Introduction to *Mathematica*

Mathematica Conventions

What *Mathematica* Is

Getting Started

Entering Input

- **Questions Must Be Precise**
- **Evaluating Questions**
- **Function Names**
- **Brackets: ( ) and [ ] and { }**
- **Equal Signs: = and == and :=**
- **Referring to Previous Results**
- **On-Line Help**

If we know the name of a function, *Mathematica* displays a usage message for the function after we enter **?***FunctionName*. For example, if we want to see how to use the **`Simplify`** command, we type **`?Simplify`**, followed by pressing SHIFT-RET.

```
? Simplify
```

On the other hand, **?***String****** finds all *Mathematica* functions, commands, constants, and expressions that contain the string *String*. This is useful if we know a word, or part of a word, that is likely to be contained in a command.

For instance, to find all *Mathematica* functions related to logarithms we type **`?*Log*`**. (The asterisks before and after **`Log`** stand for any characters, similar to MS-DOS and Unix "wild card" characters.)

```
?*Log*
```

After locating the desired command in the above list, we evaluate **?***FunctionName* as before.

```
? ProductLog
```

The on-line help is case sensitive, so `?*log*` finds only function names that contain the word “`log`” in lower case.

```
?*log*
```

■ Loading Add-On Packages

Mathematica is an extensible system. In addition to the hundreds of built-in functions, there are many more functions defined inside *packages* which come with most copies of *Mathematica*. Packages are *Mathematica* notebooks which contain programs that teach *Mathematica* additional functions, and to make these functions available to us we must load the appropriate packages. (Many packages are included with the standard distribution of *Mathematica*, and it is a straightforward exercise to create additional packages.)

To load a package we use the `Needs` command, which takes the following form.

```
Needs["Statistics`DescriptiveStatistics`"]
```

The argument to `Needs` is a string inside quotation marks. The first part of the string is the desired package directory (in this case, `Statistics`; some of the other possibilities are `DiscreteMath`, `Calculus`, and `Graphics`—see the Help Browser or the book *Standard Add-on Packages* for a complete list of available packages and directories). After the directory name goes a *backquote* character `` ` `` (usually found on the same key as the tilde `~`, and not an ordinary single quote `'`), followed by the package name (`DescriptiveStatistics`, in this case) and another backquote. (An alternative command for loading packages takes the form ``<< Statistics`DescriptiveStatistics` ``; when using this form, it is important not to try to reload a package after it has been loaded.)

Once a package has been loaded using `Needs`, we use the functions defined in the package as if they were built in. Here is the on-line help message for the `LocationReport` function defined in the ``Statistics`DescriptiveStatistics` `` package.

```
?LocationReport
```

We can now use `LocationReport` as if it were a built-in function.

```
LocationReport[{2, 1, 7, 5, 5, 5, 1, 2, 1, 2}]
```

Packages can also be loaded by clicking the button labeled Add-ons in the Help Browser, and navigating through the package directories to the listing for the desired package. In each package’s help notebook is a command to load the package, which can be evaluated by clicking in it and pressing SHIFT-RET.

■ Warning Messages

When *Mathematica* does not understand a question, cannot complete an operation, or needs to draw attention to special considerations during the course of an evaluation, it displays one or more warning messages.

Here *Mathematica* warns that the input is incomplete: in this case, it needs a closing square bracket to be valid.

```
Sin[x
```

Here *Mathematica* warns that the name **cos** is similar to the built-in function name **Cos**; this message appears when we misspell or fail to capitalize a built-in function. It is important to remember that *Mathematica* is case sensitive.

```
cos[Pi]
```

Here *Mathematica* explains the reason it did not return an answer.

```
Integrate[1 / x, {x, -1, 2}]
```

The first part of a message (`Integrate::idiv`) in the previous example) is the name of the message. Messages can be turned off by entering **Off[***messagename***]**. For example, to turn off the message named `General::spell1` we enter the following.

```
Off[General::spell1]
```

Mathematica no longer prints warning messages about possible spelling errors. Here we misspell the command **Integrate**.

```
Integrat[x^5, x]
```

Mathematica does not print the warning message.

To turn the warning message back on, we enter the following.

```
On[General::spell1]
```

The *Mathematica* Front End

Notebooks

Cells

Word Processing

Special Characters

Two-Dimensional Input

Different Forms of Input and Output

Mathematica understands a few different forms of input and output. By default, input and output are in a form called **`StandardForm`**, which is an unambiguous two-dimensional form that uses the capitalization and bracketing conventions described earlier.

We can convert any input or output to **`TraditionalForm`**, which follows the rules of traditional typeset notation. For example, here is some **`StandardForm`** input and output.

$$\int \frac{1}{x^3 - 1} \, dx$$

To convert the input and output to traditional mathematical notation, we select the input and output cells, pull down the Cell menu, and choose TraditionalForm from the Convert To submenu. Here is the result, which uses the conventions of traditional mathematical typesetting.

$$\int \frac{1}{x^3 - 1} \, dx$$

Entering Hyperlinks

Numbered Equations and Figures

Numerics

Basic Calculator Functions

We enter arithmetic calculations in *Mathematica* just as on a calculator, followed by pressing SHIFT-RET.

Addition and subtraction are denoted by the usual symbols.

```
2 + 3.45 - 0.4
```

A space denotes multiplication, as does an asterisk **`*`** or the character ×, entered **`\[Times]`**.

```
2 * 2 × 2 (2 × 2)
```

A forward slash **/** denotes division, as does the two-dimensional form $\frac{\blacksquare}{\square}$.

```
5 / 3
```

A caret **^** or a superscript $\blacksquare^{\square}$ stands for exponentiation.

```
2^5 - 2⁴
```

Because *Mathematica* uses the standard order of arithmetic operations, it is sometimes necessary to group parts of a calculation using parentheses. Note that in **InputForm** and **StandardForm** parentheses are not used for function notation, as they are in written mathematics, or to enclose lists of elements, as they are in some programming languages.

```
2 + 4 (2 + 9.25) ^2
```

See also **NonCommutativeMultiply**

■ Exercises: Basic Calculator Functions

What is the ratio of heights between a person 5 feet, 8 inches tall and a person 6 feet, 4 inches tall?

Here is an exact value for the ratio. Note that we group the numerator and denominator with parentheses, and that we do not include units in the calculation.

```
(5 * 12 + 8) / (6 * 12 + 4)
```

Applying the function **N** to the previous exact result returns an approximate figure.

```
N[%]
```

If a copy of *Mathematica for Students* costs $139, and sales tax is 7%, what is the total cost?

The total cost is $139 plus 7% tax on $139. Note that there is no built-in percent function, so we express 7% as 0.07.

```
139 + (0.07 * 139)
```

Alternatively, we can define a unit multiplier called **percent**.

```
percent = 0.01;
```

```
139 + (7 percent * 139)
```

How many days are there in 35 years? How many hours? Minutes? Seconds? (Ignore the complication of leap years.)

```
days = 35 * 365
```

```
hours = 24 * days
```

```
minutes = 60 * hours
```

```
seconds = 60 * minutes
```

Numbers and Constants

■ Integers, Rationals, and Reals

When working with numbers, *Mathematica* returns an answer as precise as it can justified by each calculation. For this reason, it has different rules for working with exact and approximate quantities.

An approximate quantity can be identified by the presence of a decimal point; therefore 17.0 and 0.71 are approximate numbers. The integer 17 and the ratio 71/100, on the other hand, do not contain a decimal point and are considered exact numbers. In the following discussion, we refer to an approximate (non-complex) number containing a decimal point as a *real* number, an integer without a decimal point as an *integer*, and a ratio of two exact integers as a *rational* number. To *Mathematica* 7.0 is an approximate real number, and 7 is an exact integer.

When doing arithmetic with exact numbers, *Mathematica* returns an exact number. For example, the following input contains only integers, so the result is an integer.

$$2 + (3 * 5)^7$$

Mathematica leaves rational numbers (quotients of two integers, reduced to lowest terms) in explicit fractional form.

$$3 + \frac{1}{7}$$

This occurs even when there is a precise decimal equivalent.

$$\frac{2}{3} + \frac{7}{12}$$

If a calculation involves even one approximate number (a number that contains a decimal point), however, the result will be an approximate number because the uncertainty associated with the approximate number propagates through the whole calculation.

```
2   7.0
— + ———
3   12
```

■ Irrational Numbers

Irrational numbers (numbers that cannot be written as the quotient of two integers) are held in exact symbolic form. *Mathematica* allows us to use exact irrational numbers in calculations and, unless we ask, it does not automatically approximate these numbers. The following difference of two exact irrational numbers is an exact irrational number.

$\sqrt{27} - \sqrt{12}$

The following difference is also left in exact form.

$\sqrt{27} - \sqrt{13}$

If we repeat the calculation with inexact input, however, we get an inexact answer.

$\sqrt{27.0} - \sqrt{13}$

■ Mathematical Constants

Mathematica also has a number of common mathematical constants built in, defined so that we can take an approximation to whatever precision we want; the only limits being the amount of RAM installed on the computer and the amount of time we are willing to wait for an answer. Two of the more well known constants are **Pi** and **E**.

```
Pi > E
```

Pi and **E** can be entered in the special forms π and e by typing ESCpESC and ESCeeESC.

$e^{\pi i}$

As with other irrational numbers, *Mathematica* leaves constants in symbolic form unless we specifically ask for an approximation with the function **N**, described below. Here is a 250-digit approximation to π.

```
N[π, 250]
```

Mathematica also knows the standard rules for dealing with infinity, entered as **Infinity** or ∞ (ESCinfESC).

```
{1/Infinity, Infinity - 1}
```

The add-on package **Miscellaneous`PhysicalConstants`** defines a wide range of physical constants such as the speed of light, the radius of the earth, acceleration due to gravity, Avogadro's constant, and many more.

See also **N**, **Degree** (°), **GoldenRatio**, **EulerGamma**, **Catalan**, **Indeterminate**, **DirectedInfinity**, **Miscellaneous`Units`**

■ Complex Numbers

I denotes the imaginary unit $\sqrt{-1}$.

```
Sqrt[-9]
```

We can enter **I** in the special form *ⅈ* by typing ESCiiESC.

```
ⅈ^2
```

We can also use the form *ⅉ*, used in some scientific fields, by typing ESCjjESC.

```
(5 ⅉ)^2
```

As with all numbers, exact input generates exact output.

```
(3 + 19 I) / (2 - 9 I)
```

Inexact input leads to inexact output.

```
(3.0 + 19 I) / (2 - 9 I)
```

Mathematica knows the standard functions for describing and manipulating complex numbers. **Re[z]** returns the real part of *z*, **Im[z]** returns the imaginary part, and so forth.

Here is a complex number called **num**.

```
num = 1.23 + 4.56 I
```

Here is a list of the real and imaginary parts of **num**.

```
{Re[num], Im[num]}
```

Here is the conjugate of **num**.

```
Conjugate[num]
```

Here are the absolute value and approximate argument (or phase), in radians, of **num**.

```
{Abs[num], Arg[num]}
```

■ Converting between Types of Numbers

To convert from an exact number to an approximate number we use the function **N**. **N[*expr*]** returns a numerical approximation to **expr**.

$$N\left[\sqrt{27}-\sqrt{13}\right]$$

N[expr, n] does computations to at most n significant digits. Here is a 100-digit approximation to the difference between $\frac{22}{7}$ and π.

```
N[22 / 7 - π, 100]
```

The presence of the decimal point in the following example indicates the change from exact integer to approximate real number.

```
N[2]
```

There are many functions that convert an approximate number into an exact integer. **Round[x]** returns the integer closest to **x**, **Floor[x]** returns the greatest integer less than **x**, and **Ceiling[x]** returns the least integer greater than **x**.

```
{Round[3.3], Floor[3.3], Ceiling[3.3]}
```

Chop[*expr*] replaces approximate real numbers in ***expr*** that are close to zero (within 10^{-10}) with the exact integer 0. **Chop[*expr*, *tol*]** replaces approximate real numbers in ***expr*** that differ from zero by less than ***tol*** with 0.

$$\text{Chop}\left[1.012+10^{-20}\,\text{I}\right]$$

When we take the Fourier transform of a list of approximate numbers, then take the inverse Fourier transform of the result, the uncertainty in the approximate input leads to spurious imaginary parts in the answer.

```
InverseFourier[Fourier[{0, 0, 0, 1., 1., 1.}]]
```

Chop removes the small imaginary parts of the answer, and returns the original data.

```
Chop[%]
```

The function **Rationalize** converts numbers into exact rational numbers. **Rationalize[x,dx]** returns a rational number equal to **x** within a tolerance of **dx**.

Here are some successively better rational approximations to π. (**Table** is explained below in the section “Matrix and Vector Operations”.)

```
Table[Rationalize[π, 10^(-2^n)], {n, 1, 4}]
```

See also **IntegerPart**, **FractionalPart**, **NumberTheory`Recognize`**

■ Finding the Type of a Number

Although *Mathematica* does not have the concept of a type declaration (a statement such as “the variable x is a real number”), the type of a number can be found using the function **Head**. (**Head** has many other uses in *Mathematica* programming.)

```
Head[2]
```

```
Head[3.2]
```

```
Head[3 + 19 I]
```

```
Head[2/3]
```

Seel also **FullForm**, **InputForm**

■ Exercises: Numbers and Constants

Set the variable **comp** equal to the complex number $23 + 19\,i$. What is the absolute value of **comp**? What is the result of adding **comp** and its conjugate?

```
comp = 23 + 19 I
```

```
Abs[comp]
```

```
comp + Conjugate[comp]
```

Catalan's constant (used in combinatorics) is built into *Mathematica*. Knowing *Mathematica*'s naming conventions, find *Mathematica*'s name for the constant, then find an approximation to this number with 100 digits of precision.

We search for a list of functions that contain the word `Catalan`; there is only one, so *Mathematica* displays the usage message for it.

```
? *Catalan*
```

We use **N** to find a 100-digit numeric approximation.

```
N[Catalan, 100]
```

Enter the expression $2\sqrt{19} + 2$ into *Mathematica*. Take approximations to 20, 30, and 100 decimal places.

Because the number is an exact quantity, *Mathematica* leaves it in symbolic form.

```
2 Sqrt[19] + 2
```

Here we use **N** to approximate the number.

```
N[2 + 2 √19 , 20]
```

```
N[2 + 2 √19 , 30]
```

```
N[2 + 2 √19 , 100]
```

Mathematical Functions

■ Elementary Functions

All the standard elementary functions are built into *Mathematica*.

All of the trigonometric functions are available. Note that when given exact input, they return exact output, and that *Mathematica* uses the standard abbreviations for trigonometric functions.

```
Sin[Pi/4]
```

By default *Mathematica* assumes that arguments to trigonometric functions are in radians. For entering arguments in degrees there is a multiplier called **Degree**. (**Degree** can be entered in the special form ° by typing ⎋deg⎋.)

```
Cos[15 Degree]
```

Where appropriate, functions are defined for complex values.

```
Sin[6.54321 - 1.23456 I]
```

Inverse and hyperbolic trigonometric functions are also available.

```
ArcSin[1 / 2]
```

Here no exact mathematical result is known, so *Mathematica* returns the expression unevaluated.

```
Sinh[3]
```

If we give approximate input, *Mathematica* returns an approximate answer.

```
Sinh[3.0]
```

N also returns an approximate value.

```
N[Sinh[3]]
```

Logarithmic and exponential functions are also built in. **Log[***z***]** returns the natural logarithm of *z*.

```
Log[Exp[5]]
```

Log[*b***,***z***]** returns the logarithm to base *b* of *z*.

```
Log[10, 10 000]
```

See also **Tan**, **ArcTan**, **Tanh**, **ArcTanh**

Special Functions

Mathematica includes many special functions that cover a wide range of scientific subjects. In most cases, special functions are solutions to transcendental equations, integrals, or differential equations that have no elementary symbolic solution. In many cases it is advisable to consult *The Mathematica Book* to see what definition *Mathematica* uses for a particular special function, as many special functions have conflicting definitions in different fields.

