Wolfram *Mathematica*® Tutorial Collection

MATHEMATICS AND ALGORITHMS

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Numerical Operations on Data

Mathematical Functions

Numbers

Types of Numbers

Four underlying types of numbers are built into *Mathematica*.

`Integer`	arbitrary-length exact integer
`Rational`	*integer*/*integer* in lowest terms
`Real`	approximate real number, with any specified precision
`Complex`	complex number of the form *number* + *number* `I`

Intrinsic types of numbers in *Mathematica*.

Rational numbers always consist of a ratio of two integers, reduced to lowest terms.

```
In[1]:= 12 344 / 2222
```

$$\text{Out[1]}= \frac{6172}{1111}$$

Approximate real numbers are distinguished by the presence of an explicit decimal point.

```
In[2]:= 5456.
Out[2]= 5456.
```

An approximate real number can have any number of digits.

```
In[3]:= 4.5454352345454352345345234523454 3
Out[3]= 4.54543523454543523453452345234543
```

Complex numbers can have integer or rational components.

```
In[4]:= 4 + 7 / 8 I
```

$$\text{Out[4]}= 4 + \frac{7\,i}{8}$$

They can also have approximate real number components.

```
In[5]:= 4 + 5.6 I
Out[5]= 4 + 5.6 i
```

`123`	an exact integer
`123.`	an approximate real number
`123.0000000000000`	an approximate real number with a certain precision
`123.+0.I`	a complex number with approximate real number components

Several versions of the number 123.

You can distinguish different types of numbers in *Mathematica* by looking at their heads. (Although numbers in *Mathematica* have heads like other expressions, they do not have explicit elements which you can extract.)

The object `123` is taken to be an exact integer, with head `Integer`.

```
In[6]:= Head[123]
Out[6]= Integer
```

The presence of an explicit decimal point makes *Mathematica* treat `123.` as an approximate real number, with head `Real`.

```
In[7]:= Head[123.]
Out[7]= Real
```

`NumberQ[`x`]`	test whether x is any kind of number
`IntegerQ[`x`]`	test whether x is an integer
`EvenQ[`x`]`	test whether x is even
`OddQ[`x`]`	test whether x is odd
`PrimeQ[`x`]`	test whether x is a prime integer
`Head[`x`]===`*type*	test the type of a number

Tests for different types of numbers.

`NumberQ[`x`]` tests for any kind of number.

```
In[8]:= NumberQ[5.6]
Out[8]= True
```

`5.` is treated as a `Real`, so `IntegerQ` gives `False`.

```
In[9]:= IntegerQ[5.]
Out[9]= False
```

If you use complex numbers extensively, there is one subtlety you should be aware of. When you enter a number like `123.`, *Mathematica* treats it as an approximate real number, but assumes that its imaginary part is exactly zero. Sometimes you may want to enter approximate complex numbers with imaginary parts that are zero, but only to a certain precision.

When the imaginary part is the exact integer `0`, *Mathematica* simplifies complex numbers to real ones.

```
In[10]:= Head[123 + 0 I]
Out[10]= Integer
```

Here the imaginary part is only zero to a certain precision, so *Mathematica* retains the complex number form.

```
In[11]:= Head[123. + 0. I]
Out[11]= Complex
```

The distinction between complex numbers whose imaginary parts are exactly zero, or are only zero to a certain precision, may seem like a pedantic one. However, when we discuss, for example, the interpretation of powers and roots of complex numbers in "Functions That Do Not Have Unique Values", the distinction will become significant.

One way to find out the type of a number in *Mathematica* is just to pick out its head using `Head[`*expr*`]`. For many purposes, however, it is better to use functions like `IntegerQ` which explicitly test for particular types. Functions like this are set up to return `True` if their argument is manifestly of the required type, and to return `False` otherwise. As a result, `IntegerQ[x]` will give `False`, unless `x` has an explicit integer value.

Complex Numbers

You can enter complex numbers in *Mathematica* just by including the constant `I`, equal to $\sqrt{-1}$. Make sure that you type a capital `I`.

If you are using notebooks, you can also enter `I` as i by typing Esc `ii` Esc (see "Mathematical Notation in Notebooks: Numerical Calculations"). The form i is normally what is used in output. Note that an ordinary `i` means a variable named i, not $\sqrt{-1}$.

This gives the imaginary number result $2\,i$.

```
In[1]:= Sqrt[-4]
Out[1]= 2 i
```

This gives the ratio of two complex numbers.

```
In[2]:= (4 + 3 I) / (2 - I)
Out[2]= 1 + 2 i
```

Here is the numerical value of a complex exponential.

```
In[3]:= Exp[2 + 9 I] // N
Out[3]= -6.73239 + 3.04517 i
```

x+I y	the complex number $x + i\,y$
Re[z]	real part
Im[z]	imaginary part
Conjugate[z]	complex conjugate z^* or $\overline{z}$
Abs[z]	absolute value $\lvert z \rvert$
Arg[z]	the argument φ in $\lvert z \rvert e^{i\,\varphi}$

Complex number operations.

