a function f(x) which maps R^m to R^n, we wish to calculate the matrix of partial derivatives Df(x), a n x m matrix. We will call this function `Jac`. It takes two arguments: a vector of n expressions representing the function f and a list of m variable names.

```
Jac[f_, x_] :=
```

```
Table[
  Table[ D[ f[[i]], x[[j]] ], {j, Length[x]} ],
  {i, Length[f]}
];
```

Sample usage:

```
In[2]:= Jac[{x1^2 + x2^3, Sin[x1]}, {x1,x2}]

                         2
Out[2]= {{2 x1, 3 x2 }, {Cos[x1], 0}}

In[3]:= MatrixForm[%]

Out[3]//MatrixForm=                2
                         2 x1      3 x2


                         Cos[x1]   0
```

## 1.12 Learning more

This primer only touches the bare basics of Mathematica usage. Some of the other capabilities of Mathematica include:

Numerical Functions
: Mathematica includes a full range of higher mathematical functions, from elliptic integrals and complex Bessel functions to hypergeometric surfaces and integer factorization.

Symbolic Computation
: Mathematica can do many kinds of algebraic operations, including factoring, expanding, and simplifying polynomial and rational expressions. It can find algebraic solutions to polynomial equations and systems of equations. It can evaluate derivatives and integrals symbolically and find symbolic solutions to ordinary differential equations. It can derive and manipulate power series approximations and find limits.

Graphics
: Mathematica has a large array of graphics functions, including two- and three-dimensional plots and animation. Options allow the graphs to be tailored to individual needs. Graphs can be displayed on the screen or saved in a file (as PostScript) and incorporated into documents. Mathematica also allows simple animations to be created and executed.

Mathematica Packages
: Collections of functions can be combined into libraries, called packages. Standard Mathematica packages exist for performing a wide variety of computations, including vector analysis and Laplace transforms (to name two). Many functions are available for building packages including debugging, help, and error facilities.

Other references

- Stephen Wolfram: *Mathematica: A System for Doing Mathematics by Computer*, Second Edition (Addison-Wesley, 1991). *The* Mathematica reference.
- Nancy Blachman: *Mathematica: A Practical Approach* (Prentice-Hall, 1991). A tutorial introduction to Mathematica.
- Roman Maeder: *Programming in Mathematica* (Addison-Wesley, 1989). A general introduction to Mathematica programming.

# Appendix A. Exercises

1. Find the indefinite integral of tan x.
2. Write a Mathematica function to calculate the factorial of an integer.
3. Write a function which returns True if a matrix is orthogonal.
4. Write a function to create a 2x2 rotation matrix given an angle of rotation (in the plane).
5. Write a function which finds the angle of rotation corresponding to a 2x2 rotation matrix.
6. Write a function `zeroMatrix` which computes a rectangular matrix of specified dimensions filled with zeros.
7. Write a function which stacks the columns of a list of matrices together. Hint: use the `Join` function.

# Appendix B. A partial list of Mma functions

This section lists a few of the functions which are available within mathematica. The descriptions were extracted from the Mathematica online help utility.

`/.` expr /. rules applies a rule or list of rules in an attempt to transform each subpart of an expression expr.

`/;` patt /; test is a pattern which matches only if the evaluation of test yields True. lhs :> rhs /; test represents a rule which applies only if the evaluation of test yields True. lhs := rhs /; test is a definition to be used only if test yields True.

`:=` lhs := rhs assigns rhs to be the delayed value of lhs. rhs is maintained in an unevaluated form. When lhs appears, it is replaced by rhs, evaluated afresh each time.

`=` lhs = rhs evaluates rhs and assigns the result to be the value of lhs. From then on, lhs is replaced by rhs whenever it appears. {l1, l2, ...} = {r1, r2, ...} evaluates the ri, and assigns the results to be the values of the corresponding li.

`==` lhs == rhs returns True if lhs and rhs are identical.

`->` lhs -> rhs represents a rule that transforms lhs to rhs.

`<<` <<name reads in a file, evaluating each expression in it, and returning the last one. Get["name", key] gets a file that has been encoded with a certain key.

`.` a.b.c or Dot[a, b, c] gives products of vectors, matrices and tensors.

`Abs` Abs[z] gives the absolute value of the real or complex number z.

`Append` Append[expr, elem] gives expr with elem appended.