In general, if a function is named after a person, the *Mathematica* name for it is *PersonSymbol*. For example, to find out if *Mathematica* knows any of the Bessel functions, we assume the word “Bessel” is in the name, and use the question mark to get a list of all such functions.

```
? *Bessel*
```

Here is the usage message for a particular Bessel function.

```
? BesselJ
```

Mathematica’s special functions are typically defined for both real and complex arguments, and know special values of the functions.

```
Gamma[1 / 2]
```

```
Gamma[1.23 + 4.56 I]
```

```
Zeta[8]
```

There are several hundred special functions built into *Mathematica*, and the Help Browser provides a convenient way to explore the different classes of functions.

See also **`Erf`**, **`Binomial`**, **`Multinomial`**, **`Beta`**, **`Factorial`**, **`HypergeometricPFQ`**, **`MeijerG`**

Matrix and Vector Operations

Creating Vectors, Matrices, and Tensors

The basic structural form in *Mathematica* is the list, which is an arbitrary collection of numbers, variables, data, and other objects, where the elements are separated by commas and enclosed within braces **`{ }`**, as in **`{1, 4, 2}`**.

Alist can have several interpretations, depending on its context. For example, the list **`{1, 4, 2}`** can be interpreted as a vector with three components, a point in 3-space, or a data set containing three measurements; and *Mathematica* automatically treats a list correctly depending on the functions used with it.

Table is one of several commands used to generate lists.

Table[*expr*, **{***$i_{\max}$***}]** generates a list of $i_{\max}$ copies of *expr*.

```
Table[again, {7}]
```

Here is a vector of ten zeros.

```
Table[0, {10}]
```

Table[*expr*, **{i,** $i_{\max}$**}]** generates a list of the values of *expr* when *i* runs from 1 to $i_{\max}$.

```
Table[i / 12, {i, 12}]
```

Table[*expr*, **{**i, $i_{\min}$, $i_{\max}$**}]** starts with i = $i_{\min}$.

```
Table[Sqrt[i], {i, 5, 15}]
```

Table[expr**, {**i, $i_{\min}$, $i_{\max}$, di**}]** uses the increment di.

```
Table[Sqrt[i], {i, 2, 5, 1 / 2}]
```

Table[expr**, {**i, $i_{\min}$, $i_{\max}$**}, {**j, $j_{\min}$, $j_{\max}$**},** … **]** generates a nested (multi-dimensional) list.

```
Table[i/j, {i,1,4}, {j,1,2}]
```

A matrix is represented in *Mathematica* as a two-dimensional list, where each sublist represents a separate row of the matrix. Here we create a matrix by using **Table** with two iterators.

```
Table[a + b, {a, 2, 5}, {b, 1, 3}]
```

We can also enter matrices by hand, being sure to enclose each sublist with curly brackets **{ }**. For example, a 3 × 3 matrix is a list containing three sublists, where each sublist is one row of the matrix.

```
matrix = {{1, 2, 3}, {3, 4, 5}, {5, 6, 7}}
```

We can also enter matrices in two-dimensional form by clicking a button in a palette, or pulling down the Input menu, choosing Create Table/Matrix/Palette , clicking the Matrix button, choosing the desired numbers of rows and columns, and clicking the OK button, after which we fill in the placeholders. Notice that the default output form of the following matrix is still a nested list of elements.

$$\texttt{matrix} = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 6 & 7 \end{pmatrix}$$

See also **Range**, **Array**, **DiagonalMatrix**, **IdentityMatrix**

■ Formatting Matrices and Tensors

`MatrixForm` displays a matrix in two-dimensional form. Here we use the matrix defined above.

```
MatrixForm[matrix]
```

`TableForm` is a generalization of **`MatrixForm`** that formats arbitrary arrays of elements. Here is the **`TableForm`** of a four-dimensional tensor (created by giving **`Table`** four iterators).

```
TableForm[Table[i + j + k - l, {i, 2}, {j, 3}, {k, 4}, {l, 5}]]
```

See also **`TableSpacing`**, **`TableAlignments`**

■ Describing and Manipulating Lists

There are many functions for describing *Mathematica* lists. For instance, we can compute the length or dimensions of a vector or matrix.

```
Length[{a, b, c, d, e}]
```

$$\text{Dimensions}\left[\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}\right]$$

`Part` extracts an element from a list according to its position. Here is the fourth element of a list. (Note that the first element of a list has position 1, not 0.)

```
Part[{a, b, c, d, e}, 4]
```

An abbreviation for **`Part`** is the following double square bracket notation, where the desired position is placed inside **`[[`** and **`]]`**.

```
{a, b, c, d, e}[[4]]
```

We can also extract elements according to their distance from the end of the list, by giving a negative position. Here is the last element of a list.

```
{a, b, c, d, e}[[-1]]
```

We extract ranges of elements from a list using **`Take`**. Here are the first three elements of a list.

```
Take[{a, b, c, d, e}, 3]
```

Here are the second through fourth elements.

```
Take[{a, b, c, d, e}, {2, 4}]
```

Drop removes elements from a list. Here we drop the first three elements from a list.

```
Drop[{a, b, c, d, e}, 3]
```

Here we drop the last three elements.

```
Drop[{a, b, c, d, e}, -3]
```

Mathematica has many built-in routines for sorting and otherwise rearranging lists.

```
Sort[{1, 3, 5, 2, 4, 6}]
```

```
Reverse[{a, b, c, d, e}]
```

See also **Extract**, **Select**, **Cases**, **Depth**, **TensorRank**, **Append**, **Prepend**, **Insert**, **Join**, **Union**, **Intersection**, **Flatten**, **Partition**, **Split**, **LinearAlgebra`MatrixManipulation`**

Linear Algebra

Here is a Hilbert matrix, where each element of the matrix is a function of its indices, created using **Table**.

```
hil = Table[(i + j - 1)^-1, {i, 3}, {j, 3}]
```

Here is the standard matrix form of **hil**.

```
MatrixForm[hil]
```

Standard operations, such as computing the determinant or eigenvalues of a matrix, are straightfor - ward.

```
Det[hil]
```

```
Eigenvalues[N[hil]]
```

We can also take the inverse, here giving it the name **inv**.

```
inv = Inverse[hil]
```

Cross[*a*,*b***]** or *a*×*b* (where × is entered ▌ **\[Cross]**) returns the vector cross product of *a* and *b*.

```
Cross[{1, 2, 3}, {a, b, c}]
```

Dot[*a*, *b***]** or *a*.*b* gives products of vectors, matrices, and tensors. Here is the dot product of the vectors **{1,2,3}** and **{a,b,c}**.

```
{1, 2, 3}.{a, b, c}
```

Here is the matrix product of the Hilbert matrix **hil** and its inverse **inv**.

```
MatrixForm[hil.inv]
```

An identity matrix is the expected result.

It is important not to confuse the matrix-multiplication operator **.** and ordinary multiplication *****. Using ***** to multiply the two matrices results in termwise multiplication.

```
MatrixForm[hil*inv]
```

We do use ordinary multiplication to multiply a scalar by a vector or matrix.

```
λ*{1, 3, 5, 7}
```

There are many other matrix functions defined in *Mathematica*. For example,

Minors[*m*, *k***]** gives a matrix consisting of the determinants of all $k \times k$ submatrices of m.

```
Minors[hil, 2]
```

$$\texttt{mp = MatrixPower}\left[\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, 10\right]\texttt{; MatrixForm[mp]}$$

See also **Transpose**, **Eigenvectors**, **Eigensystem**, **LinearSolve**, **NullSpace**, **RowReduce**, **LinearProgramming**, **MatrixExp**, **MatrixPower**, **Inner**, **Outer**, **LinearAlgebra`Orthogonalization`**

Exercises: Numerical Matrix Operations

Use `Table` to create a 5 × 5 matrix where each element has the form $\frac{1}{i^2+j^2-1}$.

```
matrix = Table[1/(i^2 + j^2 - 1), {i, 1, 5}, {j, 1, 5}]
```

Display the matrix using `MatrixForm`.

```
MatrixForm[matrix]
```

Compute the inverse of this matrix.

```
inv = Inverse[matrix]; MatrixForm[inv]
```

Display a numerical approximation to the inverted matrix using `MatrixForm`.

```
MatrixForm[N[inv]]
```

Find the determinant of the inverted matrix.

Here is the determinant of the exact matrix.

```
Det[inv]
```

Here is an approximation to the determinant.

```
N[%]
```

Working with Precision

Mathematica can work with numbers with any number of digits, and in the case of approximate numbers it maintains information about how many of the digits are significant. The number of significant digits in a number *x* is called the *precision* of *x*, and *Mathematica* knows the rules of numerical analysis for dealing with numbers with different amounts of precision.

`Precision[x]` returns the number of significant digits in the number *x*.

```
Precision[N[π, 40]]
```

`Accuracy` returns the number of digits to the right of the decimal point. Precision is a measure of the relative error of a number, and accuracy is a measure of the absolute error.

The following number has 24 significant digits.

```
Precision[1 234 567 890.123456789012345]
```

It has 15 digits to the right of the decimal point.

```
Accuracy[1 234 567 890.123456789012345]
```

For simplicity, we deal primarily with **`Precision`** in these notes.

Mathematica uses two types of approximate numbers, *machine-precision* numbers and *arbitrary-precision* numbers. Machine-precision numbers are numbers that can be calculated using a computer's hardware: most computers can directly handle numbers with up to 16 significant digits. (The number of digits that the hardware can handle is stored in the parameter **`$MachinePrecision`**.) For efficiency reasons, *Mathematica* does any calculation that contains even one machine-precision number to machine precision.

We determine a machine's precision by looking at the value of **`$MachinePrecision`**.

```
$MachinePrecision
```

The number 2.3 in the following calculation is a machine-precision number (while the other number is not), so the answer is a machine-precision number.

```
2.3 + 1.234567890123456789012345
```

```
Precision[%]
```

By default, **`N`** returns a machine-precision number.

```
N[√19]
```

```
Precision[%]
```

If a number has more than 16 (or the value of **`$MachinePrecision`**, if different) significant digits, it is called an arbitrary-precision number.

Here we ask for an approximation to $\sqrt{19}$ with 50 significant digits.

```
N[√19, 50]
```

The precision of the result is 50, making the approximation an arbitrary-precision number.

```
Precision[%]
```

Note that *Mathematica* defaults to machine precision when a number with lower precision than machine precision is generated.

```
N[√19, 4]
```

```
Precision[%]
```

Using **`SetPrecision`** we can artificially set the precision of a number to an arbitrary number of digits.

```
SetPrecision[N[√19, 4], 5]
```

```
Precision[%]
```

When using arbitrary-precision numbers, *Mathematica* does not generate more precision than is justified by the calculation. For example, adding two values with different numbers of significant digits returns a number as precise as the less precise value.

```
x1 = 54.23232323232312312624874590149643;
x2 = 34 323.98129712872939137913;
```

```
Precision[x1]
```

```
Precision[x2]
```

The precision of the sum is equal to the precision of x_2.

```
Precision[x1 + x2]
```

The precision of an exact number is **`Infinity`**.

We can approximate an exact number to as many digits as desired.

```
N[Sin[1], 250]
```

However, we cannot approximate the following machine-precision number to more than machine precision.

```
N[Sin[1.0], 250]
```

An additional way to enter a number with a known precision is in the form *nnnn*\`*p*, where *nnnn* is the number and *p* is the number of digits of precision. For example, here is an approximation to π with three digits of precision.

```
3.141`3
```

By default *Mathematica* prints only the correct digits, without the `` `p``. To explicitly see all of the information *Mathematica* has for a number, we look at the number's input form.

```
InputForm[%]
```

As with arbitrary-precision arithmetic, *Mathematica* keeps track of the number of significant digits in a result. Here we use several imprecise numbers in the calculation of the volume of a cylinder, and *Mathematica* returns an answer with the correct number of significant digits.

```
3.1416`4 * (11.1111`5)^2 (15.253545`6)
```

Similarly, *Mathematica* knows the standard rules of interval arithmetic. Here is a similar calculation performed using intervals.

```
Interval[{3.1415, 3.1416}] * Interval[{11.11105, 11.11110}]^2 *
 Interval[{15.25350, 15.25355}]
```

See also **`MachineNumberQ`**, **`SetAccuracy`**, **`PrecisionGoal`**, **`AccuracyGoal`**, **`$NumberMarks`**, **`$MaxExtraPrecision`**, **`Interval`**, **`IntervalMemberQ`**, **``NumericalMath`Microscope` ``**

Exercises: Working With Precision

Find the machine precision of a particular machine either by entering `$MachinePrecision` or by entering a machine-precision number and finding its precision.

```
Precision[3.0]
```

```
$MachinePrecision
```

Create an number with more than machine precision by entering digits by hand, and add it to a machine-precision number. What is the precision of the result? Now add `Sqrt[3]` to the original number. What is the precision of the result?

Here is a number with more than machine precision.

```
bignum = 12.32121212134312321234
```

```
Precision[bignum]
```

Here is the sum of `bignum` and a machine-precision number.

```
result = bignum + 2.3
```

The result has machine precision.

```
Precision[result]
```

Here is the sum of `bignum` and an exact quantity (a number with infinite precision).

```
result2 = bignum + Sqrt[3]
```

The result has the same precision as `bignum`.

```
Precision[result2]
```

Equation Solving

`Solve` is the basic function for solving equations in *Mathematica*. `Solve` finds solutions to equations using algebraic methods, which are often enough to get an exact numeric result.

When we solve this quadratic equation, we get an answer involving irrational numbers.

```
Solve[3 x^2 - 12 x + 10 == 11, x]
```

We get approximate solutions by using **N** on the exact solutions.

```
N[%]
```

But when we try to solve a quintic (fifth-degree polynomial) equation, we find that we do not get a numerical answer, but rather an implicit symbolic answer in the form of **Root** objects.

```
Solve[x^5 + 5 x + 1 == 0, x]
```

N still returns approximations to the implicit solutions.

```
N[%]
```

When an equation cannot be solved symbolically, **NSolve** will often find numeric solutions. (Using **NSolve** is different from using both **N** and **Solve**.)

```
NSolve[x^5 - x^4 + 12 x^3 - 11 x^2 + x - 12 == 0, x]
```

NSolve solves systems of any number of algebraic equations for an appropriate number of unknowns. Here we look for the points of intersection between an ellipse and a line. The graph allows us to make a visual approximation of the solutions.

NSolve numerically solves the system of equations.

```
NSolve[{3 x^2 + 12 y^2 == 10, 12 x - 19 y == 10}, {x, y}]
```

There are several pairs of functions in *Mathematica* whose names differ by the letter N, such as **Solve** and **NSolve**, and **Integrate** and **NIntegrate**. These "N-functions" perform their operations numerically, rather than symbolically. Although in most cases we arrive at the same result by taking a numerical approximation of a symbolic solution, *Mathematica* is in fact using a different algorithm.

FindRoot searches for a root of an equation, using a given starting point. This function uses Newton's method to find the roots of non-algebraic expressions that **NSolve** cannot solve.

Here is a graph of a transcendental expression; the roots of the expression are the points at which the graph crosses the *x*-axis.

```
Plot[Exp[x] - Sin[x], {x, -7, 2}, PlotLabel → "exp(x) - sin(x)"]
```

NSolve is unable to solve this equation.

```
NSolve[Exp[x] - Sin[x] == 0, x]
```

FindRoot returns one root of the equation, given a starting point for the search algorithm. From the graph it appears that one root is near $x = -6$, so we use -6 as one starting point. The result indicates that there is a root at $x = -6.28131$.

```
FindRoot[Exp[x] - Sin[x] == 0, {x, -6}]
```

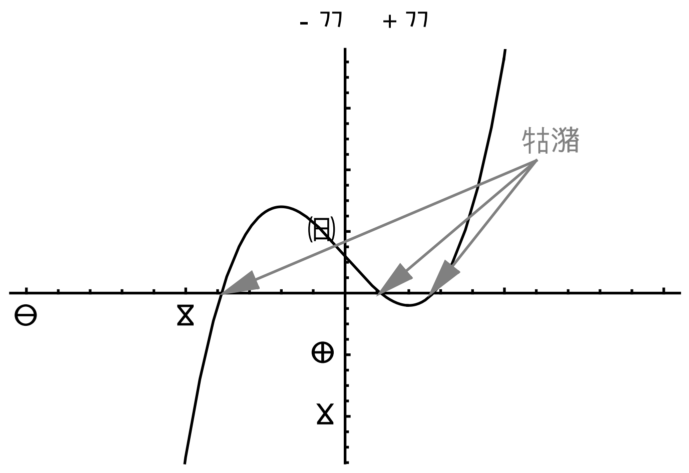
The other root appears to be near $x = -3$, so we use -3 as the second starting point.

```
FindRoot[Exp[x] - Sin[x] == 0, {x, -3}]
```

See also **`RootReduce`**, **`ToRadicals`**, **`NumericalMath`IntervalRoots``**

■ Exercises: Equation Solving

Use **`FindRoot`** to find approximations for the three roots of the function graphed below. (Hint: estimate the starting points from the graph.)