Numeric Quantities

NumberQ[*expr*]	test whether *expr* is explicitly a number
NumericQ[*expr*]	test whether *expr* has a numerical value

Testing for numeric quantities.

Pi is a symbol, so Pi + 3 is not explicitly a number.

```
In[1]:= NumberQ[Pi + 3]
Out[1]= False
```

It does however have a numerical value.

```
In[2]:= NumericQ[Pi + 3]
Out[2]= True
```

This finds the explicit numerical value of `Pi + 3`.

```
In[3]:= N[Pi + 3]
Out[3]= 6.14159
```

Mathematica knows that constants such as `Pi` are numeric quantities. It also knows that standard mathematical functions such as `Log` and `Sin` have numerical values when their arguments are numerical.

`Log[2 + x]` contains `x`, and is therefore not a numeric quantity.

```
In[4]:= {NumericQ[Log[2]], NumericQ[Log[2 + x]]}
Out[4]= {True, False}
```

Many functions implicitly use the numerical values of numeric quantities.

```
In[5]:= Min[Exp[2], Log[2], Sqrt[2]]
Out[5]= Log[2]
```

In general, *Mathematica* assumes that any function which has the attribute `NumericFunction` will yield numerical values when its arguments are numerical. All standard mathematical functions in *Mathematica* already have this attribute. But when you define your own functions, you can explicitly set the attribute to tell *Mathematica* to assume that these functions will have numerical values when their arguments are numerical.

Digits in Numbers

`IntegerDigits[`n`]`	a list of the decimal digits in the integer n
`IntegerDigits[`n`,`b`]`	the digits of n in base b
`IntegerDigits[`n`,`b`,`len`]`	the list of digits padded on the left with zeros to give total length len
`IntegerLength[`n`]`	the number of decimal digits in n
`IntegerLength[`n`,`b`]`	the number of base b digits in n
`IntegerExponent[`n`,`b`]`	the number of zeros at the end of n in base b
`RealDigits[`x`]`	a list of the decimal digits in the approximate real number x, together with the number of digits to the left of the decimal point

`RealDigits[`*x*`,`*b*`]`	the digits of *x* in base *b*
`RealDigits[`*x*`,`*b*`,`*len*`]`	the first *len* digits of *x* in base *b*
`RealDigits[`*x*`,`*b*`,`*len*`,`*n*`]`	the first *len* digits starting with the coefficient of b^n
`FromDigits[`*list*`]`	construct a number from its decimal digit sequence
`FromDigits[`*list*`,`*b*`]`	construct a number from its digit sequence in base *b*
`FromDigits["`*string*`"]`	construct an integer from a string of digits
`FromDigits["`*string*`",`*b*`]`	construct an integer from a string of digits in base *b*
`IntegerString[`*n*`]`	a string of the decimal digits in the integer *n*
`IntegerString[`*n*`,`*b*`]`	a string of the digits of *n* in base *b*

Converting between numbers and lists or strings of digits.

Here is the list of base 16 digits for an integer.

```
In[1]:= IntegerDigits[1 234 135 634, 16]
Out[1]= {4, 9, 8, 15, 6, 10, 5, 2}
```

This gives a list of digits, together with the number of digits that appear to the left of the decimal point.

```
In[2]:= RealDigits[123.4567890123456]
Out[2]= {{1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6}, 3}
```

Here is the binary digit sequence for 56, padded with zeros so that it is of total length 8.

```
In[3]:= IntegerDigits[56, 2, 8]
Out[3]= {0, 0, 1, 1, 1, 0, 0, 0}
```

This reconstructs the original number from its binary digit sequence.

```
In[4]:= FromDigits[%, 2]
Out[4]= 56
```

Here is 56 as a binary string.

```
In[5]:= IntegerString[56, 2]
Out[5]= 111000
```

This reconstructs the original number again.

```
In[6]:= FromDigits[%, 2]
Out[6]= 56
```

b^^*nnnn*	a number in base *b*
BaseForm[*x*,*b*]	print with *x* in base *b*
IntegerString[*n*,*b*]	a string representing *n* in base *b*

Numbers in other bases.

When the base is larger than 10, extra digits are represented by letters a-z.

The number $100\,101_2$ in base 2 is 37 in base 10.

```
In[7]:= 2^^100101
Out[7]= 37
```

This prints 37 in base 2.

```
In[8]:= BaseForm[37, 2]
Out[8]//BaseForm= 100101_2
```

This gives the base-2 representation as a string.

```
In[9]:= IntegerString[37, 2]
Out[9]= 100101
```

Here is a number in base 16.

```
In[10]:= 16^^ffffaa00
Out[10]= 4 294 945 280
```

You can do computations with numbers in base 16. Here the result is given in base 10.

```
In[11]:= 16^^fffaa2 + 16^^ff - 1
Out[11]= 16 776 096
```

This gives the result in base 16.

```
In[12]:= BaseForm[%, 16]
Out[12]//BaseForm= fffba0_16
```

You can give approximate real numbers, as well as integers, in other bases.

```
In[13]:= 2^^101.100101
Out[13]= 5.57813
```

Here are the first few digits of $\sqrt{2}$ in octal.

```
In[14]:= BaseForm[N[Sqrt[2], 30], 8]
Out[14]//BaseForm= 1.32404746317716746220426276611546 7_8
```