`Array` Array[f, n] generates a list of length n, with elements f[i]. Array[f, {n1, n2, ...}] generates an n1 X n2 X ... array of nested lists, with elements f[i1, i2, ...]. Array[f, dims, origin] generates a list using the specified index origin (default 1). Array[f, dims, origin, h] uses head h, rather than List, for each level of the array.

`Block` Block[{x, y, ...}, expr] specifies that expr is to be evaluated with local values for the symbols x, y, .... Block[{x = x0, ...}, expr] defines initial local values for x, .... Block[{vars}, body /; cond] allows local variables to be shared between conditions and function bodies.

`Constant` Constant is an attribute which indicates zero derivative of a symbol with respect to all parameters.

`Debug` Debug[expr] evaluates expr, allowing you to stop at certain points in some control structures, and see what's happening or run commands. Debug[expr, {f1, f2, ...}] stops only in evaluation of the functions fi (and everything they call).

`Denominator`
Denominator[expr] gives the denominator of expr.

`Det` Det[m] gives the determinant of the square matrix m.

`Dimensions`
Dimensions[expr] gives a list of the dimensions of expr. Dimensions[expr, n] gives a list of the dimensions of expr down to level n.

**Display** Display[channel, graphics] writes graphics or sound to the specified output channel.

**Do** Do[expr, {imax}] evaluates expr imax times. Do[expr, {i, imax}] evaluates expr with the variable i successively taking on the values 1 through imax (in steps of 1). Do[expr, {i, imin, imax}] starts with i = imin. Do[expr, {i, imin, imax, di}] uses steps di. Do[expr, {i, imin, imax}, {j, jmin, jmax}, ...] evaluates expr looping over different values of j, etc. for each i. Do[] returns Null, or the argument of the first Return it evaluates.

**Eigenvalues** Eigenvalues[m] gives a list of the eigenvalues of the square matrix m.

**Eigenvectors** Eigenvectors[m] gives a list of the eigenvectors of the square matrix m.

**Expand** Expand[expr] expands out products and positive integer powers in expr. Expand[expr, patt] avoids expanding elements of expr which do not contain terms matching the pattern patt.

**False** False is the symbol for the Boolean value false.

**Flatten** Flatten[list] flattens out nested lists. Flatten[list, n] flattens to level n. Flatten[list, n, h] flattens subexpressions with head h.

**For** For[start, test, incr, body] executes start, then repeatedly evaluates body and incr until test fails to give True.

**Function** Function[body] or body& is a pure function. The formal parameters are # (or #1), #2, etc. Function[x, body] is a pure function with a single formal parameter x. Function[{x1, x2, ...}, body] is a pure function with a list of formal parameters. Function[{x1, x2, ...}, body, {attributes}] has the given attributes during evaluation.

**IdentityMatrix** IdentityMatrix[n] gives the n X n identity matrix.

**If** If[condition, t, f] gives t if condition evaluates to True, and f if it evaluates to False. If[condition, t, f, u] gives u if condition evaluates to neither True nor False.

**In** In[n] is a global object that is assigned to have a delayed value of the nth input line.

**Inverse** Inverse[m] gives the inverse of a square matrix m.

**Join** Join[list1, list2, ...] concatenates lists together. Join can be used on any set of expressions that have the same head.

**LeafCount** LeafCount[expr] gives the total number of indivisible subexpressions in expr.

**Length** Length[expr] gives the number of elements in expr.

**LinearSolve** LinearSolve[m, b] gives the vector x which solves the matrix equation m.x==b.

**Listable** Listable is an attribute that can be assigned to a symbol f to indicate that the function f should automatically be threaded over lists that appear as its arguments.

**MatrixForm**
MatrixForm[list] prints with the elements of list arranged in a regular array.

**MatrixQ** MatrixQ[expr] gives True if expr is a list of lists that can represent a matrix, and gives False otherwise. MatrixQ[expr, test] gives True only if test yields True when applied to each of the matrix elements in expr.

**Module** Module[{x, y, ...}, expr] specifies that occurrences of the symbols x, y, ... in expr should be treated as local. Module[{x = x0, ...}, expr] defines initial values for x, ....

**N** N[expr] gives the numerical value of expr. N[expr, n] does computations to n-digit precision.

**Not** !expr is the logical NOT function. It gives False if expr is True, and True if it is False.

**NullSpace**
NullSpace[m] gives a list of vectors that forms a basis for the null space of the matrix m.

**Numerator**
Numerator[expr] gives the numerator of expr.

**Out** %n or Out[n] is a global object that is assigned to be the value produced on the nth output line. % gives the last result generated. %% gives the result before last. %%...% (k times) gives the kth previous result.

