Mathematica Quickstart

Grant Bunker, bunker@iit.edu, Illinois Institute of Technology
updated February 7, 2016

Mathematica is a very powerful environment and programming language that has an extensive knowledge of mathematics and computer science baked-in.

It is an integrated *system* for computation, not just a bag of parts, or a collection of function calls fronted by an interpreter. The ability to freely mix symbolic computation, numerical computation, and sophisticated graphics provides great flexibility. It also accommodates a variety of programming styles (procedural, functional, object oriented, declarative...) – you can do what you want, and mix them, it’s up to you. A key to *Mathematica*'s flexibility is that *everything in it is a computable object*.

Because of *Mathematica*'s power, it can be a little daunting to learn. The purpose of this writeup is to try to minimize some of the initial bumps in the road that new users may encounter, especially those familiar with other languages. Fortunately *Mathematica* can be learned incrementally, and approached experimentally, because the input is interpreted, and error recovery is robust. It also has a very extensive help system from which to learn.

The language is designed to be flexible, but precise. Most of the functions can be called with arguments of different forms, with the input variables packaged as lists, and often lists of lists, in a hierarchical manner. Usually the options are specified as assignment rules (see below). *Mathematica* makes intelligent guesses about options if you don't specify them. You can generally specify and monitor detailed methods if you need to.

The inputs and outputs are designed to make them easy to chain together in complex way, with minimal reformatting or repackaging. Despite this, some repackaging is sometimes necessary. Usually this can simply be done using list restructuring commands (e.g Transpose, Flatten, Partition).

It is best to write *Mathematica* programs in a high level abstract manner. Although it is possible to write *Mathematica* programs that look like C, in which you iterate over indices of arrays in for loops, it is much easier to read and more efficient to execute if you use the higher level constructs (Table, Map, Total etc). If you want to write a C program, there is not much point in using *Mathematica* to do it (although *Mathematica* is easily extensible with C and Java - see Compile/Target and *J/Link*).

Mathematica provides simple and powerful methods for interactive simulation and live graphics. The Manipulate function is particularly simple and powerful for building GUIs. Simple primitives also are available for putting up interface elements.

The purpose of this brief guide is to try to lower the activation barrier for *Mathematica*'s use by those who already are knowledgeable about use of computers in science and engineering, for which other introductory books are just too tedious. Most of the content here is presented with short explanations and representative examples.

Fortunately, since this brief intro was first written in 2010, *Mathematica*'s internal documentation has been expanded substantially more or less in the spirit of this document. See “Intro for Programmers” and “How Tos” in the Help Menu under “Wolfram Documentation” at the bottom of the screen.

This notebook is intended to be read, and the executable parts executed. Just execute cells by selecting them and hitting shift-enter, or choose the menu item Evaluation/Execute Notebook to execute the whole thing at once.

Getting Help

There is extensive help built into the program in the Help menu under *Wolfram Documentation*, and even more online.
At the bottom of the screen there are useful items “Common How Tos”, “Intro for Programmers”, and descriptions of packages (which optionally add additional functionality to *Mathematica*).

The contents of the *Mathematica* Book are also readable within the help system.

To get quick help on a function, use "?" using an asterisk as wildcard (select the next cell and hit shift-enter)

```
? *Plot3D
```

get help on specific function :

```
? ListPlot3D
```

Clicking on the ">>" symbol in the bottom right corner of the (yellow background) output will take you to more detailed documentation. Once there, click on the triangles on the left to review “Details and Options”, “Examples”, “Tutorials”, and “Related Guides/Tutorials/Training Courses”.

Using ?? instead of ? gives more detailed immediate output including function options.

Notebooks

Although you can interact with *Mathematica* through a simple command line interface (and batch mode for processing bulk data), usually the best way to interact with *Mathematica* is through the Notebook Interface. The Front End is the part of *Mathematica* that handles interaction with the user; the Kernel (or multiple Kernels) manage the back-end computations. The Front End and Kernel(s) can run on different computers, and parallel computation on multiple cores and processors is supported right out of the box.