This gives an explicit list of the first 15 octal digits.

```
In[15]:= RealDigits[Sqrt[2], 8, 15]
Out[15]= {{1, 3, 2, 4, 0, 4, 7, 4, 6, 3, 1, 7, 7, 1, 6}, 1}
```

This gives 15 octal digits starting with the coefficient of 8^{-10}.

```
In[16]:= RealDigits[Sqrt[2], 8, 15, -10]
Out[16]= {{1, 7, 7, 1, 6, 7, 4, 6, 2, 2, 0, 4, 2, 6, 2}, -9}
```

"Output Formats for Numbers" describes how to print numbers in various formats. If you want to create your own formats, you will often need to use `MantissaExponent` to separate the pieces of real numbers.

`MantissaExponent[`x`]`	give a list containing the mantissa and exponent of x
`MantissaExponent[`x`,`b`]`	give the mantissa and exponent in base b

Separating the mantissa and exponent of numbers.

This gives a list in which the mantissa and exponent of the number are separated.

```
In[17]:= MantissaExponent[3.45 × 10^125]
Out[17]= {0.345, 126}
```

Exact and Approximate Results

A standard electronic calculator does all your calculations to a particular accuracy, say 10 decimal digits. With *Mathematica*, however, you can often get *exact* results.

Mathematica gives an *exact* result for 2^{100}, even though it has 31 decimal digits.

```
In[1]:= 2^100
Out[1]= 1 267 650 600 228 229 401 496 703 205 376
```

You can tell *Mathematica* to give you an approximate numerical result, just as a calculator would, by ending your input with `//N`. The `N` stands for "numerical". It must be a capital letter. "Special Ways to Input Expressions" will explain what the `//` means.

This gives an approximate numerical result.

```
In[2]:= 2^100 //N
Out[2]= 1.26765×10^30
```

Mathematica can give results in terms of rational numbers.

```
In[3]:= 1/3+2/7
Out[3]= 13/21
```

`//N` always gives the approximate numerical result.

```
In[4]:= 1/3+2/7//N
Out[4]= 0.619048
```

expr`//N`	give an approximate numerical value for *expr*

Getting numerical approximations.

When you type in an integer like `7`, *Mathematica* assumes that it is exact. If you type in a number like `4.5`, with an explicit decimal point, *Mathematica* assumes that it is accurate only to a fixed number of decimal places.

This is taken to be an exact rational number, and reduced to its lowest terms.

```
In[5]:= 452/62
Out[5]= 226/31
```

Whenever you give a number with an explicit decimal point, *Mathematica* produces an approximate numerical result.

```
In[6]:= 452.3/62
Out[6]= 7.29516
```

Here again, the presence of the decimal point makes *Mathematica* give you an approximate numerical result.

```
In[7]:= 452. / 62
Out[7]= 7.29032
```

When any number in an arithmetic expression is given with an explicit decimal point, you get an approximate numerical result for the whole expression.

```
In[8]:= 1. + 452 / 62
Out[8]= 8.29032
```

Numerical Precision

As discussed in "Exact and Approximate Results", *Mathematica* can handle approximate real numbers with any number of digits. In general, the *precision* of an approximate real number is the effective number of decimal digits in it that are treated as significant for computations. The *accuracy* is the effective number of these digits that appear to the right of the decimal point. Note that to achieve full consistency in the treatment of numbers, precision and accuracy often have values that do not correspond to integer numbers of digits.

`Precision[`x`]`	the total number of significant decimal digits in x
`Accuracy[`x`]`	the number of significant decimal digits to the right of the decimal point in x

Precision and accuracy of real numbers.

This generates a number with 30-digit precision.

```
In[1]:= x = N[Pi^10, 30]
Out[1]= 93 648.047476083020 9737166901849
```

This gives the precision of the number.

```
In[2]:= Precision[x]
Out[2]= 30.
```

The accuracy is lower since only some of the digits are to the right of the decimal point.

```
In[3]:= Accuracy[x]
Out[3]= 25.0285
```

This number has all its digits to the right of the decimal point.

```
In[4]:= x / 10^6
Out[4]= 0.0936480474760830209737166901849
```

Now the accuracy is larger than the precision.

```
In[5]:= {Precision[%], Accuracy[%]}
Out[5]= {30., 31.0285}
```

An approximate real number always has some uncertainty in its value, associated with digits beyond those known. One can think of precision as providing a measure of the relative size of this uncertainty. Accuracy gives a measure of the absolute size of the uncertainty.

Mathematica is set up so that if a number x has uncertainty δ, then its true value can lie anywhere in an interval of size δ from $x - \delta/2$ to $x + \delta/2$. An approximate number with accuracy a is defined to have uncertainty 10^{-a}, while a nonzero approximate number with precision p is defined to have uncertainty $|x|\,10^{-p}$.

`Precision[`x`]`	$-\log_{10}(\delta/\lvert x\rvert)$
`Accuracy[`x`]`	$-\log_{10}(\delta)$

Definitions of precision and accuracy in terms of uncertainty.