The first root is near $x = -4$, so we use -4 as the starting point for the search algorithm.

```
FindRoot[x^3 - 12 x + 12 == 0, {x, -4}]
```

The second root is near $x = 1$.

```
FindRoot[x^3 - 12 x + 12 == 0, {x, 1}]
```

The third root is near $x = 3$.

```
FindRoot[x^3 - 12 x + 12 == 0, {x, 3}]
```

Find the six roots of the equation $4x^6 - 12x^5 + 8x^4 + 3x^3 - 19x^2 + 12x - 10 = 0$.

`NSolve` will find the six roots of the equation. Remember to use **`==`** to represent equations.

```
NSolve[4 x^6 - 12 x^5 + 8 x^4 + 3 x^3 - 19 x^2 + 12 x - 10 == 0, x]
```

The result is a list of two real-valued solutions, and two conjugate complex pairs.

Find six solutions of the equation $x \sin(x) = 1$.

Here is a plot of $x \sin(x) - 1$, the roots of which are the roots of $x \sin(x) = 1$.

```
Plot[{x Sin[x] - 1}, {x, 0, 20}, PlotStyle → GrayLevel[0.5`]]
```

Note that **`NSolve`** does not find the solutions of this transcendental equation.

```
NSolve[x Sin[x] == 1, x]
```

We then use **FindRoot**. Using **Table** we can find six of the infinite number of solutions at once.

```
Table[FindRoot[x Sin[x] == 1, {x, a}], {a, π, 6 π, π}]
```

Numerical Calculus

We perform integration using the commands **Integrate** and **NIntegrate**. **Integrate** uses symbolic methods to compute the value of a definite integral.

```
Integrate[Sin[x], {x, 0, π}]
```

Integrate also works on functions with more than one variable. Here is the volume under a surface, over a square region.

```
Integrate[Sin[x] + Sin[y], {x, 0, π}, {y, 0, π}]
```

Here is a special input form for the same expression. We enter the integral sign ∫ by typing ESC int ESC or **\[Integral]**, and we must use the special differential ⅆ, entered by typing ESC dd ESC or **\[DifferentialD]**, and not an ordinary keyboard *d*.

$$\int_0^\pi\int_0^\pi (\mathrm{Sin}[x] + \mathrm{Sin}[y])\, ⅆx\, ⅆy$$

Here is a more complicated integral, taken over a nonrectangular region.

$$\int_0^{2\pi}\int_0^{x} \mathrm{Sin}[x\, y]\, ⅆy\, ⅆx$$

When calculating definite integrals involving symbolic parameters, *Mathematica* may return an answer that depends on the value of the parameters.

```
Integrate[x^n, {x, 0, 1}]
```

NIntegrate uses numerical methods to approximate the area under the specified curve in the specified domain. This method produces results in many cases where the symbolic method fails.

We cannot get an exact value for the following integral using symbolic methods.

```
Integrate[E^Sin[x], {x, -2, 2}]
```

We can get an approximation using **N**.

```
N[%]
```

NIntegrate uses numerical methods from the beginning, and returns the same result.

```
NIntegrate[E^Sin[x], {x, -2, 2}]
```

By default, *Mathematica* does all numerical calculus calculations with machine-precision numbers, but most numerical functions in *Mathematica* allow changes to this through the **WorkingPrecision** option. **WorkingPrecision->***n* causes all internal computations to be done to at most *n*-digit precision.

We can see the effect of this option by looking at an example of numerical integration. Here is an example performed with machine precision.

```
NIntegrate[1/x, {x, 1, 2}]
```

```
Precision[%]
```

Now we look at the same example with a working precision of 50 digits. Note that we are setting the precision to be used while working this problem, and not the desired precision of the result.

```
NIntegrate[1 / x, {x, 1, 2}, WorkingPrecision -> 50]
```

```
Precision[%]
```

NDSolve finds numerical solutions to systems of differential equations. Here we solve a system of differential equations, giving the list of solutions the name **sol**.

```
sol = NDSolve[{x'[t] == -y[t] - x[t]^2, y'[t] == 2 x[t] - y[t],
   x[0] == y[0] == 1}, {x, y}, {t, 0, 9}]
```

The result of this calculation is a set of two interpolating functions, essentially large sets of points that we treat as continuous functions. We then look at the solution set graphically, in two different ways. For the first plot, we plot $x(t)$ (dashed line) and $y(t)$ (gray line) on the same set of axes. (The form **/.** is explained in the next chapter.)

```
Plot[{x[t] /. sol, y[t] /. sol}, {t, 0, 9},
 PlotStyle → {Dashing[{0.015`}], GrayLevel[0.5`]}]
```

For the second plot, we plot $x(t)$ against $y(t)$.

```
ParametricPlot[Evaluate[{x[t], y[t]} /. sol], {t, 0, 9},
 AspectRatio → Automatic]
```

Here is the first interpolating function, corresponding to *x*.

```
x /. sol[[1, 1]]
```

To evaluate the interpolating function at a particular point, we append an argument inside square brackets. Here is the value of the interpolating function at 6.28.

```
%[6.28]
```

Mathematica can also solve some partial differential equations numerically, given sufficient initial conditions and ranges for all variables appearing in the system. (The form `Derivative[0,1][x][t,0]` means the derivative of the function *x*, taking the first derivative with respect to the second variable of *x*, and no derivative with respect to the first variable, evaluating the derivative at (t, 0); in traditional notation, it represents $x^{(0,1)}(t, 0)$.) The result of **NDSolve** in this case is a single two-dimensional interpolating function.

```
NDSolve[{D[x[t, u], {u, 2}] == D[x[t, u], {t, 2}], x[t, 0] == Exp[-t^2],
  Derivative[0, 1][x][t, 0] == 0}, x, {t, -5, 5}, {u, -2, 2}]
```

Here is a graph of the solution.

```
Plot3D[Evaluate[x[t, u] /. First[%]], {t, -5, 5}, {u, -2, 2},
 PlotPoints → 30]
```

See also **NSum**, **NProduct**, **Derivative**, **PrincipalValue**, **Assumptions**, **GenerateConditions**, **NumericalMath`ListIntegrate`**, **Method**, **Trapezoidal**, **Oscillatory**, **MonteCarlo**

■ Exercises: Numerical Calculus

The area under the curve $\frac{4}{1+x^2}$, between 0 and 1, is exactly π. Use **NIntegrate** to generate an approximation of π to approximately 20, 30, and 40 places of precision. (Hint: use the **WorkingPrecision** option, recalling that the precision of the result is usually 10 places less than the **WorkingPrecision**.)

Here is a graph that represents the area we wish to compute. (The graph uses the add-on package **Graphics`FilledPlot`** to fill in the areas under the curve.)

When we calculate the integral with a working precision of 30 digits, the result has around 20 significant digits.

```
NIntegrate[4/(1 + x^2), {x, 0, 1}, WorkingPrecision → 30]
```

$$\texttt{NIntegrate}\left[\frac{4}{1+x^2}, \{x, 0, 1\}, \texttt{WorkingPrecision} \to 30\right]$$

```
Precision[%]
```

Using 40 digits internally results in a value with around 30 significant digits.

$$\texttt{NIntegrate}\left[\frac{4}{1+x^2}, \{x, 0, 1\}, \texttt{WorkingPrecision} \to 40\right]$$

```
Precision[%]
```

Using 50 digits returns an answer with around 40 digits.

$$\texttt{NIntegrate}\left[\frac{4}{1+x^2}, \{x, 0, 1\}, \texttt{WorkingPrecision} \to 50\right]$$

```
Precision[%]
```

Aside from rounding in the last digit, the answers are the same as a direct approximation to π.

```
N[π, 45]
```

Other Numerical Functions

■ Random Numbers

There are many more numerical functions built into the *Mathematica* kernel or defined in the standard add-on packages that cover a large number of specialized fields.

A function useful for performing simulations is **Random**, which generates pseudorandom numbers.
Random[] gives a uniformly distributed pseudorandom real number in the range 0 to 1.
Random[*type*, *range***]** gives a pseudorandom number of the specified type in the specified range. Possible types are **Integer**, **Real**, and **Complex**. The default range is from 0 to 1, and we can specify the range {*min*, *max*} explicitly.

Here is a random number between 0 and 1.

```
RandomReal[]
```

Here is a table of random integers between 1 and 10.

```
RandomInteger[{1, 10}, 15]
```

See also **`SeedRandom`**, **`$RandomState`**, **`Statistics`ContinuousDistributions``**, **`Statistics`DiscreteDistributions``**

■ Products and Sums

Products and sums are also defined in *Mathematica*. **`Product[`***f***`, {`***i***`,`** $i_{\min}$**`,`** $i_{\max}$**`,`** *di***`}]`** evaluates the product of *f* with *i* running from $i_{\min}$ to $i_{\max}$ in steps of *di*. If $i_{\min}$ and *di* are omitted they are assumed to be 0 and 1, respectively. Multiple products are entered in the form **`Product[`***f***`, {`***i***`,`** $i_{\min}$**`,`** $i_{\max}$**`}, {`***j***`,`** $j_{\min}$**`,`** $j_{\max}$**`},`** … **`]`**.

`Product` computes an exact result.

```
Product[i, {i, 1, 10}]
```

We can enter products in the following two-dimensional form.

$$\prod_{i=1}^{10} i$$

`NProduct` uses numerical methods to find an approximate product.

```
NProduct[i, {i, 1, 10}]
```

The values computed are equal to 10 factorial.

```
10 !
```

`Sum` takes the same form as **`Product`**. **`Sum`** computes exact values, and **`NSum`** computes approximate values.

```
{Sum[n^2, {n, 2, 30}], NSum[n^2, {n, 2, 30}]}
```

Here is the two-dimensional input form for **`Sum`**.

$$\sum_{n=2}^{30} n^2$$

See also **`NumericalMath`NLimit``**, **`NumericalMath`NSeries``**

■ Optimization

Mathematica has several built-in optimization functions. **`FindMinimum`** finds a local minimum for a function, given a starting value for the search. Here is a graph of the gamma function.

```
Plot[Gamma[x], {x, 0, 4}]
```

The following command finds a local minimum for the gamma function. The result states that the local minimum is 0.885603, which occurs when x is 1.46163.

```
FindMinimum[Gamma[x], {x, 1.5}]
```

Here is a function with several local minima.The starting value we give to **`FindMinimum`** can affect the local minimum that *Mathematica* returns.

```
Plot[Sin[x] + x/5, {x, -6, 6}]
```

Here is the local minimum of $\sin(x)+\frac{x}{5}$ near $x=-2$.

```
FindMinimum[Sin[x] + x/5, {x, -2}]
```

Starting the search near $x=4$ gives a different local minimum.

```
FindMinimum[Sin[x] + x/5, {x, 4}]
```

We interpret the result as telling us that the local minimum is –0.0775897, which occurs when x is 4.51103.

`ConstrainedMin[`*f***`, {`***inequalities***`}, {`***x***`,`** ***y***`,`** … **`}]`** finds the global minimum of f in the domain specified by the linear constraints *inequalities*. The variables x, y, … are all assumed to be nonnegative. Here is the minimum of $5x-3y$, constrained by $x+2y<4$ and $x+3y>6$. The constraints $x\geq 0$, $y\geq 0$ are implied.

```
ConstrainedMin[5 x - 3 y, {x + 2 y < 4, x + 3 y > 6}, {x, y}]
```

The result states that the constrained minimum is -6, which occurs when x is 0 and y is 2.

`ConstrainedMax` works similarly.

```
ConstrainedMax[12 x + 10 y, {x + y < 4, 5 x + 3 y < 15}, {x, y}]
```

Again, the Help Browser provides an easy way to explore the many different categories of functions and algorithms.

See also **`LinearProgramming`**, **`Fourier`**, **`InverseFourier`**, **`PrimeQ`**, **`FactorInteger`**

Exercises: Other Numerical Functions

Use `Table` to make a list of the squares of the first 25 integers.

```
Table[i^2, {i, 1, 25}]
```

What is the product of these integers? (Hint: do not use `Table` or the result of the last exercise.)

```
Product[i^2, {i, 1, 25}]
```

Use `Table` to create a list of fifteen random integers between 1 and 10.

```
numbers = RandomInteger[{1, 10}, 15]
```

The commands `Max` and `Min` return the greatest and least elements of a list. What are the greatest and least numbers in the list generated above?

```
Max[numbers]
```

```
Min[numbers]
```

The sum of the numbers 1, 2, …, n ($\sum_{i=1}^{n} i$) is equal to $\frac{1}{2}n(n+1)$. Use *Mathematica* to verify that this is true for $n = 50$, 100, and 200. (Compare the results of each method.)

Here we verify the formula when n is 50.

```
Sum[i, {i, 1, 50}]
```

$$\frac{1}{2} \times 50 \times (50 + 1)$$

Here we verify the formula when n is 100, using the two-dimensional input form for `Sum`, and directly testing if the two results are equal.

$$\sum_{i=1}^{100} i == \frac{1}{2} (100)(101)$$

Here we repeat the calculation for $n = 200$.

$$\sum_{i=1}^{200} i == \frac{1}{2} (200)(201)$$

Mathematica verifies the general formula when we use a variable name for the upper limit of the summation.

$$\sum_{i=1}^{n} i$$

Symbolics

Algebra

■ Entering Symbolic Expressions

We enter symbolic expressions such as polynomials the same way we enter numeric expressions. To enter the expression $x^3 + a x^2 + b x + 1$, for example, we type the following. We use a caret `^` or a superscript to represent a power, and use a space or an asterisk to represent multiplication.

```
x^3 + a x^2 + b * x + 1
```

It is very important to remember to type a space or asterisk for multiplication. *Mathematica* interprets `b x` and `b*x` as *b* times *x*, but interprets `bx` (with no space between) as a variable with the two-letter name *bx*.

It is also important to group exponents, numerators, and denominators with parentheses: *Mathematica* uses the grouping rules and order of operations of standard arithmetic, so the following two expressions are interpreted differently.

```
{E^2 π, E^(2 π)}
```

The following two expressions are also different.

```
{1 / 2 π, 1 / (2 π)}
```

Variables, functions, and other expressions can contain special characters, such as Greek and script letters. The following is a valid symbolic expression.

$$\alpha \lambda^2 + \beta \lambda + \gamma$$

We can also use two-dimensional forms in symbolic expressions.

$$\overline{x} = \frac{1}{3}(x_1 + x_2 + x_3)$$

■ Defining Variables

We define variables in *Mathematica* by typing *name* **=** *value*, using a single equal sign. Variable names can be as long as desired, and can be any combination of letters (both upper- and lower-case) and numbers, with the restriction that a variable name cannot begin with a number.

To set the variable **newvar** equal to 15, we enter the following.

```
newvar = 15
```

After we make the assignment, every time **newvar** is used it is replaced with its value.

$$\texttt{newvar}^2 - 2^{\texttt{newvar}}$$

Variable names are case sensitive, so the following names are all different.

```
newvar + newVAR + NewVar + NeWvAr
```

Variables can have symbolic values, so the following is a valid assignment. (We keep *Mathematica* from printing the value of a variable when it is assigned by ending the line with a semicolon.)

$$\texttt{zzz} = \texttt{xxx}^2 + \texttt{yyy}^2;$$

The right-hand side of a variable assignment can be a function or program; *Mathematica* will set the value of the variable equal to the result of the right-hand side. Here we set the value of **solutionset** to be the result of solving an equation.

```
solutionset = Solve[x^2 == 2 x + 1, x]
```

We can even set a variable equal to a function name. For instance, we can set the variable **int** equal to the built-in **Integrate** function.

```
int = Integrate;
```

We can now use **int** where we would use **Integrate**.

```
int[Tan[x], x]
```

It is important to realize that variable assignments are permanent. The values of **newvar**, **zzz**, **solutionset**, and **int** will remain in memory until we quit *Mathematica*, or until we use the **Clear** command to tell *Mathematica* to forget the value of the variable. Here is another calculation that uses the value of **newvar** defined above.

```
newvar^2 + newvar - 200
```

The `Clear` function erases the value of `newvar`.

```
Clear[newvar]
```

We see that `newvar` no longer has a value.

```
newvar^2 + newvar - 200
```

Here we clear the other variables used in this section.

```
Clear[zzz, solutionset, int]
```

■ Defining Functions

Defining functions in *Mathematica* is somewhat different from writing functions by hand. The reason is that standard mathematical notation is ambiguous, while *Mathematica* requires a precise definition in order to understand a question. For example, it is unclear in traditional notation whether $q(1-p)$ means the function q evaluated at $1-p$, or q times the quantity $1-p$.