The input text, output text, and graphics live inside of "cells" within a notebook, which are graphically indicated by the cell brackets on the right side of the window.

You can add new cells between existing ones by putting the cursor there (it will change to a horizontal I-beam cursor), and clicking. You select a cell bracket by clicking on it.
Execute the cell by pressing shift-Enter.

To delete a cell after you select it, use the delete button on the keyboard.

Cells can be merged and split as desired; they can be grouped hierarchically (automatically based on their heading styles, or manually), and expanded and contracted like an outliner. See the Cell/Grouping menu for options.

Cells are of different types - they can be inert text cells (like this one), which are not executable; or input cells, which are executable; or output cells. The *Mathematica* front end gives you as much control over the output as a word processing program if you choose to use those features. You can design your own stylesheets if you want to. Entire books with extensive mathematical typesetting have been written in *Mathematica*.

You prepare instructions in an input cell (Format/Style menu) and execute them by selecting the bracket, and pressing **shift-return.** This generates an output cell containing the result. Output cell brackets look different than input cell brackets.

The output is printed by default if it's not horrendously big. If it *is* horrendously big, the system deliver an output cell that will allow you to choose how much you want to show.

If you want to suppress the output altogether (i.e. avoid printing the result), put a semicolon at the end of your input expression(s).

If you want to print something explicitly, for example within a program, use "Print". Here are some examples:

```
Sqrt[-1]
```

```
Sqrt[-1];
```

```
Print[Sqrt[-1]];
```

On input, as in normal algebra, if there is a space between symbols or numbers, multiplication is implied:

```
a = 2; b = 3;
a b
```

If you want to explicitly write a multiplication symbol that's fine too – use an asterisk:

```
1 * 2 * 3
```

Variables and functions are case-sensitive.
Built-in functions all start with capital letters. "Cos" is a built-in function;   "cos" is not.
By convention, compound names of built-in functions have the first letters of each word capitalized (so-called “CamelCase”): e.g. LaplaceTransform or HyperlinkCreationSettings.

A large number of fundamental constants are built-in e.g. E, Pi, I, Infinity, GoldenRatio, Euler Mascheroni constant (EulerGamma)..:
E^(I Pi) = -1
You can refer to them in several ways: Pi, or π.   You can get the special characters by sandwiching the name between Escape characters: π = <ESC> Pi <ESC>.
If you want to, you can also get them using escape sequences like this: \[thesymbol].
Or through the GUI, use the Basic Math Input and Basic Typesetting Palettes in the Palette/Other menu.

Natural Language Input

If you have internet access so that the Wolfram servers are accessible, you can use natural language input, based on the same technology that was built for *Wolfram Alpha*.

Typing “=” at the start of a line within a notebook will put you into natural language input mode for that cell (click on the little plus sign on the output cell to expand it):

Typing <ctrl>= allows you to use natural language input in the *middle* of an expression so you can use it for subsequent calculations

Typing two == signs together at the start of a cell puts you into a mode in which you get Wolfram Alpha output directly into your notebook in an interactive form:

igor stravinsky bela bartok maurice ravel frank zappa

Executing, Suspending, and Aborting Execution of Notebooks

If you want to run the whole notebook at once, use the menu item **Evaluation/Evaluate Notebook**, or select the outer cell bracket and hit shift-enter

To temporarily interrupt or stop execution of a notebook, Choose **Evaluation/Interrupt Evaluation** or **Evaluation/Abort Evaluation** from the menu, or use the appropriate key sequence (e.g. cmd-. on Mac, ctrl-. on Linux/Windows) that is used on your platform.

You can delete all the output by choosing the menu item Cell/Delete All Output

To clear out all variables by quitting the kernel, choose the menu item Evaluation/Quit Kernel

Mathematica has good debugging tools built-in that can be accessed through the **Evaluation** menu. You can even interrupt a running evaluation and open up a subsession to do something else, finish that, and resume.