Adding or subtracting a quantity smaller than the uncertainty has no visible effect.

```
In[6]:= {x - 10^-26, x, x + 10^-26}
Out[6]= {93 648.047476083020973716690 1849,
 93 648.047476083020973716690 1849, 93 648.047476083020973716690 1849}
```

`N[`*expr*`,`n`]`	evaluate *expr* to n-digit precision using arbitrary-precision numbers
`N[`*expr*`]`	evaluate *expr* numerically using machine-precision numbers

Numerical evaluation with arbitrary-precision and machine-precision numbers.

Mathematica distinguishes two kinds of approximate real numbers: *arbitrary-precision* numbers, and *machine-precision* numbers or *machine numbers*. Arbitrary-precision numbers can contain any number of digits, and maintain information on their precision. Machine numbers, on the other hand, always contain the same number of digits, and maintain no information on their precision.

Here is a machine-number approximation to π.

```
In[7]:= N[Pi]
Out[7]= 3.14159
```

These are both arbitrary-precision numbers.

```
In[8]:= {N[Pi, 4], N[Pi, 20]}
Out[8]= {3.142, 3.1415926535897932385}
```

As discussed in more detail below, machine numbers work by making direct use of the numerical capabilities of your underlying computer system. As a result, computations with them can often be done more quickly. They are however much less flexible than arbitrary-precision numbers, and difficult numerical analysis can be needed to determine whether results obtained with them are correct.

`MachinePrecision`	the precision specification used to indicate machine numbers
`$MachinePrecision`	the effective precision for machine numbers on your computer system
`MachineNumberQ[`x`]`	test whether x is a machine number

Machine numbers.

This returns the symbol `MachinePrecision` to indicate a machine number.

```
In[9]:= Precision[N[Pi]]
Out[9]= MachinePrecision
```

On this computer, machine numbers have slightly less than 16 decimal digits.

```
In[10]:= $MachinePrecision
Out[10]= 15.9546
```

When you enter an approximate real number, *Mathematica* has to decide whether to treat it as a machine number or an arbitrary-precision number. Unless you specify otherwise, if you give less than `$MachinePrecision` digits, *Mathematica* will treat the number as machine precision, and if you give more digits, it will treat the number as arbitrary precision.

`123.4`	a machine-precision number
`123.45678901234567890`	an arbitrary-precision number on some computer systems
`123.45678901234567890``	a machine-precision number on all computer systems
`123.456`200`	an arbitrary-precision number with 200 digits of precision
`123.456``200`	an arbitrary-precision number with 200 digits of accuracy
`1.234*^6`	a machine-precision number in scientific notation (1.234×10^6)
`1.234`200*^6`	a number in scientific notation with 200 digits of precision
`2^^101.111`200`	a number in base 2 with 200 binary digits of precision
`2^^101.111`200*^6`	a number in base-2 scientific notation ($101.111_2\times2^6$)

Input forms for numbers.

When *Mathematica* prints out numbers, it usually tries to give them in a form that will be as easy as possible to read. But if you want to take numbers that are printed out by *Mathematica*, and then later use them as input to *Mathematica*, you need to make sure that no information gets lost.

In standard output form, *Mathematica* prints a number like this to six digits.

```
In[11]:= N[Pi]
Out[11]= 3.14159
```

In input form, *Mathematica* prints all the digits it knows.

```
In[12]:= InputForm[%]
Out[12]//InputForm= 3.141592653589793
```

Here is an arbitrary-precision number in standard output form.

```
In[13]:= N[Pi, 20]
Out[13]= 3.1415926535897932385
```

In input form, *Mathematica* explicitly indicates the precision of the number, and gives extra digits to make sure the number can be reconstructed correctly.

```
In[14]:= InputForm[%]
Out[14]//InputForm= 3.14159265358979323846264338327950288 42`20.
```

This makes *Mathematica* not explicitly indicate precision.

```
In[15]:= InputForm[%, NumberMarks -> False]
Out[15]//InputForm= 3.14159265358979323846
```

`InputForm[`*expr*`, NumberMarks->True]`	use ` marks in all approximate numbers
`InputForm[`*expr*`, NumberMarks->Automatic]`	use ` only in arbitrary-precision numbers
`InputForm[`*expr*`, NumberMarks->False]`	never use ` marks

Controlling printing of numbers.

The default setting for the `NumberMarks` option, both in `InputForm` and in functions such as `ToString` and `OpenWrite` is given by the value of `$NumberMarks`. By resetting `$NumberMarks`, therefore, you can globally change the way that numbers are printed in `InputForm`.

This makes *Mathematica* by default always include number marks in input form.

```
In[16]:= $NumberMarks = True
Out[16]= True
```

Even a machine-precision number is now printed with an explicit number mark.

```
In[17]:= InputForm[N[Pi]]
Out[17]//InputForm= 3.141592653589793`
```

Even with no number marks, `InputForm` still uses `*^` for scientific notation.

```
In[18]:= InputForm[N[Exp[600], 20], NumberMarks -> False]
Out[18]//InputForm= 3.7730203009299398234*^260
```

In doing numerical computations, it is inevitable that you will sometimes end up with results that are less precise than you want. Particularly when you get numerical results that are very close to zero, you may well want to *assume* that the results should be exactly zero. The function `Chop` allows you to replace approximate real numbers that are close to zero by the exact integer 0.