To avoid this ambiguity, in `StandardForm` and `InputForm` we type `q[1-p]` to denote the function q evaluated at $1-p$, and `q(1-p)` to denote q times $1-p$.

To define a function, say areaCircle(r), we must tell *Mathematica* that we wish areaCircle to be applicable to *any* argument r, and not just to the literal symbol r. In order to do this, we set up a *pattern* on the left side of the function definition. A pattern is a blank that can match any single argument given to a function. For example, to define areaCircle$(r) = \pi r^2$, we type the following. (Note the use of `:=` to separate the left and right sides of a function definition.)

```
areaCircle[r_] := π r^2
```

There are several things to keep in mind. The name of the function is `areaCircle`, and because it is a function, its argument goes inside square brackets. The `r_` term inside the square brackets is the pattern for the argument to `areaCircle`, which we read as "any r"; the underscore `_` is a blank that matches any single argument, and the `r` next to it is the name of the pattern, which is used to refer to the argument on the right-hand side of the function definition.

We use the function after defining it by entering a line such as the following.

```
areaCircle[10]
```

When we evaluate `areaCircle[10]`, *Mathematica* looks in its database to see if it has a definition for `areaCircle` called with one argument. Because we used the pattern `r_` in the definition of `areaCircle`, and `r_` is a pattern that matches any single argument, *Mathematica* puts the `10` into the blank called `r`, then substitutes `10` everywhere `r` appears on the right-hand side of the definition.

When we define a function, the name we give to the pattern is unimportant, except that we must use the same name on the right side of the definition. For instance, we could have defined the function **areaCircle** by entering **areaCircle[x_] := π x²**.

Arguments we give to **areaCircle** do not have to be numbers. When we type **areaCircle[19 c]**, *Mathematica* matches **19 c** with the pattern **r_**, and again substitutes **19 c** everywhere **r** appears on the right side of the definition.

```
areaCircle[19 c]
```

Here is a more complicated use of **areaCircle**.

```
1
— (areaCircle[s / 2] + areaCircle[s])
2
```

Once we define a function, we use it just as we use a built-in function. Here is a plot of **areaCircle**.

```
Plot[areaCircle[t], {t, 0, 2}]
```

Here is the derivative of **areaCircle** with respect to **z**.

```
D[areaCircle[z], z]
```

Similarly, we can define a function $\text{volumeCylinder}(r, h) = \pi r^2 h$ by entering the following.

```
volumeCylinder[r_, h_] := π r² h
```

We read the left side of the definition as "volumeCylinder of any *r* and any *h*". Here is an application of the function.

```
volumeCylinder[2, 10]
```

Like variables, function definitions are permanent; the definitions of the functions **areaCircle** and **volumeCylinder** will remain in memory until we explicitly clear the definitions using **Clear**, or we quit *Mathematica*.

```
Clear[areaCircle, volumeCylinder]
```

■ Manipulating Polynomials

Given a symbolic polynomial, *Mathematica* does not carry out much manipulation without being told to do so; that is, it makes few assumptions about the form in which we want a polynomial.

One of the operations *Mathematica* carries out automatically is putting expressions into a standard order. In **StandardForm** and **OutputForm** *Mathematica* puts constants in front, and arranges terms in order of increasing powers.

```
2 x + x^2 + 1
```

In **TraditionalForm** the reverse order is used.

```
TraditionalForm[2 x + x^2 + 1]
```

Mathematica automatically adds and subtracts like terms.

```
(1 + 2 x) + (4 x + 3)
```

$3^{(2 x - x)}$

However, *Mathematica* does not automatically expand, factor, or greatly simplify a polynomial.

$(x + 3 y - 5 z)^9$

To have *Mathematica* expand an expression, we explicitly tell it to do so. In this case, we apply the function **Expand** to the previous output (which we refer to as **%**).

```
Expand[%]
```

Similarly, *Mathematica* does not automatically simplify an expression. The most commonly used simplification command is **Simplify**, which tries a long list of transformation rules, returning the smallest equivalent expression it finds, in this case the original expression.

```
Simplify[%]
```

Simplify$\left[\left(1 - x^3\right)\left(1 + x^3 + x^6\right)\right]$

One area where it is important to recognize that *Mathematica* does not automatically simplify expressions is equation solving. *Mathematica*'s **==** construct, for instance, returns **True** only if the left and right sides of the equation are identical in *form*, and not necessarily identical in a mathematical sense.

In the following example, the left and right sides of the equation are mathematically equal, but not equal in form, so *Mathematica* returns the equation unevaluated.

$(b + x)(b - x) == b^2 - x^2$

One way to get *Mathematica* to recognize the equality is to use the **Simplify** command to transform each side of the equation into an equivalent form.

Simplify$\left[(b + x)(b - x) == b^2 - x^2\right]$

There are several ways to rearrange polynomials generated in *Mathematica*, or to pick out particular features or specific terms of a polynomial.

Using **Collect** on the expansion of $(a+b+c)^5$, we can arrange terms with respect to the variable c.

```
Collect[Expand[(a + b + c)^5], c]
```

Using **Coefficient**, we can select the coefficient of any expression in a polynomial.

```
Coefficient[Expand[(a + b + c)^5], a^3]
```

Factor represents an expression as a product of factors.

```
Factor[x^9 + 1]
```

Here we find the names of all the variables in an expression.

```
Variables[Expand[(a + b + c)^5]]
```

This is the greatest exponent in the polynomial.

```
Exponent[(1 + x) (1 - x - x^2) (1 - x), x]
```

See also **PolynomialMod**, **PolynomialQuotient**, **PolynomialRemainder**, **PolynomialGCD**, **PolynomialLCM**, **FactorList**, **CoefficientList**, **InterpolatingPolynomial**, **Fit**, **Cyclotomic**

■ Manipulating Rational Expressions

There are several commands that work exclusively on rational expressions and formulas.

Apart performs partial-fraction decomposition of a rational expression.

```
Apart[1 / (1 - x^5)]
```

Together does the opposite: it puts two or more rational expressions over a common denominator, without simplifying.

```
Together[%]
```

We can choose what part of a rational expression to expand, such as the numerator or denominator.

```
ExpandDenominator[%]
```

Cancel divides out common factors.

```
Cancel[(x^5 - 1)/(x - 1)]
```

See also **Numerator**, **Denominator**, **ExpandAll**

Manipulating Symbolic Functions

Mathematica carries out basic simplifications and computes special values of mathematical functions.

```
(Csc[x] Tan[w]) / (Cot[x] Sec[w])
```

```
BesselY[5 / 2, ξ]
```

In most cases, however, we must explicitly tell *Mathematica* to manipulate a symbolic expression. **TrigExpand** expands trigonometric expressions.

```
TrigExpand[Sin[α + β + γ]]
```

TrigToExp converts trigonometric expressions into exponential form.

```
TrigToExp[Cos[z] + I Sin[z]]
```

ExpToTrig does the opposite.

```
ExpToTrig[%]
```

FullSimplify generates the smallest possible form of an expression involving special functions. (**Simplify** works primarily on polynomial expressions.)

```
FullSimplify[Gamma[ω] Gamma[1 - ω]]
```

```
FullSimplify[Abs[z] Exp[I Arg[z]]]
```

See also **FunctionExpand**, **TrigReduce**, **ComplexityFunction**

Options

Mathematica has many options to symbolic functions that allow us to change the default assumptions used to perform a calculation. For instance, by default the **Factor** command allows only real integers in the factorization of a polynomial.

```
Factor[x^2 - 1]
```

For this reason, *Mathematica* does not extract any factors from x^2-2.

```
Factor[x^2-2]  (* use the default behavior of Factor *)
```

There may be instances, however, in which algebraic numbers should be allowed in the factorization.

Here is a list of options to `Factor`, along with their default values.

```
Options[Factor]
```

To allow algebraic numbers in the factorization of a polynomial, we use the `Extension` option.

```
?Extension
```

Mathematica now factors x^2-2.

```
Factor[x^2 - 2, Extension -> √2]
```

Similarly we can factor over complex integers by including the complex unit i in the field over which polynomials are factored.

```
Factor[x^2 + 1, Extension → i]
```

See also `GaussianIntegers`

■ *Mathematica*'s Assumptions

By default, *Mathematica* assumes that any variable or symbol that does not have an explicit value can take any complex value.

Mathematica returns the real part of any number.

```
Re[3+4 I]
```

However, *Mathematica* does not automatically simplify `Re[x + I y]` to `x`, because *Mathematica* assumes the variables `x` and `y` could have complex values.

```
Re[x + I y]
```

To instruct *Mathematica* to assume that `x` and `y` have real values, we use `ComplexExpand`. When `ComplexExpand` is applied to an expression, *Mathematica* treats all the variables in the expression as if they have real values. When `x` and `y` are assumed to be real-valued, the real part of `x + I y` is `x`.

```
ComplexExpand[Re[x + I y]]
```

Similarly, *Mathematica* simplifies $\left(\sqrt{z}\right)^2$ to z because the simplification is correct for any value of z.

```
Sqrt[z]^2
```

Mathematica does not, however, automatically simplify $\sqrt{z^2}$ to z: the simplification is valid only for nonnegative real values of `z`, and by default *Mathematica* assumes `z` can take any complex value. Here is *Mathematica*'s default behavior.

```
Sqrt[z^2]
```

However, there are ways to tell *Mathematica* to perform the simplification. The `PowerExpand` function multiplies the exponents in an expression like $\sqrt{z^2}$, as the usage message reveals.

```
? PowerExpand
```

Here *Mathematica* performs the simplification.

```
PowerExpand[√z²]
```

See also `TargetFunctions`, `Miscellaneous`RealOnly``

■ Exercises: Algebra

Add, subtract, multiply, and divide any two polynomials, simplifying the result if necessary.

Here we define two polynomials, `polyone` and `polytwo`.

```
polyone = x² + 2 x + 1;
polytwo = x³ - 3 x - 2;
```

Here are the sum and difference.

```
polyone + polytwo
```

```
polyone - polytwo
```

Here is the product.

```
polyone × polytwo
```

The result needs to be expanded.

```
Expand[%]
```

Here is the quotient.

```
polyone / polytwo
```

In this case, the quotient can be simplified.

```
Simplify[%]
```

Expand the expression $(1+x)^{10}$. Factor the expression $1+x^{105}$.

This is simply a matter of using proper *Mathematica* syntax.

```
Expand[(1 + x)^10]
```

```
Factor[1 + x^105]
```

Use `ComplexExpand` and its option `TargetFunctions` to convert `Abs[x + I y]` to `Sqrt[x^2 + y^2]`.

By default *Mathematica* does not simplify `Abs[x + I y]` because it assumes `x` and `y` could be complex-valued variables.

```
Abs[x + I y]
```

The option `TargetFunctions` allows us to specify the form in which we want the result of `ComplexExpand`.

```
? TargetFunctions
```

Here is the default setting of `TargetFunctions`.

```
Options[ComplexExpand]
```

To simplify `Abs[x + I y]`, we need to give a value for `TargetFunctions` that does not include `Abs`. Here we use the option value `{Re, Im}`, and we get the desired result.

```
ComplexExpand[Abs[x + I y], TargetFunctions -> {Re, Im}]
```

Substitution with Replacement Rules

Suppose we want to substitute a particular value for the variable `y` into the formula `Sqrt[x^2 + y^2]`. One way to substitute the value 7 for `y` is to set `y` equal to 7.

```
y = 7;
```

Evaluating `Sqrt[x^2 + y^2]` now reflects the new value of `y`.

```
Sqrt[x^2 + y^2]
```

The trouble with this approach is that the value 7 (in this case) will be substituted wherever `y` appears in any expressions evaluated after the assignment, at least until `y` is cleared of its value. For instance, if we later try to solve an equation with respect to `y`, *Mathematica* substitutes the value 7 into `y`, in effect assuming we want to solve the equation with respect to 7.

$$\texttt{Solve}\left[y^{10} + x^{10} == 1,\ y\right]$$

To clear the value of `y`, we use `Clear`.

```
Clear[y]
```

A better method for replacing any part of an expression with another value or expression uses the replacement operator pair `/.` and `->`. Expressions of the form *replacethis* `->` *withthis* are called *rules*, and *Mathematica* carries out any substitutions described by a rule or set of rules placed after the slash-period `/.`. The general syntax for making a substitution is *expr* `/.` *replacethis* `->` *withthis*.

For instance, to replace `y` in the expression `Sqrt[x^2 + y^2]` with 7, we type the following.

```
Sqrt[x^2 + y^2] /. y -> 7
```

Although we replaced the variable `y` with the value 7 in the expression `Sqrt[x^2 + y^2]`, the value for `y` remains undefined.

```
y
```

We are not limited to numerical replacements. Here we replace `y` with `1 + a`.

```
Sqrt[x^2 + y^2] /. y -> 1 + a
```

Here is an expression that contains the variable `x`.

```
Log[(1 - x) x]
```

We can replace `x` with a numerical value by using a replacement rule. (We enter the arrow character `→` by typing ESC `->` ESC.)

```
Log[(1 - x) x] /. x → 0.35
```

We can also replace any number of variables or subexpressions in an expression with others by giving a list of replacement rules.

```
a + b^2 + c^3 /. {a → α, b → β, c → γ}
```

We can also create a list of expressions formed by different lists of replacement rules by telling *Mathematica* to substitute more than one set of values for an expression. Below we compute three values for `Sqrt[x^2 + y^2]` by substituting into it a list containing three lists of replacement rules.

```
Sqrt[x^2 + y^2] /.
    {{x → 3, y → 4}, {x → 5, y → 12}, {x → 1, y → 1}}
```

A somewhat tricky case occurs when we perform more than one substitution in an expression with only one variable. For instance, to substitute two values for `x` into the expression defined above, we might try the following.

```
Log[(1 - x) x] /. {x -> 0.35, x -> 0.55} (* this is incorrect *)
```

Mathematica returned only one of the two desired values. The reason is that *Mathematica* performs the substitutions from left to right: First the rule `x -> 0.35` was applied to the formula, replacing all occurrences of `x` with 0.35; then the rule `x -> 0.55` was applied to *that* result, which was free of any occurrences of `x`.

To avoid this difficulty, we must put each set of replacement rules inside its own list, even if each set contains only one rule: this indicates to *Mathematica* that each replacement rule should be considered a separate solution set.

```
Log[(1 - x) x] /. {{x -> 0.35}, {x -> 0.55}} (* this is correct *)
```

In general *Mathematica* returns as many values from a substitution as there are sets of replacement rules.

One reason that replacement rules are important is that functions such as `Solve` return results in the form of a list of lists of replacement rules.

```
Solve[x^2 - 1 == 0, x]
```

Here we verify the solutions separately by substituting each one into the original equation, using the replacement operator `/.`.

```
x^2 - 1 == 0 /. x → -1
```

```
x^2 - 1 == 0 /. x → 1
```

We can test both solutions at the same time by substituting a list of solution sets into the equation.

```
x^2 - 1 == 0 /. {{x → -1}, {x → 1}}
```

An easier way to do the same thing is to name the list of replacement rules. Here we solve the same equation, this time naming the solution set of replacement rules `solset`.

```
solset = Solve[x^2 - 1 == 0, x]
```

Now, instead of retyping the solutions, we can directly substitute `solset` into the equation (note that the entire rule is contained in `solset`, so it is important not to type `x → solset` after the `/.`).

```
x^2 - 1 == 0 /. solset
```

Similarly, we can generate a list of the solutions by substituting the rules containing the solutions into the variable(s) of the equation.

```
x /. solset
```

See also `ReplaceAll`, `ReplaceRepeated`, `RuleDelayed`, `Dispatch`

■ Exercises: Substitution with Replacement Rules

Using a replacement rule, replace x in the expression $x^2 + 2x - 1$ with 5.