Assign a value to a variable

One of the first things you will want to do is define variables

Use equal signs (=) to make immediate assignments (Set) :

```
a = 1; b = 2;
a + b
```

You can remove ("Unset") the assignment like this :

```
a =.
```

Clear the value, or completely remove one or more symbols by using Clear or ClearAll

```
Clear[a, b]
```

Use := to make delayed assignments, which are evaluated only when they are invoked :

```
y := Sqrt[x]
```

```
y
```

```
x = -1
```

```
y
```

In a recursive function (e.g. fibonacci sequence), or dynamic programming methods, in which for the sake of efficiency you want the function to remember previously computed values, do this:
f[x_] := f[x] = the function definition.
If you want to put your function on a restricted diet, so that it only acts on specified types of data, you can do it like this (here, it requires an Integer argument) :

```
f[x_?IntegerQ] := x^2
```

```
f[3]
```

If you feed it a real number, it returns the input unevaluated:

```
f[3.0]
```

There are many such "query" functions, all ending in "Q" :

```
? *Q
```

Heads of Expressions

Everything in *Mathematica* is an expression.   All expressions have a "Head". Examples (evaluate the following cells) :

```
Head[6]
```

```
Head[I]
```

```
Head["huh?"]
```

```
Head[{1, 2, 3}]
```

```
Head[Head]
```

```
Head[Plot[x^2, {x, 0, 1}]]
```

To see the internal representation of an object use FullForm:

```
FullForm[Sqrt[a^2 + b^2]]
```

or as a tree :

```
TreeForm[Sqrt[a^2 + b^2]]
```

Exact results

Mathematica does exact calculations whenever it can.   In contrast with nonsymbolic (numeric only) programs, strives to keep calculations as general and exact as possible.

```
1 / Sqrt[Pi]
```

Mathematica will keep the computational results exact as long as it can:

```
Exp[1]
```

Numerical calculations are not in general limited by machine precision: you can do arbitrary precision arithmetic.
Feeding a function an approximate (floating point) number will make the result an approximate number.

```
Exp[1.0]
```

Use the N function to get a numerical result to whatever precision you need, which *Mathematica* will do its best to provide:

```
N[Exp[1], 200]
```

```
ϵ = 10^(-6)
```

```
N[BesselJ[0, 1000 π + ϵ] - BesselJ[0, 1000 π], 200]
```

Mathematica keeps track of the loss of Accuracy (-log base 10 of the absolute uncertainty) and Precision (-log base 10 of *relative* uncertainty) of approximate numbers in computations.

```
1.000000001 - 1.000000000
```

```
% // Accuracy
```

```
%% // Precision
```

If desired you can specify the precision of a number by using a backtick:

```
1.001`30
```

```
Precision@%
```

Feeling listless?

let’s build a list explicitly (it’s very flexible: use any data type you want):

```
mylist =
{"albert", 3.1416, True, π, 
```

```
}
```

```
Head /@ mylist
```

To add up all the stuff in a list use Total:

```
Total[mylist]
```

Of course that didn’t make much sense but you get the idea.
To generate a list of incremental partial sums use Accumulate:

```
Accumulate[mylist] // ColumnForm
```

You can refer to specific elements in a list using double brackets :

```
mylist[[5]]
```

extract sublists using Take, or its shorthand double semicolon version:

```
Take[mylist, {1, 3}]
```

```
mylist[[1 ;; 3]]
```

Use Range to make an evenly spaced range of numbers:

```
Range[1, 10, .5]
```

Or generate a table of values algorithmically, which can then be plotted:

```
dat = Table[{n, Log[n!]}, {n, 1, 16}]; (*this makes a list of x,y pairs;  the range is contained in a list*)
TableForm[dat]
```

```
ListPlot[dat, Joined → True, Frame → True, FrameLabel → {"n", "Log[n!]"}]
```

Use Array to make a symbolic array

```
a =.
```

```
A = Array[a, {3, 3}]
```

```
A // MatrixForm
```

here is the determinant of A

```
Det[A]
```

Don't make the following mistake (here the definition of matrix is wrapped in the MatrixForm function, so Det can’t do its job, and returns the input unevaluated - it’s mysterious if you don’t see what going on).

```
A = Array[a, {3, 3}] // MatrixForm
```

```
Det[A]
```

```
FullForm[Det[A]]
```