**ParametricPlot**
ParametricPlot[{fx, fy}, {t, tmin, tmax}] produces a parametric plot with x and y coordinates fx and fy generated as a function of t. ParametricPlot[{{fx, fy}, {gx, gy}, ...}, {t, tmin, tmax}] plots several parametric curves.

**Part** expr[[i]] or Part[expr, i] gives the ith part of expr. expr[[-i]] counts from the end. expr[[0]] gives the head of expr. expr[[i, j, ...]] or Part[expr, i, j, ...] is equivalent to expr[[i]] [[j]] .... expr[[ {i1, i2, ...} ]] gives a list of the parts i1, i2, ... of expr.

**Partition**
Partition[list, n] partitions list into non-overlapping sublists of length n. Partition[list, n, d] generates sublists with offset d. Partition[list, {n1, n2, ...}, {d1, d2, ...}] partitions successive levels in list into length ni sublists with offsets di.

**Plot** Plot[f, {x, xmin, xmax}] generates a plot of f as a function of x from xmin to xmax. Plot[{f1, f2, ...}, {x, xmin, xmax}] plots several functions fi.

**Plot3D** Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax}] generates a three-dimensional plot of f as a function of x and y. Plot3D[{f, s}, {x, xmin, xmax}, {y, ymin, ymax}] generates a three-dimensional plot in which the height of the surface is specified by f, and the shading is specified by s.

**Print** Print[expr1, expr2, ...] prints the expri, followed by a newline (line feed).

**Product** Product[f, {i, imax}] evaluates the product of f with i running from 1 to imax. Product[f, {i, imin, imax}] starts with i = imin. Product[f, {i, imin, imax, di}] uses steps di. Product[f, {i, imin, imax}, {j, jmin, jmax}, ...] evaluates a multiple product.

| | |
|---|---|
| `Quit` | Quit[ ] terminates a Mathematica session. |
| `Return` | Return[expr] returns the value expr from a function. Return[ ] returns the value Null. |
| `SetAttributes` | SetAttributes[s, attr] adds attr to the list of attributes of the symbol s. |
| `Simplify` | Simplify[expr] performs a sequence of algebraic transformations on expr, and returns the simplest form it finds. |
| `Show` | Show[graphics, options] displays two- and three-dimensional graphics, using the options specified. Show[g1, g2, ...] shows several plots combined. |
| `Solve` | Solve[eqns, vars] attempts to solve an equation or set of equations for the variables vars. Solve[eqns, vars, elims] attempts to solve the equations for vars, eliminating the variables elims. |
| `Table` | Table[expr, {imax}] generates a list of imax copies of expr. Table[expr, {i, imax}] generates a list of the values of expr when i runs from 1 to imax. Table[expr, {i, imin, imax}] starts with i = imin. Table[expr, {i, imin, imax, di}] uses steps di. Table[expr, {i, imin, imax}, {j, jmin, jmax}, ...] gives a nested list. The list associated with i is outermost. |
| `Take` | Take[list, n] gives the first n elements of list. Take[list, -n] gives the last n elements of list. Take[list, {m, n}] elements m through n of list. |
| `Times` | x*y*z or x y z represents a product of terms. |
| `Together` | Together[expr] puts terms in a sum over a common denominator, and cancels factors in the result. |
| `Transpose` | Transpose[list] transposes the first two levels in list. Transpose[list, {n1, n2, ...}] transposes list so that the nk-th level in list is the k-th level in the result. |
| `Trig` | Trig is an option for algebraic manipulation functions which specifies whether trigonometric functions should be treated as rational functions of exponentials. |
| `True` | True is the symbol for the Boolean value true. |
| `TrueQ` | TrueQ[expr] yields True if expr is True, and yields False otherwise. |
| `VectorQ` | VectorQ[expr] gives True if expr is a list, none of whose elements are themselves lists, and gives False otherwise. VectorQ[expr, test] gives True only if test yields True when applied to each of the elements in expr. |
| `Which` | Which[test1, value1, test2, value2, ...] evaluates each of the testi in turn, returning the value of the valuei corresponding to the first one that yields True. |
| `While` | While[test, body] evaluates test, then body, repetitively, until test first fails to give True. |