To replace x with 5, we use the replacement operator pair `/.` and `->` on the polynomial.

```
x^2 + 2 x - 1 /. x → 5
```

Verify that $x = 3$ and $x = 5$ are roots of the expression $x^3 - 9x^2 + 23x - 15$, and that $x = 4$ is not.

Enter the polynomial, calling it `cubic`.

```
cubic = x^3 - 9 x^2 + 23 x - 15
```

Replace x with 3.

```
cubic /. x → 3
```

The result is zero, so $x = 3$ is a root.

Next replace x with 4.

```
cubic /. x → 4
```

The result is not zero, so $x = 4$ is not a root.

Now replace *x* with 5.

```
cubic /. x → 5
```

Again we have a root.

We can test all three roots at once by giving a list of lists of replacement rules.

```
cubic /. {{x → 3}, {x → 4}, {x → 5}}
```

We generate `True` or `False` values by substituting the solutions into the equation `cubic==0`.

```
cubic == 0 /. {{x → 3}, {x → 4}, {x → 5}}
```

Verify that $-b-\sqrt{b^2-c}$ is a root of $x^2+2bx+c$.

The basic idea is the same as in the preceding problem. (We must be certain to type a space or asterisk between the `b` and `x` in the term `2 b x` to denote multiplication.)

```
x^2 + 2 b x + c /. x → -b - Sqrt[b^2 - c]
```

In this case, a complication is that the symbolic result does not automatically simplify. Therefore, we explicitly tell *Mathematica* to simplify the result.

```
Simplify[%]
```

Thus we see that the given expression is a root of $x^2+2bx+c$.

Solving Equations

■ Basic Solving

Using `Solve` gives generic solutions to an equation or system of equations. Recall that we use a double equal sign `==` to separate the left and right sides of an equation, and that we should specify a variable or list of variables to solve for. Here we solve the general quadratic equation with respect to *x*.

```
Solve[a x^2 + b x + c == 0, x]
```

Inside `Solve`, we enter a system of equations as a list of equations, followed by a list of variables to solve for.

```
Solve[{x + 5 y == c, 2 x + y == d}, {x, y}]
```

We can also solve some systems of equations expressed as matrices. For instance, the same set of linear equations as above can be solved by expressing the coefficients in the system as a matrix.

```
coeffs = {{1, 5}, {2, 1}};
```

We can then use **LinearSolve** to solve the same system.

```
?LinearSolve
```

```
LinearSolve[coeffs, {c, d}]
```

As expected, the answers are the same.

Naturally, there are many equations that cannot be solved using symbolic techniques, and when such equations are encountered we must use numeric solving or root-finding techniques.

Solve solves equations for general values of the parameters, so the solutions returned may be incorrect for special values of the parameters. For example, the solutions to the general quadratic equation are incorrect for the value $a = 0$.

```
Solve[a x^2 + b x + c == 0, x]
```

Reduce returns a list of logical statements that account for special values of parameters. (The form **&&** stands for the logical function **And**, and **||** for the logical function **Or**.)

```
Reduce[a x^2 + b x + c == 0, x]
```

The result states that when $a \neq 0$, the solutions $x = \frac{-b \pm \sqrt{b^2 - 4 a c}}{2 a}$ are correct; that when a, b, and c are all zero, any value of x is a solution; and that when $a = 0$ and $b \neq 0$, $x = -\frac{c}{b}$ is a solution.

See also **Eliminate**, **NSolve**, **FindRoot**, **ToRules**, **Root**, **ToRadicals**, **RootReduce**, **LogicalExpand**, **Algebra`InequalitySolve`**, **Calculus`RSolve**

■ Using Solutions as Replacement Rules

The following command loads from the **Graphics** package directory the package containing the **ImplicitPlot** command.

```
Needs["Graphics`ImplicitPlot`"]
```

Once the package is loaded, we can make a plot of two relations on the same set of axes.

```
imp = ContourPlot[{11 x^2 + 23 y^2 == 200, 10 y - 6 x^2 == -50}, {x, -5, 5},
    {y, -5, 5}];
```

Given symbols and integers as coefficients, *Mathematica* returns a list of exact solutions to the equations.

```
solns =
    Solve[
        {11 x^2 + 23 y^2 == 200, 10 y - 6 x^2 == -50},
        {x, y}]
```

As before, the answers are in the form of lists of replacement rules. As always, we can get numeric approximations to the solutions.

```
solns = N[solns]
```

We can then substitute the *x*-*y* values into the `Point` graphics primitive.

```
Point[{x, y}] /. solns
```

By default, points are the same color and thickness as lines, so we need to prepend the graphics directive `PointSize` to make the points visible.

```
bigpoints = Prepend[%, PointSize[0.04]]
```

When we plot the two relations, we can then highlight the solutions by including the `bigpoints` object in the plotting command.

```
Show[imp, Epilog → bigpoints]
```

■ Exercises: Solving Equations

Solve the equation $x^3 = 8$ with respect to x, then substitute the solutions back into the equation.

`Solve` gives an answer in the form of a list of replacement rules, and we call the list of rules `solns`. (It is important to use a double equal sign inside the `Solve` command to tell *Mathematica* that we are testing for the equality of the polynomial and 0.) The single equal sign after `solns` denotes that we wish to set `solns` equal to the result returned by `Solve`.

```
solns = Solve[x^3 == 8, x]
```

To verify the solutions, we substitute the solutions (`solns`) back into the equations using the replacement notation slash-period (`/.`).

```
x^3 == 8 /. solns
```

Solve the general quadratic equation, substituting the solutions into the equation to verify that they are roots.

Solve the equation, getting the two roots as a list of replacement rules.

```
Solve[a x^2 + b x + c == 0, x]
```

Next, substitute the roots into the equation.

```
a x^2 + b x + c == 0 /. %
```

The resulting equations need to be simplified.

```
Simplify[%]
```

Calculus and Analysis

■ Differentiation, Integration, and Taylor Series Expansion

For differentiation, integration, and series expansions we must indicate the variable(s) with respect to which the operations are being performed. Below we define an expression called `expr` and set it equal to $\sin(n \pi x)$.

```
Clear[expr] (* clear any previous values of expr *)
```

```
expr = Sin[n Pi x]
```

We take the derivative of `expr` with respect to `x`, using the differentiation operator `D`.

```
D[expr, x]
```

We can also use the special form $\partial_{\square}$ ■.

```
∂x expr
```

Similarly we integrate `expr` with respect to `x` using `Integrate`.

```
Integrate[expr, x]
```

We can also use the following form. We enter the integral sign ∫ by typing ⎋int⎋ or `\[Integral]`, and we must use the special differential ⅆ, entered by typing ⎋dd⎋ or `\[DifferentialD]`, and not an ordinary keyboard *d*.

```
∫expr ⅆx
```

Here we take a Taylor series expansion of **expr** with respect to **x**, around $x = 0$, up to degree 7.

```
Series[expr, {x, 0, 7}]
```

The $O[x]^8$ term denotes the extra terms beginning at order 8. To remove the order term, we use the function **Normal**. The result is a polynomial.

```
Normal[%]
```

We can apply these calculus operations to general functions and expressions as well.

```
D[f[x] g[x] h[x], x]
```

If *Mathematica* cannot return an antiderivative for an expression, it returns the expression unevaluated.

```
Integrate[f[x], x]
```

```
Integrate[Sin[Sin[x]], x]
```

Given limits of integration, we can use numerical methods to obtain an approximation.

```
NIntegrate[Sin[Sin[x]], {x, 0, π}]
```

All of the calculus functions apply to multivariate expressions. Here is a mixed partial derivative.

```
D[π^(x y), x, y]
```

In these examples, **Exp** is *Mathematica*'s name for the exponential function, and **Erf** is the name for the error function.

```
Integrate[Exp[-(α^2 + β^2)], α, β]
```

```
∫∫∫Exp[-(x² + y² + z²)] ⅆz ⅆy ⅆx
```

Here we see that *Mathematica* knows how to apply the fundamental theorem of calculus.

```
D[∫_a^b[z] f[x] ⅆx, z]
```

■ Other Calculus and Analysis Functions

`Sum` allows us to evaluate many finite and infinite sums and products.

We can evaluate summations with symbolic or infinite limits.

```
Sum[k^5, {k, 1, n}]
```

Mathematica recognizes many special summations, as well.

$$\sum_{k=0}^{\infty} \frac{x^k}{k!}$$

We can do a spot check of the answer by comparing a partial summation to the series expansion of *Mathematica*'s result.

```
Sum[x^k / k!, {k, 0, 7}]
```

```
Normal[Series[E^x, {x, 0, 7}]]
```

Mathematica solves a large class of ordinary differential equations (or systems of ODEs) symbolically, given the equations and initial conditions, a function or list of functions to solve for, and independent variables.

```
DSolve[y'[x] + 2 y[x] == 3 Exp[x], y[x], x]
```

```
DSolve[y'[x]/x - 2 y[x]/x^2 == x Cos[x], y[x], x]
```

The `C[1]` in each of the previous examples is an undetermined coefficient. To replace it with a numerical (or other) value, we use replacement rules.

```
% /. C[1] → 5
```

We can specify initial conditions, expressing them in the form of an equation (that is, using `==` notation).

```
DSolve[{y'[x] / x - (2 y[x] x) == x, y[π] == 3}, y[x], x]
```

Here *Mathematica* recognizes a special differential equation, and returns the answer as a linear combination of Bessel functions.

```
DSolve[z^2 y''[z] + z y'[z] + (z^2 - 169) y[z] == 0, y[z], z]
```

Mathematica can solve some partial differential equations. Here the solution includes an undetermined *function* `C[1]` of the quantity `2 t + u`.

```
DSolve[x[t, u] == D[x[t, u], t] - 2 D[x[t, u], u], x[t, u], {t, u}]
```

Mathematica calculates limits.

```
Limit[(x^2 - 4)/(x - 2), x -> 2]
```

Graphics are useful as an informal way to verify a limit. Here we find the limit of `Sin[x] / x` as `x` approaches zero.

```
Limit[Sin[x] / x, x -> 0]
```

A plot of $\frac{\sin(x)}{x}$ suggests that the result returned by `Limit` is correct.

```
Plot[Sin[x]/x, {x, -10, 10}]
```

We can specify the direction from which a limit is taken, setting the option `Direction` to 1 to take the limit from the left, or setting `Direction` to −1 to take the limit from the right.

Here is a plot of $\frac{1}{x}$.

```
Plot[1/x, {x, -0.5`, 0.5`}]
```

Setting `Direction` to 1, we take the limit from the left.

```
Limit[1 / x, x -> 0, Direction -> 1]
```

Setting `Direction` to −1, we take the limit from the right.

```
Limit[1 / x, x -> 0, Direction -> -1]
```

We can take limits of purely symbolic expressions.

```
Limit[(1 + m / n)^n, n → Infinity]
```

The `Interval` object returned by `Limit[Sin[x], x -> Infinity]` reflects the fact that the sine function oscillates forever between −1 and 1.

```
Limit[Sin[x], x → Infinity]
```

There are hundreds of mathematical functions available for use with symbolic arguments. Capabilities exist in the *Mathematica* kernel and standard packages to solve recurrence relations, calculate Laplace transforms, compute orthogonal polynomials, and much more. The Help Browser, on-line help, and *Mathematica* books provide convenient ways to explore these functions.

See also `Residue`, `ComposeSeries`, `InverseSeries`, `` Calculus`LaplaceTransform` ``, `` Calculus`FourierTransform` ``

■ Exercises: Calculus and Analysis

Find where the polynomial $x^3 - 6x^2 + 11x - 6$ crosses the x-axis, and where its derivative is equal to zero.

First we plot the curve.

```
Plot[x^3 - 6 x^2 + 11 x - 6, {x, 0, 4}]
```

To find where the curve crosses the *x*-axis, we use `Solve`.

```
Solve[x^3 - 6 x^2 + 11 x - 6 == 0, x]
```

For the second part of the question, we take the derivative of the polynomial, using the derivative operator `D`.

```
der = D[(x^3 - 6 x^2 + 11 x - 6), x]
```

Next, we use `Solve` again.

```
Solve[der == 0, x]
```

Using the command `InterpolatingPolynomial`, create a polynomial whose graph passes through the points (0, 1), (1, 11), (2, 21), and (3, 17); then take its derivative. If possible, plot the polynomial and its derivative.

We use `InterpolatingPolynomial` with the given points to create the polynomial.

```
mypoly = InterpolatingPolynomial[{{0, 1}, {1, 11}, {2, 21}, {3, 17}}, x]
```

Here it is in simpler form.

```
mypoly = Simplify[mypoly]
```

Here is the derivative of our interpolating polynomial.

```
der = ∂x mypoly
```

Here is a somewhat simpler form of the derivative.

```
der = Simplify[der]
```

Here is a plot of `mypoly` and its derivative.

```
Plot[{mypoly, der}, {x, -0.1`, 3.1`},
 PlotStyle → {GrayLevel[0.5`], GrayLevel[0.2`]}]
```

Solve the differential equation $y''(x) = y'(x) + e^x$, when $y(0) = 2$ and $y'(0) = 1$. Additionally, try plotting $y(x)$ over $-1 \leq x \leq 1$.

First we solve the differential equation using `DSolve`. Note that the initial conditions are written as equations, not assignments (`y[0]==2`, not `y[0]=2`). For convenience, we call the solution `dsol`.

```
dsol = DSolve[{y''[x] == y'[x] + E^x, y[0] == 2, y'[0] == 1}, y[x], x]
```

We can isolate $y(x)$ using the replacement operator `/.`.

```
y[x] /. dsol
```

Next, plot $y(x)$ using `Plot`.

```
Plot[%, {x, -1, 1}]
```

Load the package `DiscreteMath`RSolve``. Using the on-line help, solve the following system of recurrence equations, and compute a_0 through a_{10}.

$$a_n = a_{n-1} + a_{n-2},\ a_0 = 1,\ a_1 = 3$$

Here we load the package and examine the usage message for `RSolve`.

```
Needs["DiscreteMath`RSolve`"]
```

```
?RSolve
```

We see from the usage message that `RSolve` accepts a list of recurrence equations, a list of functions to solve for, and an independent variable. Here we enter the given system of equations, following the directions of the usage message, and get the result. (It is possible that we have previously defined values or rules for `a` and `n`; to be safe, we clear the variables first.)

```
Clear[a, n];
```

```
RSolve[{a[n] == a[n - 1] + a[n - 2], a[0] == 1, a[1] == 3}, a[n], n]
```

Extract the general result, calling it `gen`, by replacing `a[n]` with the solution.

```
gen = (a[n] /. First[%])
```

Here we test some values of `n`, replacing the `n` in `gen` with 0.

```
gen /. n → 0
```

The answer can be simplified.

```
Simplify[%]
```

We test the case $n = 1$.

```
Simplify[gen /. n → 1]
```

Now we generate the table of values.

```
Table[Simplify[gen /. n → j], {j, 0, 10}]
```

These numbers are called the Lucas numbers.

Lists and Functions

Function Definitions

Function definitions that we have seen look like the following.

```
func[z_] := 1 + z^10
```

The function has a name (`func`) and a pattern to match the argument(s) given to the function (`z_`, read as "any *z*"; the name of the pattern is unimportant, except that we must use the same name on the right side of the definition) on the left side of the colon-equal, and something to do with the argumen-t(s) on the right side of the colon-equal.