Do it this way instead:

```
A = Array[a, {3, 3}];
A // MatrixForm
```

You can choose all the rows for column 2 this way;

```
A[[All, 2]]
```

Rules

Rules are used for transforming expressions and are often used for temporarily defining values.
They are written in the form a->b, which means the symbol a is replaced by the symbol b

To type a right arrow, type “-“ and “>” one after the other.

for example: r → Sqrt[x^2 + y^2]

Often rules are grouped together into lists like this {a->b, c->d}

To apply a rule to an expression, use /.
for example:

```
x =.; r =.; y =.;
```

```
(x + r) /. {r → Sqrt[x^2 + y^2]}
```

The options for built-in functions are usually written as replacement rules:

```
fig = Plot[Sin[x] / x, {x, -20, 20}, PlotRange → All, Frame → True, FrameLabel → {"x", "Sin[x]/x"}]
```

You can find out the options for built - in functions through the help system, or by using "Options[functionname]", or by typing ?? functionname

```
Options[ListPlot3D]
```

If you don' t explicitly set options, Mathematica will make intelligent guesses for reasonable values, depending on the nature of the information that it has. If you then want to find out the settings for those options, use AbsoluteOptions[ ] on the object it produces.

```
TableForm[Partition[AbsoluteOptions[fig], 3]] // Quiet
```

Functions and patterns

It is usually best to define functions with underscores and := like this: h[x_, y_] := Sqrt[x^2 + y^2]

As described above, the delayed assignment := means the right side is only evaluated at the time the function h is executed.

The underscore “blank” (in x_) represents a *pattern,* i.e. a placeholder that stands for any single symbol, but with the tag "x".
It means you can put in anything as a function argument.

Without the underscore, the function definition would apply specifically only to the variable called x.
For example in the following example, h is defined only for the variables named x and y:

```
Clear[h];
h[x, y] := Sqrt[x^2 + y^2];
h[a, b]
```

In this case, using the _ pattern, anything at all can be fed in as arguments to the function.

```
Clear[h];
h[x_, y_] := Sqrt[x^2 + y^2];
h[a, b]
```

```
reddot = Graphics[{Red, Disk[ ]}, ImageSize → 36];
```

```
h[I, reddot]
```

Mathematica contains an extensive set of pattern matching and manipulation constructs. See the documentation tutorials “Introduction to Patterns” and “Patterns and Transformation Rules”.
If you are more comfortable using unix regular expressions, those are supported too:

```
? RegularExpression
```

Applying Functions

If there is only one argument to a function (which could, however, be a list, or some complicated data structure), you can apply functions in at least 3 different ways:
bracket notation: f[x]
prefix notation: f@x
postfix notation: x//f

```
{Exp[x], Exp@x, x // Exp}
```

You can apply a function to each of the elements of a list by using Map, or in the shorthand form /@

```
xs = {1, 2, 3, 4};
```

```
Map[Exp, xs]
```

```
Exp /@ xs
```

It is also very useful to define "pure functions" (or “anonymous functions”) that use no named arguments; for example
(1 + #)^2 &
Here the argument is represented as # and the ampersand & indicates the end of the function definition

```
f := Exp[#] &
```

```
f /@ xs
```

fancy moves

You can do all kinds of useful and crazy stuff in *Mathematica*.
If you want to change the head of an expression to a different head, you can do it by using "Apply" or @@.

```
dat = {1, 2, 3}
```

```
Head[dat]
```

```
Apply[Times, dat]
```

```
Apply[MysteryFunction, dat]
```

```
Apply[Hypergeometric1F1, dat]
```

If you want to algorithmically generate a string that is itself interpreted as *Mathematica* code, use ToExpression.

You can also have the executable code running in the kernel execute menu items in the front end.