Chop[*expr*]	replace all approximate real numbers in *expr* with magnitude less than 10^{-10} by 0
Chop[*expr*, *dx*]	replace numbers with magnitude less than *dx* by 0

Removing numbers close to zero.

This computation gives a small imaginary part.

```
In[19]:= Exp[N[2 Pi I]]
Out[19]= 1. - 2.44921×10^-16 I
```

You can get rid of the imaginary part using Chop.

```
In[20]:= Chop[%]
Out[20]= 1.
```

Arbitrary-Precision Calculations

When you use //N to get a numerical result, *Mathematica* does what a standard calculator would do: it gives you a result to a fixed number of significant figures. You can also tell *Mathematica* exactly how many significant figures to keep in a particular calculation. This allows you to get numerical results in *Mathematica* to any degree of precision.

expr//N or N[*expr*]	approximate numerical value of *expr*
N[*expr*, *n*]	numerical value of *expr* calculated with *n*-digit precision

Numerical evaluation functions.

This gives the numerical value of π to a fixed number of significant digits. Typing N[Pi] is exactly equivalent to Pi // N.

```
In[1]:= N[Pi]
Out[1]= 3.14159
```

This gives π to 40 digits.

```
In[2]:= N[Pi, 40]
Out[2]= 3.1415926535897932384626433832795028841 97
```

Here is $\sqrt{7}$ to 30 digits.

```
In[3]:= N[Sqrt[7], 30]
Out[3]= 2.64575131106459059050161575364
```

Doing any kind of numerical calculation can introduce small roundoff errors into your results. When you increase the numerical precision, these errors typically become correspondingly smaller. Making sure that you get the same answer when you increase numerical precision is often a good way to check your results.

The quantity $e^{\pi\sqrt{163}}$ turns out to be very close to an integer. To check that the result is not, in fact, an integer, you have to use sufficient numerical precision.

```
In[4]:= N[Exp[Pi Sqrt[163]], 40]
Out[4]= 2.625374126407687439999999999992500725972×10^17
```

Arbitrary-Precision Numbers

When you do calculations with arbitrary-precision numbers, *Mathematica* keeps track of precision at all points. In general, *Mathematica* tries to give you results which have the highest possible precision, given the precision of the input you provided.

Mathematica treats arbitrary-precision numbers as representing the values of quantities where a certain number of digits are known, and the rest are unknown. In general, an arbitrary-precision number x is taken to have `Precision[`x`]` digits which are known exactly, followed by an infinite number of digits which are completely unknown.

This computes π to 10-digit precision.

```
In[1]:= N[Pi, 10]
Out[1]= 3.141592654
```

After a certain point, all digits are indeterminate.

```
In[2]:= RealDigits[%, 10, 13]
Out[2]= {{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, Indeterminate, Indeterminate, Indeterminate}, 1}
```

When you do a computation, *Mathematica* keeps track of which digits in your result could be affected by unknown digits in your input. It sets the precision of your result so that no affected

digits are ever included. This procedure ensures that all digits returned by *Mathematica* are correct, whatever the values of the unknown digits may be.

This evaluates $\Gamma(1/7)$ to 30-digit precision.

```
In[3]:= N[Gamma[1 / 7], 30]
Out[3]= 6.54806294024782443771409334943
```

The result has a precision of exactly 30 digits.

```
In[4]:= Precision[%]
Out[4]= 30.
```

If you give input only to a few digits of precision, *Mathematica* cannot give you such high-precision output.

```
In[5]:= N[Gamma[0.142], 30]
Out[5]= 6.58965
```

If you want *Mathematica* to assume that the argument is *exactly* `142 / 1000`, then you have to say so explicitly.

```
In[6]:= N[Gamma[142 / 1000], 30]
Out[6]= 6.58964729492039788328481917496
```

In many computations, the precision of the results you get progressively degrades as a result of "roundoff error". A typical case of this occurs if you subtract two numbers that are close together. The result you get depends on high-order digits in each number, and typically has far fewer digits of precision than either of the original numbers.

Both input numbers have a precision of around 20 digits, but the result has much lower precision.

```
In[7]:= 1.1111111111111111111 - 1.1111111111111111000
Out[7]= 1.1×10^-18
```

Adding extra digits in one number but not the other is not sufficient to allow extra digits to be found in the result.

```
In[8]:= 1.11111111111111111111345 - 1.1111111111111111000
Out[8]= 1.1×10^-18
```

The precision of the output from a function can depend in a complicated way on the precision of the input. Functions that vary rapidly typically give less precise output, since the variation of

the output associated with uncertainties in the input is larger. Functions that are close to constants can actually give output that is more precise than their input.