We define functions that take more than one argument in the same way, except we type in a pattern for each argument that the function is to accept. The following function `dist` takes two arguments, any x_1 and any x_2, and returns the absolute value of the difference of the two arguments.

```
dist[x1_, x2_] := Abs[x1 - x2]
```

The *body* of a function (the part on the right-hand side of the definition) can be as complicated as necessary, and can contain compound expressions. The following function assigns values to the variables `X` and `Y`, then prints the new values.

```
setXandY[xval_, yval_] :=
    (X = xval; Y = yval; Print["X is now ", X, ", and Y is now ", Y])
```

Mathematica executes all of the commands in the body of the function.

```
setXandY[3, -7]
```

```
X / Y
```

When we finish using `X` and `Y` we clear their values.

```
Clear[X, Y]
```

We can define functions so that only certain types of arguments are valid. For instance, here is a recursive definition of a factorial function.

```
fac[n_] := n fac[n - 1]; fac[0] := 1
```

Here we compare a value computed with `fac` to a value computed using the built-in factorial function.

```
{fac[35], 35 !}
```

A shortcoming of `fac` is that it should apply only to integers. We can determine a number's type by using the `Head` function. Possible values for the head of a number are `Integer`, `Rational`, `Real`, and `Complex`.

```
Head[3]
```

Head$\left[\frac{22}{7}\right]$

```
Head[3.14159]
```

We tell *Mathematica* that a function applies only to a certain type of number by typing the permissible head of the number after the underscore in the function definition. (Before redefining `fac`, we use `Clear` to erase the old definition.)

```
Clear[fac]
```

Here we restrict `fac` to arguments that have the head `Integer`.

```
fac[n_Integer] := n fac[n - 1]; fac[0] := 1
```

The function works as intended when given an integer argument.

```
fac[35]
```

Mathematica returns **fac** unevaluated if called with anything but an integer.

```
fac[1.23456]
```

A further shortcoming of **fac** is that it should accept only *positive* integers. We can include conditions (such as "*n* must be positive") by typing slash-semicolon **/;** after the body of the function, followed by the condition. First we clear **fac**.

```
Clear[fac]
```

Now we include the condition by typing **/;** after the body of the definition, followed by the condition $n > 0$.

```
fac[n_Integer] := n fac[n - 1] /; n > 0
```

```
fac[0] := 1
```

The function works for arguments that are positive integers.

```
fac[40]
```

It returns unevaluated if called with anything other than a positive integer.

```
fac[-10]
```

See also **If**, **Which**, **Switch**, **Do**, **For**, **Alternatives**, **Optional**, **PatternTest**, **MatchQ**

■ List Functions

Mathematica has many functions for creating lists. One such function is **Range**, which generates a list of numbers. **Range[***n***]** creates a list of numbers going from 1 to *n*.

```
Range[15]
```

Range[*m***,** *n***]** returns a list of numbers from *m* to *n*.

```
Range[20, 30]
```

Range[*m***,** *n***,** *s***]** returns a list of numbers from *m* to *n* in increments of *s*.

```
Range[3, 4, 1 / 10]
```

Another function for creating lists is **Table**, which we have seen before. Here is a table of approximations to the natural logarithm of the i^{th} prime number, as i goes from 1 to 5.

```
Table[N[Log[Prime[i]]], {i, 1, 5}]
```

The elements of a list can be other lists, and lists can be nested to any depth. For instance, a matrix is a list of lists, where each sublist contains the elements of one row of the matrix. Here is a 3×3 matrix.

```
mymat = {{i, j, k}, {1, 2, 3}, {-1, 0, -1}};
```

We format matrices and arbitrary arrays of elements using **MatrixForm** and **TableForm**.

```
MatrixForm[mymat]
```

See also **Part**, **Extract**, **Take**, **Drop**, **Append**, **AppendTo**, **Prepend**, **PrependTo**, **Insert**, **Delete**, **Join**, **Intersection**, **Union**

■ Using Functions with Lists

Mathematica provides many functions designed to allow functions and lists to work together.

One such function is **Map**, which applies a function to each element of a list. Here is a simple function.

```
nlp[x_Integer] := N[Log[Prime[x]]]
```

To apply the function to each element of a list of integers, we use **Map**. Notice that we give only the name of the function (**nlp**) as the first argument to **Map**, and not **nlp[x]**.

```
Map[nlp, Range[5]]
```

There is a special class of functions called *predicate functions*, each of which returns **True** or **False** depending on whether its condition is met. All the built-in predicate functions in *Mathematica* end with the letter Q. Here are the names of all the built-in functions that end with the letter Q (not all of them are predicate functions).

```
Names["*Q"]
```

Here are some predicate functions applied to the number 101. Here we check if 101 is prime.

```
PrimeQ[101]
```

Here we determine if 101 is even.

```
EvenQ[101]
```

Mathematica has a function called `Select` that extracts all the elements of a list that satisfy a particular predicate function. For example, we extract all the prime elements of a list of integers from 1 to 100. (Notice again that we use only the name of the predicate function.)

```
Select[Range[100], PrimeQ]
```

We can define our own predicate functions. The following predicate function returns `True` for numbers between 0.33 and 0.66.

```
middlethird[x_] := 0.33 < x < 0.66
```

Given a list of numbers, we can select all the numbers that satisfy `middlethird`. Here is a list of 25 random real numbers between 0 and 1.

```
randompoints = RandomReal[{0, 1}, 25]
```

Here is the subset of `randompoints` whose elements satisfy `middlethird`.

```
Select[randompoints, middlethird]
```

See also `Cases`, `MemberQ`, `FreeQ`, `Count`, `Position`, `DeleteCases`

Graphics

Two-Dimensional Graphics

■ Plot

The simplest example of *Mathematica*'s graphing capabilities is a graph of a function of one variable created with `Plot`. `Plot` takes a function to be graphed and a domain for the variable, and generates a two-dimensional graph.

```
Plot[Sin[x] / x, {x, -10, 10}]
```

`Plot` also accepts a list of functions to plot on the same set of axes.

```
Plot[{Sin[x], Cos[x]}, {x, -π, π}]
```

■ Options

There are dozens of options we can use to control almost every aspect of a graph.

Options[*FunctionName***]** returns a list of the options available for a function, along with their default values.

```
Options[Plot]
```

For example, by default, *Mathematica* uses an algorithm to choose the most "interesting" *y*-range for a graph. In the above list we see that the default value for **PlotRange** is **Automatic**.

```
Plot[Sin[x^2]/x^2, {x, -10, 10}]
```

We can override the default setting by giving a different value to the **PlotRange** option.

```
Plot[Sin[x^2]/x^2, {x, -10, 10}, PlotRange → {-0.25`, 1.05`}]
```

In every case, options are added after the required arguments to the function. We set an option by typing the name of the option, an arrow made by the two characters **-** and **>** (or the special character **→** made by typing ESC**->**ESC), and the new value of the option. Note that most plotting commands accept the same set of options.

```
Plot[Sin[x]/x, {x, -10, 10}, Frame → True, PlotLabel → "sinc function",
 GridLines → Automatic, PlotRange → {{-11, 11}, {-0.5`, 1.15`}},
 AspectRatio → 1]
```

In the following sections we will change many of a graph's default option settings.

See also **SetOptions**, **FullOptions**, **FullGraphics**

■ ParametricPlot

ParametricPlot plots a two-dimensional curve described by two functions of the same parameter, one that describes movement in the *x* direction and one for the *y* direction. This allows us to plot curves that are not functions, in the mathematical sense. Here is a parametric plot of a circle.

```
ParametricPlot[{Sin[t], Cos[t]}, {t, 0, 2π}]
```

The option **AspectRatio** controls the relative sizes of units on the two axes; the setting **Automatic** makes them equal (that is, makes one unit on the vertical axis equal to one unit on the horizontal axis).

```
ParametricPlot[{Sin[t], Cos[t]}, {t, 0, 2 π}, AspectRatio → Automatic]
```

ParametricPlot can be used to plot graphs of complicated curves that cannot be expressed as a function of the form $y = f(x)$.

```
ParametricPlot[{4 Cos[-5 t/4] + 7 Cos[t], 4 Sin[-5 t/4] + 7 Sin[t]},
  {t, 0, 8 π}, AspectRatio → Automatic]
```

ParametricPlot takes many of the same options as **Plot**.

```
ParametricPlot[{4 Cos[-11 t/4] + 7 Cos[t], 4 Sin[-11 t/4] + 7 Sin[t]},
  {t, 0, 8 π}, AspectRatio → Automatic, Axes → False, Frame → True,
  FrameLabel → {"x", "y"}]
```

ImplicitPlot

ImplicitPlot allows us to plot implicit relations, rather than functions. It is defined in one of the standard packages, so we must load it first with the **Needs** command.

```
Needs["Graphics`ImplicitPlot`"]
```

ImplicitPlot[*eqn***, {***x***,** *a***,** *b***}]** draws a graph of the set of points that satisfy *eqn*. The variable *x* is associated with the horizontal axis and ranges from *a* to *b*. The remaining variable in the equation is associated with the vertical axis. We can also specify a vertical range for the graph using the form **ImplicitPlot[***eqn***, {***x***,** *a***,** *b***}, {***y***,** *c***,** *d***}]**.

```
ImplicitPlot[3 x^2 + 3 x y + 12 y^2 == 12, {x, -2.5, 2.5}, AxesOrigin → {0, 0}];
```

Like most graphing functions, **ImplicitPlot** accepts a list of functions to plot on the same set of axes.

```
ContourPlot[{3 x^2 + 3 x y + 12 y^2 == 12, 12 x^2 + 3 x y + 3 y^2 == 12,
    3 x^2 + 12 x y + 3 y^2 == 1}, {x, -2.5`, 2.5`}, {y, -2.5`, 2.5`},
   AxesOrigin → {0, 0}];
```

Graphics Directives and Plot Styles

Mathematica contains several objects called *graphics directives*, which specify the style in which a graph should be drawn. Graphics directives control the color, thickness, point size, and dashing of a lines, points, and other objects.

For example, to specify that lines should be drawn with a specified thickness, we use the directive **Thickness[*t*]**, where *t* is given as a percentage of the width of a graph.

All two-dimensional graphing functions have an option called **PlotStyle**, which allows us to specify a list of graphics directives that control how the actual curve (as opposed to the surrounding axes, grid lines, etc.) is drawn. To draw a sine wave so that the curve is drawn with a thickness 2% of the width of the graph, we set the option **PlotStyle -> Thickness[0.02]**.

```
Plot[Sin[x], {x, -3, 3}, PlotStyle → {Thickness[0.02`]}]
```

To draw a curve with a dashed line, we use the directive **Dashing**. **Dashing[{*d*}]** draws a line so that it alternates between line segments *d* percent of the width of the graph and gaps *d* percent of the width of the graph. **Dashing[{**d_1**,**d_2**}]** alternates between line segments d_1 long and gaps d_2 long, and **Dashing[{**d_1**,**d_2**,**…**}]** applies the successive widths cyclically. The following graph uses line segments twice as long as the gaps.

```
Plot[Sin[x], {x, -3, 3}, PlotStyle → {Dashing[{0.04`, 0.02`}]}]
```

There are several ways to specify colors using graphics directives. **RGBColor[*r*,*g*,*b*]** describes a color made up of *r*, *g*, and *b* percent of red, green, and blue. Thus **RGBColor[1,0,0]** is red, and **RGBColor[1,0,1]** is purple. (The parameters *r*, *g*, and *b* must all be between 0 and 1.)

The add-on package **Graphics`Colors`** defines a list of English names for colors and their **RGBColor** values. Here we load the package.

```
Needs["Graphics`Colors`"]
```

Here are the first ten colors defined in the package.

```
Take[AllColors, 10]
```

Here is the **RGBColor** value of the color apricot.

```
Apricot
```

Other color functions are **Hue[*h*]**, which represents the spectrum of colors going through red, orange, yellow, green, blue, purple, and back to red; and **GrayLevel[*g*]** ($0 \le g \le 1$), where **GrayLevel[0]** is black and **GrayLevel[1]** is white.

To change the style of each graph in a list given to **Plot**, we give a list containing as many graphics directives as there are functions being plotted. Here we draw the first curve in the list using a thick line (**Thickness[0.02]**), and the second curve using a green line (**RGBColor[0, 1, 0]**).

```
Plot[{Sin[x], Cos[x]}, {x, -3, 3},
 PlotStyle → {Thickness[0.02`], RGBColor[0, 1, 0]}]
```

To apply multiple styles to each function in a list, we surround the styles that apply to each function inside a set of list brackets.

```
Plot[{Sin[x], Cos[x]}, {x, -3, 3},
 PlotStyle → {{Thickness[0.02`], Apricot},
   {Dashing[{0.04`, 0.02`}], Green}}]
```

One tricky case to be aware of is that to specify more than one graphics directive in the plot style of a single function, we must surround the graphics directives with double list brackets `{{` and `}}`.

```
Plot[Sin[x]/x, {x, -10, 10},
 PlotStyle → {{RGBColor[0, 0, 1], Thickness[0.015`]}}]
```

See also `PointSize`, `CMYKColor`, `AbsoluteThickness`, `AbsolutePointSize`, `Graphics`ArgColors``

■ Combining Graphs

`Show` allows us to display a previously computed graph without having to recompute any of the points that make up the curve(s).

`Show[`*graphics*`,` *options*`]` displays two- and three-dimensional graphics using the new option settings specified. `Show` accepts options that affect the way a graph or its surrounding elements (axes, frame, etc.) is drawn, without requiring any points of the graph to be recomputed.

```
p1 = Plot[Sin[x]/x, {x, -10, 10}]
```

None of the options given below require points on the graph to be recomputed, so we can use them inside `Show`.

```
Show[p1, Frame → True, AxesStyle → Hue[0], PlotRange → {-0.1`, 0.8`},
 PlotLabel → "p1"]
```

When given a list of graphics, `Show` combines them onto the same set of axes. Here is another two-dimensional plot.

```
p2 = Plot[BesselJ[2, x], {x, -10, 10}, PlotStyle → {GrayLevel[0.5`]}]
```

We combine `p1` and `p2` by placing their names in a list and giving the list to `Show`.

```
Show[{p1, p2}]
```

`Show` also combines three-dimensional graphics. Here are two three-dimensional graphs.

We put them in the same box using **Show**.

```
Show[{p3, p4}]
```

The object **GraphicsArray** takes an array of graphs, and when used with **Show** displays the graphs in an array. Here is an array containing the four previous graphs.

```
Show[GraphicsGrid[{{p1, p3}, {p4, p2}}], ImageSize → {400, 250}]
```

The package **Graphics`Graphics`** defines functions **DisplayTogether** and **DisplayTogetherArray**, which allow us to combine graphs on the same set of axes or in an array without rendering each graph beforehand. To use the functions we first load the package.

```
Needs["BarCharts`"]; Needs["Histograms`"];
Needs["PieCharts`"]
```

```
Show[Plot[Sin[x], {x, -3, 3}, PlotStyle → GrayLevel[0.5`]],
 Plot[Sin[3 x], {x, -3, 3}, PlotStyle → Dashing[{0.01`}]]]
```

```
GraphicsRow[{Plot[Sin[x], {x, -3, 3}, PlotStyle → GrayLevel[0.5`]],
  Plot[Sin[3 x], {x, -3, 3}, PlotStyle → Dashing[{0.01`}]]}]
```

See also **DisplayFunction**, **$DisplayFunction**, **Identity**

■ How *Mathematica* Draws a Graph

Mathematica uses an adaptive sampling algorithm to choose the points sampled in a two-dimensional plot. Beginning with 25 equally spaced points dividing the domain to be plotted, *Mathematica* looks at each set of three consecutive points and computes the angle between the line segment joining the first and second points and the line segment joining the second and third points. If this angle is close to 180 degrees, then *Mathematica* connects the points with lines. If not, *Mathematica* subdivides that interval and tries again. This allows *Mathematica* to sample more points in a “curvy” section of the function than in a flat section.

This process can be controlled by the options **PlotDivision**, which is the upper limit on the number of times an interval will be divided, and **PlotPoints**, which sets the initial number of points to be sampled.

Although this is a very robust algorithm, which produces accurate results in most cases, any scheme using a finite number of sampled points is prone to miss sometimes. Here is an example of a function that is plotted incorrectly using the default number of plot points, but which can be accurately plotted by raising the initial number of plot points used.

```
Plot[x + Sin[2 π x], {x, 0, 25}]
```

```
Plot[x + Sin[2 π x], {x, 0, 25}, PlotPoints → 50]
```

■ Exercises: Two-Dimensional Graphics

Load the package `Graphics`Master``. This loads the name of every function defined in a graphics package into memory, and tells *Mathematica* to load the appropriate package when a package function is first used.

```
Needs["Graphics`Master`"]
```

Make a simple two-dimensional plot of $x^2 - 20\cos(x^2)$ between -10 and 10. (If there are obvious flaws in the graph, plot it again using more plot points.)

```
Plot[x² - 20 Cos[x²], {x, -10, 10}]
```

Use `Options[Plot]` or the on-line help to find a list of all the options that **Plot** accepts. Plot the same function as above, this time changing at least five of *Mathematica*'s default options.

```
Plot[x² - 20 Cos[x²], {x, -10, 10}, PlotPoints → 75, Frame → True,
  GridLines → Automatic, PlotLabel → "exercise one", AspectRatio → 0.5`]
```

Three-Dimensional Graphics

■ Plot3D

Plot3D is the three-dimensional analog of the **Plot** command. Given a function of two variables and a domain for each variable, **Plot3D** produces a surface plot.

```
Plot3D[Sin[x - Cos[y]], {x, -3, 3}, {y, -3, 3}]
```

Applying options to three-dimensional graphics works the same as with two-dimensional graphics; in fact, many of the options are the same.