For example, the following cell contains a small code fragment that you may want to paste into your own code. It makes a palette, and clicking the palette buttons moves one cell forward or back; the more useful button advances one cell and then evaluates it.   If you uncomment the second line by removing "(*" and "*)", it also emits a musical sound instead of the system beep to confirm that it evaluated.   You may want to customize that, as well as the default palette position, which is chosen as 1500 pixels from the left side of your screen.
Just copy and paste the cell below into your notebooks to use it.

```
nb = SelectedNotebook[]; beep := Beep[ ];
(*snd=Sound[SoundNote["F0",.5,"Marimba",SoundVolume→.7]];beep:=EmitSound@snd;*)
beep;
mvback := SelectionMove[nb, Previous, Cell];
mvfwd := SelectionMove[nb, Next, Cell];
doit := SelectionEvaluateCreateCell[nb];
CreatePalette[
 {Button["Move Forward", mvfwd], Button["Move Back", mvback], Button["Evaluate Next Cell", {doit, mvfwd, beep}]},
 WindowMargins → {{1500, Automatic}, {Automatic, 0}}, WindowTitle → "Navigate Cells"];
mvfwd;
```

Plotting lists of values

ListPlot is set up to plot lists of x, y pairs, each as a list:

```
dat = Table[{n, n^2}, {n, 0, 5}]
```

```
ListPlot[dat]
```

A simple way to convert back and forth between xy pairs and separate x and y arrays is to use Transpose :

```
{x, y} = Transpose[dat]
```

```
x
```

```
y
```

You can customize nearly everything if you wish to. Options are generally given as lists of rules.

```
ListPlot[dat, Joined → True, PlotStyle → Thick, Frame → True,
 FrameLabel → {"n", "n*ln(n)"}, LabelStyle → {Larger, Bold}, InterpolationOrder → 2]
```

```
Options[ListPlot] // ColumnForm
```

Series

Mathematica will happily calculate Taylor and Laurent Series. Here is generating function for Legendre Polynomials

```
generaL = Series[(1 + 2 u r + r^2)^(-1 / 2), {r, 0, 5}]
```

"Normal" turns the series form (with O[something]^n) into an expression suitable for evaluating.

```
Normal[generaL]
```

Of course Mathematica knows about special functions like LegendrePolynomials

```
Table[LegendreP[n, u], {n, 0, 5}]
```

expanding around Infinity gives asymptotic expansion

```
Series[BesselK[0, r], {r, Infinity, 1}]
```

Graphics

It' s easy to build up complex forms in the *Mathematica* 8 graphics language; there are a number of 2D and 3D graphics primitives including splines and Bezier curves.

```
pts = Table[{Sin[θ], Cos[θ]}, {θ, 0, 2 π, 2 π / 5}];
```

```
disk[pt_] := Disk[pt, .6 ] (*here I'm defining a function with one argument,
 so I can map it across the list of points*)
```

```
Graphics[{RGBColor[1, .7, 0], disk /@ pts, Blue, Thick, Line[pts]}]
```

you can transform them various ways:

```
Graphics[
 GeometricTransformation[{RGBColor[1, .7, 0], disk /@ pts, Blue, Thick, Line[pts]}, RotationTransform[180 Degree]]]
```

Manipulate lets you interactively control any object with virtually no programming.

You can even plug in a game controller and Mathematica automatically will connect to its buttons and dials

```
a =.; b =.; n =.;
```

example : factor a^3 - b^3

```
Factor[a^3 - b^3]
```

now do it for parameter n (values are kept local using "With")

```
With[{n = 3}, Factor[a^n - b^n]]
```

Now vary n interactively;
Click the "+" signs by the slider to get more control/animate options

```
Manipulate[
 Row[{"n=", n, "     ", a^n - b^n, "     ", Factor[a^n - b^n]}],
 {{n, 16}, 1, 32, 1}]
```

here are the options for Manipulate , returned as a list of rules:

```
Options[Manipulate]
```

Partitioning the list of rules into 4 sub-lists makes a nicer table

```
Partition[Options[Manipulate], 4] // TableForm
```