Here is a case where the output is less precise than the input.

```
In[9]:= Sin[111 111 111.0000000000000000]
Out[9]= -0.2975351033349432
```

Here is e^{-40} evaluated to 20-digit precision.

```
In[10]:= N[Exp[-40], 20]
Out[10]= 4.2483542552915889953×10^-18
```

The result you get by adding the exact integer 1 has a higher precision.

```
In[11]:= 1 + %
Out[11]= 1.0000000000000000042483542552915889953
```

It is worth realizing that different ways of doing the same calculation can end up giving you results with very different precisions. Typically, if you once lose precision in a calculation, it is essentially impossible to regain it; in losing precision, you are effectively losing information about your result.

Here is a 40-digit number that is close to 1.

```
In[12]:= x = N[1 - 10^-30, 40]
Out[12]= 0.9999999999999999999999999999999990000000000
```

Adding 1 to it gives another 40-digit number.

```
In[13]:= 1 + x
Out[13]= 1.9999999999999999999999999999999990000000000
```

The original precision has been maintained.

```
In[14]:= Precision[%]
Out[14]= 40.301
```

This way of computing 1 + x loses precision.

```
In[15]:= (x^2 - 1) / (x - 1)
Out[15]= 2.000000000
```

The result obtained in this way has quite low precision.

```
In[16]:= Precision[%]
Out[16]= 9.69897
```

The fact that different ways of doing the same calculation can give you different numerical answers means, among other things, that comparisons between approximate real numbers must be treated with care. In testing whether two real numbers are "equal", *Mathematica* effectively finds their difference, and tests whether the result is "consistent with zero" to the precision given.

These numbers are equal to the precision given.

```
In[17]:= 3 == 3.000000000000000000
Out[17]= True
```

The internal algorithms that *Mathematica* uses to evaluate mathematical functions are set up to maintain as much precision as possible. In most cases, built-in *Mathematica* functions will give you results that have as much precision as can be justified on the basis of your input. In some cases, however, it is simply impractical to do this, and *Mathematica* will give you results that have lower precision. If you give higher-precision input, *Mathematica* will use higher precision in its internal calculations, and you will usually be able to get a higher-precision result.

`N[`*expr*`]`	evaluate *expr* numerically to machine precision
`N[`*expr*`,`*n*`]`	evaluate *expr* numerically trying to get a result with *n* digits of precision

Numerical evaluation.

If you start with an expression that contains only integers and other exact numeric quantities, then `N[`*expr*`, `*n*`]` will in almost all cases succeed in giving you a result to *n* digits of precision. You should realize, however, that to do this *Mathematica* sometimes has to perform internal intermediate calculations to much higher precision.

The global variable `$MaxExtraPrecision` specifies how many additional digits should be allowed in such intermediate calculations.

variable	*default value*	
`$MaxExtraPrecision`	50	maximum additional precision to use

Controlling precision in intermediate calculations.

Mathematica automatically increases the precision that it uses internally in order to get the correct answer here.

```
In[18]:= N[Sin[10^40], 30]

Out[18]= -0.569633400953636327308034181574
```

Using the default setting `$MaxExtraPrecision = 50`, *Mathematica* cannot get the correct answer here.

```
In[19]:= N[Sin[10^100], 30]

    N::meprec : Internal precision limit $MaxExtraPrecision = 50.` reached while evaluating
        Sin[100000000000000000000000000000000000000000000000000000000000000000000000⋱
            00000000000000000000000000000]. ≫

Out[19]= 0.
```

This tells *Mathematica* that it can use more digits in its internal calculations.

```
In[20]:= $MaxExtraPrecision = 200

Out[20]= 200
```

Now it gets the correct answer.

```
In[21]:= N[Sin[10^100], 30]

Out[21]= -0.372376123661276688262086695553
```

This resets `$MaxExtraPrecision` to its default value.

```
In[22]:= $MaxExtraPrecision = 50

Out[22]= 50
```

Even when you are doing computations that give exact results, *Mathematica* still occasionally uses approximate numbers for some of its internal calculations, so that the value of `$MaxExtraPrecision` can thus have an effect.

Mathematica works this out using bounds from approximate numbers.

```
In[23]:= Sin[Exp[100]] > 0

Out[23]= True
```

With the default value of `$MaxExtraPrecision`, *Mathematica* cannot work this out.

```
In[24]:= Sin[Exp[200]] > 0

    N::meprec : Internal precision limit $MaxExtraPrecision = 50.` reached while evaluating -Sin[ⅇ^200]. ≫

Out[24]= Sin[ⅇ^200] > 0
```

Temporarily resetting `$MaxExtraPrecision` allows *Mathematica* to get the result.

```
In[25]:= Block[{$MaxExtraPrecision = 100}, Sin[Exp[200]] > 0]
Out[25]= False
```

In doing calculations that degrade precision, it is possible to end up with numbers that have no significant digits at all. But even in such cases, *Mathematica* still maintains information on the accuracy of the numbers. Given a number with no significant digits, but accuracy a, *Mathematica* can then still tell that the actual value of the number must be in the range $\{-10^{-a}, +10^{-a}\}/2$. *Mathematica* by default prints such numbers in the form $0. \times 10^{e}$.