One of the differences between two- and three-dimensional plotting in *Mathematica* is point sampling. Instead of adaptive sampling, three-dimensional plots rely on a fixed grid of points at which to evaluate the function. By default, a 15×15 grid is used, resulting in $15^2 = 225$ points plotted; raising this number results in a smoother graph, but takes more time and memory to generate.

Here is a smoother graph of the same function as above.

```
Plot3D[Sin[x - Cos[y]], {x, -3, 3}, {y, -3, 3}, Axes → False,
  FaceGrids → All, PlotPoints → 25]
```

See also **HiddenSurface**, **RenderAll**, **Lighting**, **ColorFunction**

■ Changing the Viewpoint

One important option to three-dimensional plotting functions is the viewpoint, the point in space from which the observer looks at the object. `ViewPoint` is an option to all three-dimensional graphics functions. Its default value is `{1.3, -2.4, 2.0}`, which can be changed by entering a new value directly as an option.

```
Show[%, ViewPoint → {0, 3, 2}]
```

Mathematica provides an easier way to do this using the 3D ViewPoint Selector. To use this front-end feature we pull down the Input menu and choose 3D ViewPoint Selector. Rotating the box with the mouse will have *Mathematica* compute the point from which to view the object. The Paste button enters the view point at the current text insertion point.

■ ParametricPlot3D

`ParametricPlot3D` is the three-dimensional analog of `ParametricPlot`. Depending on the input, `ParametricPlot3D` produces a space curve or a surface.

When we give `ParametricPlot3D` a list of three parametric functions in one parameter, the result is a space curve.

```
ParametricPlot3D[{Sin[t], Cos[t], t/3}, {t, 0, 6π}, Axes → False]
```

A list of three parametric functions in two parameters results in a surface.

```
ParametricPlot3D[{Sin[v] Cos[u], Sin[v] Sin[u], Cos[v]},
  {u, 0, 1.5` π}, {v, 0, π}]
```

Like most graphing functions, `ParametricPlot3D` accepts a list of sets of parametric equations and plots the surfaces together.

```
ParametricPlot3D[{{Sin[v] Cos[u], Sin[v] Sin[u], Cos[v]},
   {1/2 Sin[v] Cos[4 u/3], 1/2 Sin[v] Sin[4 u/3], Cos[v]/2}}, {u, 0, 1.5` π},
  {v, 0, π}]
```

Options are given to `ParametricPlot3D` the same way as for `Plot3D`. Most of the options are the same.

■ Exercises: Three-Dimensional Graphics

Make a three-dimensional plot of the function $\sin(x + \sin(y))$ between -3 and **3** on both axes.

```
Plot3D[Sin[x + Sin[y]], {x, -3, 3}, {y, -3, 3}]
```

Evaluate **Options[Plot3D]** or use the on-line help to find a list of all the options that **Plot3D** accepts. Plot the same function again, this time changing at least four of *Mathematica*'s default options, including the options that control the smoothness of the plot and the color.

```
changedplot3d = Plot3D[Sin[x + Sin[y]], {x, -3, 3}, {y, -3, 3},
  PlotPoints → {15, 45}, Mesh → False, ColorFunction → Hue,
  FaceGrids → All]
```

Contour and Density Graphics

■ ContourPlot and DensityPlot

Mathematica plots contour and density plots of functions of two or three variables. With the exception of special options that apply only to these types of graphics, these functions work very much like **Plot** and **Plot3D**.

ContourPlot displays a graphics of a function of two variables, where regions of different intensities of gray have (nearly) the same function value.

```
ContourPlot[Exp[x] Sin[y], {x, -3, 3}, {y, -3, 3}]
```

DensityPlot by default generates a grid of gray levels, where the lighter gray areas have greater function values than the darker gray areas.

```
DensityPlot[Exp[x] Sin[y], {x, -3, 3}, {y, -3, 3}]
```

See also **ColorFunction**, **Mesh**, **Contours**, **ContourLines**, **ContourStyle**

■ ContourPlot3D

The function **ContourPlot3D** provides a way to plot surfaces showing particular values of a function of three variables. This function is defined in one of the standard add-on packages, so we must load the package before using the function.

```
Needs["Graphics`ContourPlot3D`"]
```

ContourPlot3D[*fun***, {***x***,** x_0**,** x_1**}, {***y***,** y_0**,** y_1**}, {***z***,** z_0**,** z_1**}]** plots the surface implicitly defined by `fun[`*x*`,` *y*`,` *z*`] == 0`. Setting the option `Contours` to `{`*val*$_1$`,` *val*$_2$`, … }` plots the level surfaces corresponding to the values *val*$_1$, *val*$_2$, …

```
ContourPlot3D[√(x² + y² + z²) , {x, -1, 1}, {y, 0, 1}, {z, 0, 1},
 Contours → {0.25`, 0.5`, 0.75`}]
```

Exercises: Contour and Density Graphics

Create a density plot of the function $\sin(x - \sin(y))$ over any range that includes the origin. Render the graphic with twice as many plot points. Experiment with other options.

```
DensityPlot[Sin[x - Sin[y]], {x, -10, 10}, {y, -10, 10}]
```

```
DensityPlot[Sin[x - Sin[y]], {x, -10, 10}, {y, -10, 10},
 PlotPoints → 30, Mesh → False, FrameLabel → {"x", "y"}]
```

Repeat the above exercise using `ContourPlot` instead of `DensityPlot`. Experiment with the options to `ContourPlot` that do not apply to `DensityPlot`.

```
ContourPlot[Sin[x - Sin[y]], {x, -10, 10}, {y, -10, 10}]
```

```
ContourPlot[Sin[x - Sin[y]], {x, -10, 10}, {y, -10, 10},
 PlotPoints → 30, Contours → 30, ContourStyle → None]
```

Plotting Data

There are many occasions when we want to work with data rather than functions. There are several functions designed to visualize data in two or three dimensions. For these examples, we need data to work with. In practice, we would most likely read this data from a file or use the output of other calculations. For this demonstration we will create a list of ordered pairs to use as data.

```
exampleData = N[Table[{n, n + Sin[n] + RandomReal[]}, {n, 0, 5 π, 0.2` π}]];
```

`ListPlot` takes a vector or array of data and plots it in two dimensions. Given a one-dimensional set of data such as `{10, 20, 30, 40}`, *Mathematica* plots the ordered pairs `{{1, 10}, {2, 20}, {3, 30}, {4, 40}}`. In this case, we supply a list of ordered pairs and *Mathematica* plots the points using our explicit *x* values. (The graphics directive `PointSize[`*p*`]` specifies that points should be drawn so they are *p* percent of the width of the graph.)

```
pointplot = ListPlot[exampleData, PlotStyle → PointSize[0.02`]]
```

Options to `ListPlot` include nearly all of those applicable to `Plot`. One exception is the option `PlotJoined`, which when set to `True` draws a line connecting each of the points.

```
joinedplot = ListPlot[exampleData, Joined → True,
  PlotStyle → RGBColor[0, 0, 1]]
```

At this point in our analysis we can easily find a good least-squares fit to this data. The function `Fit` takes as arguments a set of data, a set of basis functions for the best-fit polynomial, and a list of variables to be used. Below we include only constant, linear, and quadratic terms for the best-fit function.

```
exampleFit = Fit[exampleData, {1, x, x^2}, x]
```

Here is a plot of the best-fit quadratic polynomial.

```
fitplot = Plot[exampleFit, {x, 0, 5 π}, PlotStyle → Dashing[{0.01`}]]
```

Here we combine the previous three graphs.

```
Show[{pointplot, joinedplot, fitplot}]
```

When working with three-dimensional data, we use analogs to **`Plot3D`**, **`DensityPlot`**, and **`ContourPlot`**. **`ListPlot3D`** plots a three-dimensional surface from a rectangular array of height values.

```
examplearray = Table[n + Sin[n] + 3 RandomReal[], {i, 1, 5 π, 0.3` π},
   {n, 1, 5 π, 0.3` π}];
```

```
ListPlot3D[examplearray]
```

`ListContourPlot` and **`ListDensityPlot`** create density and contour plots from rectangular arrays of data.

```
ListDensityPlot[examplearray]
```

■ Exercises: Plotting Data

Create tabular data from the curve $x\cos(x) - \sin(x^2)$ between $x = -2\pi$ and $x = 2\pi$ and display the data using **`ListPlot`**.

```
mydata = Table[N[{x, x Cos[x] - Sin[x²]}], {x, -2 π, 2 π, 0.05 π}];
```

```
ListPlot[mydata, PlotStyle → PointSize[0.02`]]
```

Create a list of the first twenty prime numbers. (Hint: use **`Table`** and **`Prime[`***n***`]`**, which gives the n^{th} prime number.) Plot the list using **`ListPlot`**, then fit the data to a quadratic polynomial, and plot the data and the best-fitting curve on the same set of axes. Use options to change the color and other aspects of the graphic.

```
primes = Table[Prime[n], {n, 1, 20}]
```

```
pointplot = ListPlot[primes, PlotStyle → {Red, PointSize[0.02`]}]
```

```
fitline = Fit[primes, {1, x, x²}, x]
```

```
fitplot = Plot[fitline, {x, 0, 20}, PlotStyle → {Blue}]
```

```
Show[{pointplot, fitplot}]
```

Graphics Primitives

In addition to the high-level plotting functions just described, *Mathematica* allows us to build up graphics in two and three dimensions from the basic building blocks of points, lines, circles, and so on. These building blocks are called *graphics primitives*.

Graphics primitives are the actual objects that are drawn, while graphics directives control the style in which an object is drawn.

First we look at an example in two dimensions. The following syntax is used to render a series of graphics primitives.

```
Show[Graphics[{graphics primitives and directives}]]
```

The primitives **`Point`**, **`Line`**, **`Polygon`**, **`Text`**, **`Rectangle`**, **`Cuboid`**, **`Circle`**, and **`Disk`** form the basis for most graphics.

Circle[{ x**,** y**},** r**]** is a two-dimensional graphics primitive that represents a circle of radius r centered at the point $\{x, y\}$. **Circle[{** x**,** y**}, {** r_x**,** r_y **}]** yields an ellipse with semi-axes r_x and r_y. **Circle[{** x**,** y **},** r**, {** θ_1**,** θ_2 **}]** represents a circular arc. **Line[{** p_1**,** p_2**,** … **}]** is a graphics primitive which represents a line joining a sequence of points.

Here is a diagram made up of text, line, and circle primitives.

```
Show[Graphics[{Text["r", {1.6`, -0.2`}], Text["θ", {0.8`, 0.35`}],
   Thickness[0.015`], Circle[{0, 0}, 3], Thickness[0.01`],
   Line[{{3, 0}, {0, 0}, {3 Cos[π/4], 3 Sin[π/4]}}], Thickness[0.005`],
   Dashing[{0.0075`}], Circle[{0, 0}, 1.25`, {0, π/4}]}],
 AspectRatio → Automatic]
```

Here is a more complicated example.

```
Show[
 Graphics[
  {{GrayLevel[0.75`], Polygon[{{0, 0}, {1, 1}, {0, 2}, {-1, 1}, {0, 0}}]},
   {Hue[0], Thickness[0.01`],
    Line[{{0, 0}, {1, 1}, {0, 2}, {-1, 1}, {0, 0}}]},
   Line[{{0, 0}, {0, 2}}],
   {Dashing[{0.01`}], Circle[{0, 0}, 1], Circle[{0, 0}, √2],
    Line[{{-2, -2}, {2, 2}}], Line[{{-2, 2}, {2, -2}}],
    Line[{{-2, 0}, {2, 4}}], Line[{{-2, 4}, {2, 0}}]},
   {Thickness[0.01`], Line[{{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}}]}}],
 Axes → True, AxesOrigin → {0, 0}, AspectRatio → Automatic,
 PlotRange → {Automatic, {-0.2`, 2.2`}},
 PlotLabel → "duplicating the square"]
```

This works similarly in three dimensions using **Graphics3D** in place of **Graphics**.

```
Show[
 Graphics3D[{Polygon[{{0, 0, 0}, {0, 4, 0}, {4.5`, 4, 0}, {4.5`, 0, 0}}],
   PointSize[0.03`], Table[Point[{t, 2, Abs[50 + 20 t - 8 t^2]}],
    {t, 0, 4, 0.2`}]}], BoxRatios → {1, 0.25`, 1},
 ViewPoint → {-0.012`, -3.22`, 1.04`}, Axes → {True, False, True},
 AxesLabel → {"time", None, "height"}]
```

Several of the standard *Mathematica* packages define more graphics objects and tools to manipulate them.

■ Exercises: Graphics Primitives and Directives

Below is the *Mathematica* code to draw a face. Use other two-dimensional graphics commands (such as **`Polygon`**) to add other features (nose, beard, hat, etc.) and make changes to the face (such as eye color). Experiment and have fun.

```
Show[Graphics[{Thickness[0.03`], Circle[{0, 0}, 1], PointSize[0.04`],
   Point[{-0.5`, 0.3`}], Point[{0.5`, 0.3`}],
   Circle[{0, -0.1`}, 0.5`, {5π/4, 7π/4}]}], AspectRatio → Automatic]
```

Here is one variation.

```
Show[Graphics[{Thickness[0.03`], Circle[{0, 0}, 1], Thickness[0.009`],
   Blue, Circle[{-0.5`, 0.3`}, 0.04`], Circle[{0.5`, 0.3`}, 0.04`],
   Red, Circle[{0, -0.1`}, 0.5`, {5π/4, 7π/4}], Pink,
   Polygon[{{-0.25`, -0.9`}, {0.25`, -0.9`}, {0, -1.25`}}],
   Green, Polygon[{{-0.6`, 0.9`}, {0.6`, 0.9`}, {0.6`, 1.1`},
     {-0.6`, 1.1`}}],
   Polygon[{{-0.4`, 1.1`}, {0.4`, 1.1`}, {0.4`, 1.5`},
     {-0.4`, 1.5`}}]}], AspectRatio → Automatic]
```

Shapes and Polyhedra

The packages **`Graphics`Shapes``** and **`Graphics`Polyhedra``** provide *Mathematica* definitions of common three-dimensional shapes and the regular Platonic polyhedra, as well as functions for affine transformations on them.

```
Needs["Graphics`Shapes`"]
```

Once a package is loaded, we can get a list of all the objects that it defines by giving using **`?`** with the context name and **`*`** (to denote all names).

```
?Graphics`Shapes`*
```

`Torus[`r_1**`,`** r_2**`,`** n **`,`** m **`]`** is a list of m n polygons approximating a torus centered around the z-axis with radii r_1 and r_2.

We use **`Show`** and **`Graphics3D`** to have *Mathematica* render the shape.

```
Show[Graphics3D[Torus[1, 0.75`, 25, 25]]]
```

Transformations on these *Mathematica* objects, such as `WireFrame` and `RotateShape`, are wrapped around the shape, using the following syntax.

```
Show[WireFrame[Graphics3D[Torus[1, 0.75`, 25, 25]]], Boxed → False]
```

Here we load the package in which various polyhedra are defined.

```
Needs["PolyhedronOperations`"]
```

We can now use the standard polyhedra as graphics primitives.

```
Show[PolyhedronData["GreatIcosahedron"]]
```

See also `Geometry`Polytopes``

■ Exercises: Shapes and Polyhedra

Have *Mathematica* draw the default `Cylinder`. Draw another cylinder whose length is twice its diameter. Use `RotateShape` to rotate this `Cylinder` off the vertical. Use `AffineShape` to deform the cylinder in any way.