Solve

here is a simple example - find intersection points between a circle and a line

```
x =.; y =.;
```

```
soln = Solve[{(x / 2) ^2 + y^2 == 1, 3 x + 2 y == 1}, {x, y}];
soln // ColumnForm
```

convert from rules to values:

```
pts = {x, y} /. soln
```

```
fig = ContourPlot[{(x / 2) ^2 + y^2 == 1, 3 x + 2 y == 1}, {x, -2, 2}, {y, -2, 2}];

Show[fig, Graphics[{PointSize[.03], Point[pts]}]]
```

Solve gives generic solutions

```
Solve[Tan[x] == 1, x]
```

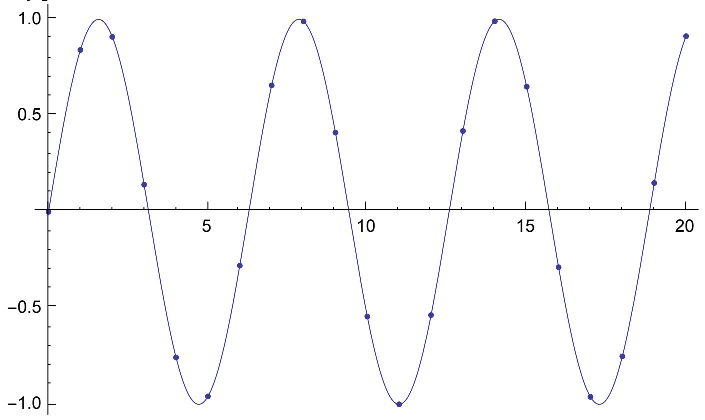
Reduce gives all solutions, which generally have conditions

```
Reduce[Tan[x] == 1, x]
```

Interpolation

Interpolation - interpolate data, returns InterpolatedFunction object that can be used more or less like regular (numeric) function

```
dat = Table[{x, Sin[x] // N}, {x, 0, 20}];
fun = Interpolation[dat, InterpolationOrder → 3]
InterpolatingFunction[{{0., 20.}}, <>]
Show[{
  ListPlot[dat],
  Plot[fun[x], {x, 0, 20}]
 }]
```



Importing and Exporting files

Mathematica has robust and powerful features for importing and exporting files in many formats (e.g. Excel)

To pick a file or path, use **Insert/File Path** from the menu. Use SetDirectory to choose the directory/folder you want to work in. Use FileNames[ ] to get a list of files, or FileNames["*.dat"] to list all the files with extension "dat".

To show the current directory/folder, use Directory[ ].
To set the directory/folder to the location of the notebook itself, use SetDirectory[NotebookDirectory[ ]].

```
SetDirectory[NotebookDirectory[]]
```

```
dat = Table[Random[ ], {5}, {5}]; TableForm[dat]
```

```
Export["test.dat", dat]
```

```
Import["test.dat", "Table"]
```

```
FileNames[ ]
```

Now do graphics:

```
fig = ArrayPlot[dat, ImageSize → 1 * 72]
```

```
Head[fig]
```

write it out as PDF (or PNG or JPEG whatever):

```
Export["fig.pdf", fig, "PDF"]
```

you can find out its metadata this way :

```
Import["fig.pdf", "Elements"]
```

```
Import["fig.pdf", "PageCount"]
```

read it back in :

```
refig = Import["fig.pdf"]
```

```
Head[refig]
```

```
Head[refig[[1]]]
```

Computable Data and Entities

it’s simplest to use the <ctrl>= input method to discover and create entities

```
chi = Chicago (city) ;
```

```
FullForm[chi]
```

```
chi@ 1 bedroom apartment fair market rent
```

```
chi["Properties"]
```

```
? *Data
```

Some other features/functions for science and engineering

This is a quick introduction, not a book, so it' s not possible to cover much of the functionality of Mathematica 10, but here are a few useful and interesting functions/capabilities.