Here is a number with 20-digit precision.

```
In[26]:= x = N[Exp[50], 20]
Out[26]= 5.1847055285870724641×10^21
```

Here there are no significant digits left.

```
In[27]:= Sin[x] / x
Out[27]= 0.×10^-22
```

But *Mathematica* still keeps track of the accuracy of the result.

```
In[28]:= Accuracy[%]
Out[28]= 21.7147
```

Adding the result to an exact 1 gives a number with quite high precision.

```
In[29]:= 1 + %%
Out[29]= 1.00000000000000000000
```

One subtlety in characterizing numbers by their precision is that any number that is consistent with zero must be treated as having zero precision. The reason for this is that such a number has no digits that can be recognized as significant, since all its known digits are just zero.

This gives a number whose value is consistent with zero.

```
In[30]:= d = N[Pi, 20] - Pi
Out[30]= 0.×10^-20
```

The number has no recognizable significant digits of precision.

```
In[31]:= Precision[d]
Out[31]= 0.
```

But it still has a definite accuracy, that characterizes the uncertainty in it.

```
In[32]:= Accuracy[d]
Out[32]= 19.5029
```

If you do computations whose results are likely to be near zero, it can be convenient to specify the accuracy, rather than the precision, that you want to get.

N[*expr*, *p*]	evaluate *expr* to precision *p*
N[*expr*, {*p*, *a*}]	evaluate *expr* to at most precision *p* and accuracy *a*
N[*expr*, {Infinity, *a*}]	evaluate *expr* to any precision but to accuracy *a*

Specifying accuracy as well as precision.

Here is a symbolic expression.

```
In[33]:= u = ArcTan[1 / 3] - ArcCot[3]
Out[33]= -ArcCot[3] + ArcTan[1/3]
```

This shows that the expression is equivalent to zero.

```
In[34]:= FullSimplify[u]
Out[34]= 0
```

N cannot guarantee to get a result to precision 20.

```
In[35]:= N[u, 20]
N::meprec :
  Internal precision limit $MaxExtraPrecision = 50.` reached while evaluating -ArcCot[3]+ArcTan[1/3]. >>
Out[35]= 0.×10^-71
```

But it can get a result to accuracy 20.

```
In[36]:= N[u, {Infinity, 20}]
Out[36]= 0.×10^-20
```

When *Mathematica* works out the potential effect of unknown digits in arbitrary-precision numbers, it assumes by default that these digits are completely independent in different numbers. While this assumption will never yield too high a precision in a result, it may lead to unnecessary loss of precision.

In particular, if two numbers are generated in the same way in a computation, some of their unknown digits may be equal. Then, when these numbers are, for example, subtracted, the unknown digits may cancel. By assuming that the unknown digits are always independent, however, *Mathematica* will miss such cancellations.

Here is a number computed to 20-digit precision.

```
In[37]:= d = N[3^-30, 20]
Out[37]= 4.8569357496188611379×10^-15
```

The quantity 1 + d has about 34-digit precision.

```
In[38]:= Precision[1 + d]
Out[38]= 34.3136
```

This quantity has lower precision, since *Mathematica* assumes that the unknown digits in each number d are independent.

```
In[39]:= Precision[(1 + d) - d]
Out[39]= 34.0126
```

Numerical algorithms sometimes rely on cancellations between unknown digits in different numbers yielding results of higher precision. If you can be sure that certain unknown digits will eventually cancel, then you can explicitly introduce fixed digits in place of the unknown ones. You can carry these fixed digits through your computation, then let them cancel, and get a result of higher precision.

`SetPrecision[x,n]`	create a number with n decimal digits of precision, padding with base-2 zeros if necessary
`SetAccuracy[x,n]`	create a number with n decimal digits of accuracy

Functions for modifying precision and accuracy.

This introduces 10 more digits in d.

```
In[40]:= d = SetPrecision[d, 30]
Out[40]= 4.856935749618861137906242664 97×10^-15
```

The digits that were added cancel out here.

```
In[41]:= (1 + d) - d
Out[41]= 1.0000000000000000000000000000000000000000000000
```

The precision of the result is now about 44 digits, rather than 34.

```
In[42]:= Precision[%]
Out[42]= 44.0126
```

`SetPrecision` works by adding digits which are zero in base 2. Sometimes, *Mathematica* stores slightly more digits in an arbitrary-precision number than it displays, and in such cases, `SetPrecision` will use these extra digits before introducing zeros.

This creates a number with a precision of 40 decimal digits. The extra digits come from conversion to base 10.

```
In[43]:= SetPrecision[0.400000000000000, 40]
Out[43]= 0.400000000000000022204460492503130808472 6
```

variable	*default value*	
`$MaxPrecision`	`Infinity`	maximum total precision to be used
`$MinPrecision`	0	minimum precision to be used

Global precision-control parameters.

By making the global assignment `$MinPrecision = `n, you can effectively apply `SetPrecision[`*expr*`, `n`]` at every step in a computation. This means that even when the number of correct digits in an arbitrary-precision number drops below n, the number will always be padded to have n digits.

If you set `$MaxPrecision = `n as well as `$MinPrecision = `n, then you can force all arbitrary-precision numbers to have a fixed precision of n digits. In effect, what this does is to make *Mathematica* treat arbitrary-precision numbers in much the same way as it treats machine numbers—but with more digits of precision.