Here we load the package `Graphics`Shapes``.

```
Needs["Graphics`Shapes`"]
```

Here is the default cylinder.

```
Show[Graphics3D[Cylinder[]], Boxed → False]
```

Here is a cylinder with a height twice its diameter.

```
Show[Graphics3D[Cylinder[1, 2]], Boxed → False]
```

Here we use `RotateShape` to rotate the cylinder.

```
Show[Graphics3D[RotateShape[Cylinder[1, 2], 0, π/2, π/2]],
 Boxed → False]
```

Here we deform the cylinder with `AffineShape`.

```
Show[Graphics3D[AffineShape[Cylinder[1, 2], {0.25`, 0.5`, 0.15`}]],
 Boxed → False]
```

Do the previous exercise with any other shape.

```
Show[Graphics3D[RotateShape[AffineShape[Cone[], {0.25`, 0.5`, 1.15`}],
   π, π/3, -π/4]], Boxed → False]
```

Customizing Graphics

■ Arrows

Now that we know how to use graphics primitives to create arbitrary graphics objects, we can combine these with other types of graphics.

Epilog is an option for most plotting functions that allows us to specify graphics directives and primitives to be drawn after the main graphics are generated.

We can use **Epilog** to add arbitrary graphics to any plot. The standard package **Graphics`Arrow`** defines an arrow graphics primitive. We load it in the standard way, using **Needs**.

```
Needs["Graphics`Arrow`"]
```

Arrow[*start***,** *finish***]** is a graphics primitive representing an arrow starting at the point *start* and ending at the point *finish*.

```
Plot[Sin[x]/x, {x, -10, 10}, PlotRange → All, AxesLabel → {"x", "y"},
 Epilog → {Arrow[{7, 0.6`}, {0.01`, 0.99`}],
   Arrow[{7, 0.4`}, {4.49`, -0.21`}]}]
```

See also **Prolog**

■ Text in Graphs

Text[*expr***,** *coords***]** is a graphics primitive that represents text corresponding to the printed form of *expr*, centered at the point specified by *coords*.

```
Show[Graphics[Text["look here", {0, 0}]]]
```

We can use text as part of a list of graphics primitives given to the option **Epilog**.

```
Plot[Sin[x]/x, {x, -10, 10}, PlotRange → All, AxesLabel → {"x", "y"},
 Epilog → {Arrow[{7, 0.6`}, {0.01`, 0.99`}],
   Arrow[{7, 0.4`}, {4.49`, -0.21`}], Text["some extrema", {7, 0.5`}]}]
```

The option **TextStyle** accepts a list of options that change the font used for all text in a graph, as well as its size, color, weight, and slant. (The specific options are **FontFamily**, **FontSize**, **FontColor**, **FontWeight**, and **FontSlant**.) In the following example all text is in 9-point Helvetica, drawn in 50% gray.

```
Plot[Sin[x]/x, {x, -10, 10}, PlotRange → All, AxesLabel → {"x", "y"},
 Epilog → {Arrow[{7, 0.6`}, {0.01`, 0.99`}],
   Arrow[{7, 0.4`}, {4.49`, -0.21`}], Text["some extrema", {7, 0.5`}]},
 BaseStyle → {FontFamily → "Helvetica", FontSize → 9,
   FontColor → GrayLevel[0.5`]}]
```

We can change the font styles for a particular piece of text by putting the text inside **StyleForm** and including the desired changes. In this example all settings are the same as above, except the text “some extrema” is drawn in red 12-point bold Times.

```
Plot[Sin[x]/x, {x, -10, 10}, PlotRange → All, AxesLabel → {"x", "y"},
 Epilog → {Arrow[{7, 0.6`}, {0.01`, 0.99`}],
   Arrow[{7, 0.4`}, {4.49`, -0.21`}],
   Text[Style["some extrema", FontFamily → "Times", FontSize → 12,
     FontWeight → "Bold", FontColor → Hue[0]], {7, 0.5`}]},
 BaseStyle → {FontFamily → "Helvetica", FontSize → 9,
   FontColor → GrayLevel[0.5`]}]
```

See also **$TextStyle**, **FormatType**, **$FormatType**, **Background**

■ Graphics Formats

By default *Mathematica* generates graphics using a subset of the PostScript language, which is transportable among all types of computers *Mathematica* runs on. Graphics in the PostScript language can be enlarged or reduced to any size without loss of resolution.

The function **DisplayString** allows us to see the PostScript code that makes up a graph. We can save the PostScript to a file that can be read by many of the highest-quality graphics processors. Here is a two-dimensionalplot.

```
sinplot = Plot[Sin[x], {x, -3, 3}]
```

The PostScript code that makes up the graph is rather long, so here we use **StringTake** to show only the first 150 characters.

```
StringTake[ExportString[sinplot, "EPS"], 150]
```

The function **Display** will save a graph in a file. Here we save the graph called **sinplot** into a file called **sinfile**.

```
Export["sinfile", sinplot]
```

Display["*filename***",** *graphics***, "***format***"]** saves a graph to a file after converting the graph to another format. Some of the possible values for the parameter *format* are **GIF**, **EPS**, **Illustrator**, **Metafile**, **PICT**, **TIFF**, and **XBitmap**.

There is also a package **Utilities`DXF`** that saves three-dimensionalgraphics in DXF format, the standard used in AutoCad and other modeling programs.

The *Mathematica* front end will convert graphics to several formats. We select a graphic, then pull down the Edit menu and choose a format from the Copy As submenu to copy the graph to the system clipboard in the specified form, or choose a format from the Save As submenu to save the converted graph to a file.

See also **`Graphics`ThreeScript``**, **`ImageSize`**

■ Exercises: Customizing Graphics

Scroll back in this notebook and copy the code used to generate the last graphic in the two-dimensional graphics section. Use `Arrow` and `Text` to draw an arrow pointing at an arbitrary point on the graphic with the caption “Look Here!”

```
Plot[x^2 - 20 Cos[x^2], {x, -10, 10}, PlotStyle → {Green},
 PlotPoints → 75, Frame → True, GridLines → Automatic,
 PlotLabel → "Exercise One",
 Epilog → {Arrow[{5, 62}, {0, -19}], Text["Look Here!", {3, 64}]}]
```

Animation

All versions of *Mathematica* can create animations. Animation results when a series of *Mathematica* graphics are displayed quickly in succession to create the illusion of smooth movement.

Mathematica provides many features to aid this process. Here is a simple example. The command **`Table`** creates an array of results by iterating commands. Here we will create ten different plots of sin($a\,x$), letting a vary. Notice that in the next example we explicitly set the value for the option **`PlotRange`** because by default *Mathematica* picks a new value for **`PlotRange`** for each frame of the animation, causing the axes to move from one frame to the next.

```
Table[Plot[Sin[a x], {x, 0, 10}, PlotRange → {{0, 10}, {-1, 1}}],
    {a, 1, 5, 0.5}];
```

(The cells of the animation have been deleted to save space. Enter the code into *Mathematica* to see the animation.)

The package **`Graphics`Animation`** defines several functions for automating the creation of animations. Here we load the package.

```
Needs["Graphics`Animation`"]
```

Here are the names of all the functions defined in the package.

```
?Graphics`Animation`*
```

MoviePlot[f[x**,**t**],{**x**,**x_0**,**x_1**},{**t**,**t_0**,**t_1**}]** animates plots of **f[**x**,**t**]** regarded as a function of x, with t serving as the animation, or time, variable.

```
MoviePlot[Sin[a x], {x, 0, 10}, {a, 1, 5, 0.5}];
```

(The cells of the animation have been deleted to save space. Enter the code into *Mathematica* to see the animation.)

Notice that `MoviePlot` is essentially a shortcut for using the `Table` command. One difference is that `MoviePlot` automatically uses the same value for `PlotRange` for each frame.

Another interesting animation results from varying the viewpoint, thereby creating a revolution or a "fly-by" of an object. `SpinShow` automates this process.

```
SpinShow[Graphics3D[Stellate[Icosahedron[]]], Boxed → False];
```

(The cells of the animation have been deleted to save space. Enter the code into *Mathematica* to see the animation.)

Other effects can be achieved by varying colors, options, ranges, and so on.

Exercises: Animation

Using `Table`, `Do`, or `MoviePlot`, create a two-dimensional animation of a function that changes over time. Ensure that the domain and range remain the same throughout the animation.

```
myanim = Table[Plot[Sin[k x], {x, 0, 3 π}, PlotRange → {{0, 10}, {-1, 1}},
    PlotPoints → 50], {k, 1, 9}];
```

The line above will generate the animation, but for better viewing on the printed page we use `GraphicsArray` to view all the frames at once.

```
Show[GraphicsGrid[Partition[myanim, 3]]]
```

Choose your favorite three-dimensional graphic from this set of exercises (or make a new one) and create an animation using `SpinShow`.

To use `SpinShow`, we must load the package `Graphics`Animation``.

```
Needs["Graphics`Animation`"]
```

$$\texttt{simpleplot3d = Plot3D}\left[\frac{1}{\texttt{Abs}\left[(x + i\, y)^5 - 1\right]}, \{x, -1, 1\}, \{y, -1, 1\}\right]$$

```
my3danim = SpinShow[simpleplot3d, Frames → 12];
```

Using the same technique as above, we look at all of the frames at once.

```
Show[GraphicsGrid[Partition[my3danim, 3]]]
```

Additional Topics

This section is a sampler, rather than a tutorial, of a miscellany of *Mathematica*'s capabilities. Documentation for the commands used here are found in *The Mathematica Book*, *Standard Add-on Packages*, and the Help Browser.

Mathematica and Files

Mathematica contains a virtual operating system with which we can navigate directories and list their contents, as well as create, delete, and get information about files. Here we get the name of the current working directory, where any files we create will be saved.

```
Directory[]
```

Here we get a list of all files and subdirectories found in our current working directory.

```
FileNames[]
```

Mathematica contains functions for reading and writing many kinds of data files. We can read and write numbers, strings, lists, *Mathematica* expressions, or anything else.

To illustrate, here we create a file called `datafile` in which to write data, by using **`OpenWrite`**.

```
? OpenWrite
```

```
stream = OpenWrite["datafile"]
```

Here we set up a loop that writes twelve random numbers to `datafile`.

```
Do[Write[stream, RandomReal[]], {12}]
```

When finished writing data, we close the file.

```
Close[stream]
```

Here we display the file, using `!!datafile`.

```
!!datafile
```

In order to use the data in computations, here we read the contents of `datafile`, putting the contents in a list called **`somedata`** using **`ReadList`**.

```
somedata = ReadList["datafile"];
```

The list can now be treated as a list generated any other way. For example, we can plot the list or sort its elements.

```
ListPlot[somedata, Joined → True]
```

```
ListPlot[Sort[somedata], Joined → True]
```

Moreover, we can specify to **`ReadList`** the type of data we wish to read. For example, suppose we want to read `datafile` as six ordered pairs of data, rather than twelve data values. To do this, we specify to **`ReadList`** that we are reading data of the form `{Number, Number}`.

```
datapairs = ReadList["datafile", {Number, Number}];
```

```
ListPlot[datapairs, Joined → True]
```

Similarly, we can read `datafile` as a list of ordered triples.

```
datatriples = ReadList["datafile", {Number, Number, Number}];
```

We now plot the ordered triples.

```
Show[Graphics3D[Line[datatriples]]]
```

See also **`Read`**, **`RecordSeparators`**, **`Utilities`BinaryFiles``**

Statistics and Data Analysis

To use *Mathematica*'s statistical functions, we first load the appropriate packages from the Statistics directory. The package names can be found in the Help Browser or the book *Standard Add-on Packages*, and the packages can be loaded using the Help Browser or the **`Needs`** command.

Mathematica knows about many continuous and discrete statistical distributions. We first load the package containing the continuous distributions.

```
Needs["Statistics`ContinuousDistributions`"]
```

Once the package is loaded, on-line help for the statistics functions is available.

```
?PDF
```

The loaded functions can then be plotted or manipulated in the usual ways. Below we generate two statistical plots. Here is a plot of the p.d.f. of the standard normal distribution and the p.d.f. of an extreme-value distribution.

```
Plot[{PDF[NormalDistribution[0, 1], x],
  PDF[ExtremeValueDistribution[-1, 1], x]}, {x, -3, 3}]
```

Here is a three-dimensional plot of binomial coefficients.

```
ListPlot3D[Table[Binomial[m, n], {m, 1, 6}, {n, 1, 6}]]
```

Here we load a package that computes descriptive statistics from lists of data.

```
Needs["Statistics`DescriptiveStatistics`"]
```

The functions defined in the package can be used on numeric lists of data.

```
Mean[{0.1, 1, 10, 100, 1000}]
```

They can also be used on symbolic lists of data.

```
HarmonicMean[{a, b, c}]
```

```
Skewness[{a, b}]
```

The **Fit** function built into *Mathematica* performs least-squares fitting to a list of data. If we want to fit to data a function that is not a linear combination of the basis functions, we need to load the package **Statistics`NonlinearFit`**.

```
Needs["NonlinearRegression`"]
```

```
?NonlinearFit
```

For this example we generate a list of points and call it **dataToFit**.

```
dataToFit = Table[{x, N[3 Sin[7 x/4] + RandomReal[]/3 - 1/6]}, {x, 0, 3, 1/4}];
```

Here is a graph of the points.

```
ListPlot[dataToFit, PlotStyle → PointSize[0.02`]]
```

Here we use **`NonlinearFit`** to fit our model (which is not a linear combination of basis functions) to the list **`dataToFit`**. Here is the resulting model.

```
nlmodel = NonlinearFit[dataToFit, b Sin[a x], x, {a, b}]
```

Here we plot the nonlinear model called **`nlmodel`**.

```
Plot[nlmodel, {x, 0, 3}, Epilog → {PointSize[0.02`], Point /@ dataToFit}]
```

To augment the built-in (least-squares) **`Fit`** function, a package exists to perform full linear regression on a set of data. We load the appropriate package.

```
Needs["LinearRegression`"]
```

Now we perform the linear regression, fitting **`dataToFit`** with constant, linear, quadratic, and cubic basis functions.

```
NotebookCompatibility`Dump`LinearModelFit[dataToFit, {1, x, x^2, x^3},
  x][{"ParameterTable", "RSquared", "AdjustedRSquared",
  "EstimatedVariance", "ANOVATable"}]
```

Part of the default result is an ANOVAtable. There are options we can set to obtain covariance and correlation matrices, residual tables, confidence intervals, and more. If all we want is the fitted function, we can use *Mathematica*'s built-in **`Fit`** function.

```
Fit[dataToFit, {1, x, x^2, x^3}, x]
```

We plot the fitted function and again compare it to the given points.

```
Plot[%, {x, 0, 3}, Epilog → {PointSize[0.02`], Point /@ dataToFit}]
```

Additional Graphics Functions

Other specialized graphics functions are contained in the packages.

```
(Needs["BarCharts`"]; Needs["Histograms`"]; Needs["PieCharts`"]);
```

While not *Mathematica*'s main purpose, it can create most of the common business graphics. As an example, **BarChart** is an easy function to start with.

```
BarChart[Prime[Range[5]], PlotLabel → "primes"]
```

There are several log-plotting functions available. The list generated below shows eight different log plots, including functions for data plotting.

```
? *Log*Plot*
```

$$\text{LogPlot}\left[\text{Abs}\left[\frac{100}{(i\,x)^2+2\,i\,x+100}\right], \{x, 1, 20\}, \text{GridLines} \to \{\text{LogGridMajor}, \text{Automatic}\}\right]$$

Many variants of standard plots are defined in *Mathematica*. To visualize the space between curves, we use **FilledPlot**.

```
Needs["Graphics`FilledPlot`"]
```

$$\text{Plot}\left[\left\{\frac{\text{Sin}[t]}{t}, \text{BesselJ}[1, t]\right\}, \{t, -10, 10\}, \text{Filling} \to \{1 \to \{2\}\}\right];$$

See also **PieChart**

System Parameters

Mathematica contains many global and system parameters, most of which begin with **$**. Here is a list of the system parameters beginning with **$M**.

```
Names["$M*"]
```

The on-line help tells us what each parameter is.

```
? $MaxNumber
```

To find the setting or value of a parameter, type its name and press SHIFT-[RET].

```
$MaxNumber
```

```
$MachineType
```

```
$Version
```

Here is a list containing the date and time at which this notebook was evaluated.

```
DateList[]
```

Here is the number of seconds that have elapsed since the turn of the century.

```
AbsoluteTime[]
```