List/matrix/tensor restructuring: Transpose, Partition, RotateRight, RotateLeft, Flatten, FlattenAt
String processing, sorting and searching
multilingual dictionaries are built-in - linguists and literary scholars take note
Select elements within expressions such as lists: Drop, Take, TakeWhile, Select, Pick
Integrated Units system: Quantity/Unit

Nest, NestList, NestWhile, NestWhileList, FixedPoint, Fold: repeated application of functions
Plotting - Plot, LogPlot,LogLogPlot, ListPlot, Plot3D, ListPlot3D, ParametricPlot, ContourPlot, ArrayPlot/MatrixPlot, GraphPlot, TreePlot, VectorPlot, ListVectorPlot, VectorPlot3D, ListVectorPlot3D,VectorDensityPlot, StreamPlot ...
Import/Export - very flexible input/output, many file types supported
NumberForm - set digits etc for output; Chop drops terms less than machine precision
Image Processing, convolution etc.

Statistical Functions - very robust symbolic and numeric capabilities (also use R/Link to incorporate R functions)
Regions - symbolically define geometric regions and use them

Extensive special functions: Bessel, Neumann, Legendre, Chebyshev, HyperGeometric, Hermite, Clebsch Gordan ...
Graphics, Graphics3D - powerful simple graphics language with robust set of primitives
BSplineFunction - represent flexible surfaces and curves

Manipulate: easy and very powerful GUI building, good for animations and simulations
Dynamic: keep objects alive throughout notebook
Block, Module: keep variables local to function

Piecewise: define piecewise function
Integrate: symbolic integration - complex plane, handles singularities
NIntegrate: numerical integration - complex plane, handles singularities
Limit
Series, Normal: Taylor and Laurent Series. Expand around Infinity to get asymptotic expansions. "Normal" truncates series, converts to normal expression
Solve: symbolic solve equations
Reduce: symbolic solve, all solutions
NSolve: numerically solve equations
FindRoot: numerical rootfinding

DSolve: differential equation solving
NDSolve: numerical solution to differential equations
DEigensystem: differential equation eigenvalue problems
FindMinimum
NMinimize: fancier methods of minimization
LinearProgramming
Fit - simple fit of data to linear combination of functions
NonlinearModelFit - function minimization and statistical analysis of the fit

Probability and Statistics - random variates, extremely flexible chocie your desired parent distribution, data type, range
Inverse - matrix inversion
SingularValueDecomposition

PseudoInverse - uses SVD

Fourier/InverseFourier - discrete fourier transforms in any dimension, no 2^n dimension restrictions; various cos etc transforms also supported
FourierTransform, LaplaceTransform, DiracDelta, KroneckerDelta, HeavisideTheta
ListConvolve - convolutions

Sparse - define sparse arrays using lists of rules; convert normal format sparse array into Sparse data type for efficiency.
Associations - data structures for flexible lookup and “big data”

Compile - compiles user's numerical valued functions to internal bytecode or machine code, for speed; even use an external c compiler
GUIKit - don't like the notebook interface? Go build your own GUI
J/Link - exposes all *Mathematica* functionality to Java, and vice-versa - simple and powerful
R/Link - opens up all of R statistics language functionality to *Mathematica*
WSTP - install your own external functions written in C or whatnot and use them as if they were built in
Run - execute system commands at shell
Database access
XML support
Web Services/SOAP
Play, EmitSound - Arbitrary waveform generation
Speech output
Signal Processing
Time Series Analysis
Control Systems, Queues, Markov models

Automatic parallelization on multiple cores or processors (ParallelTable, DistributeDefinitions, Parallelize, ParallelMap, ParallelTry...)
GPU computing with *CUDALink* and OpenCLLink

"Computable Data": everything from elementary particles to isotopes to Elements to Chemical compounds to Protein/Genomes to Geodesy to city/country/world data to mathematical data e.g. graphs, lattices, finite groups, knots. Execute ?*Data to find out more. This requires an internet connection.
There is a related extended framework for handling real world object data: Entity

This is just the tip of the iceberg. There is a massive amount of other powerful functionality in *Mathematica*/Wolfram Language that you can discover on your own.

Enjoy your exploration!