Fixed-precision computation can make some calculations more efficient, but without careful analysis you can never be sure how many digits are correct in the results you get.

Here is a small number with 20-digit precision.

```
In[44]:= k = N[Exp[-60], 20]
Out[44]= 8.7565107626965203385×10^-27
```

With *Mathematica*'s usual arithmetic, this works fine.

```
In[45]:= Evaluate[1 + k] - 1
Out[45]= 8.7565107626965203385×10^-27
```

This tells *Mathematica* to use fixed-precision arithmetic.

```
In[46]:= $MinPrecision = $MaxPrecision = 20
Out[46]= 20
```

The first few digits are correct, but the rest are wrong.

```
In[47]:= Evaluate[1 + k] - 1
Out[47]= 8.7565107626963908935×10^-27
```

Machine-Precision Numbers

Whenever machine-precision numbers appear in a calculation, the whole calculation is typically done in machine precision. *Mathematica* will then give machine-precision numbers as the result.

Whenever the input contains any machine-precision numbers, *Mathematica* does the computation to machine precision.

```
In[1]:= 1.4444444444444444444^5.7
Out[1]= 8.13382
```

`Zeta[5.6]` yields a machine-precision result, so the `N` is irrelevant.

```
In[2]:= N[Zeta[5.6], 30]
Out[2]= 1.02338
```

This gives a higher-precision result.

```
In[3]:= N[Zeta[56 / 10], 30]
Out[3]= 1.02337547922702991086041788103
```

When you do calculations with arbitrary-precision numbers, as discussed in "Arbitrary-Precision Numbers", *Mathematica* always keeps track of the precision of your results, and gives only those digits which are known to be correct, given the precision of your input. When you do calculations with machine-precision numbers, however, *Mathematica* always gives you a machine-precision result, whether or not all the digits in the result can, in fact, be determined to be correct on the basis of your input.

This subtracts two machine-precision numbers.

```
In[4]:= diff = 1.11111111 - 1.11111000
Out[4]= 1.11×10^-6
```

The result is taken to have machine precision.

```
In[5]:= Precision[diff]
Out[5]= MachinePrecision
```

Here are all the digits in the result.

```
In[6]:= InputForm[diff]
Out[6]//InputForm= 1.1099999999153454`*^-6
```

The fact that you can get spurious digits in machine-precision numerical calculations with *Mathematica* is in many respects quite unsatisfactory. The ultimate reason, however, that *Mathematica* uses fixed precision for these calculations is a matter of computational efficiency.

Mathematica is usually set up to insulate you as much as possible from the details of the computer system you are using. In dealing with machine-precision numbers, you would lose too much, however, if *Mathematica* did not make use of some specific features of your computer.

The important point is that almost all computers have special hardware or microcode for doing floating-point calculations to a particular fixed precision. *Mathematica* makes use of these features when doing machine-precision numerical calculations.

The typical arrangement is that all machine-precision numbers in *Mathematica* are represented as "double-precision floating-point numbers" in the underlying computer system. On most current computers, such numbers contain a total of 64 binary bits, typically yielding 16 decimal digits of mantissa.

The main advantage of using the built-in floating-point capabilities of your computer is speed. Arbitrary-precision numerical calculations, which do not make such direct use of these capabilities, are usually many times slower than machine-precision calculations.

There are several disadvantages of using built-in floating-point capabilities. One already mentioned is that it forces all numbers to have a fixed precision, independent of what precision can be justified for them.

A second disadvantage is that the treatment of machine-precision numbers can vary slightly from one computer system to another. In working with machine-precision numbers, *Mathematica* is at the mercy of the floating-point arithmetic system of each particular computer. If floating-point arithmetic is done differently on two computers, you may get slightly different results for machine-precision *Mathematica* calculations on those computers.

`$MachinePrecision`	the number of decimal digits of precision
`$MachineEpsilon`	the minimum positive machine-precision number which can be added to 1.0 to give a result distinguishable from 1.0
`$MaxMachineNumber`	the maximum machine-precision number
`$MinMachineNumber`	the minimum positive machine-precision number
`$MaxNumber`	the maximum magnitude of an arbitrary-precision number
`$MinNumber`	the minimum magnitude of a positive arbitrary-precision number

Properties of numbers on a particular computer system.

Since machine-precision numbers on any particular computer system are represented by a definite number of binary bits, numbers which are too close together will have the same bit pattern, and so cannot be distinguished. The parameter `$MachineEpsilon` gives the distance between `1.0` and the closest number which has a distinct binary representation.

This gives the value of `$MachineEpsilon` for the computer system on which these examples are run.

```
In[7]:= $MachineEpsilon
Out[7]= 2.22045×10^-16
```

Although this prints as `1.`, *Mathematica* knows that the result is larger than 1.

```
In[8]:= 1. + $MachineEpsilon
Out[8]= 1.
```

InputForm reveals that the result is not exactly 1.

```
In[9]:= % // InputForm
Out[9]//InputForm= 1.0000000000000002
```

Subtracting 1 gives $MachineEpsilon.

```
In[10]:= % - 1.
Out[10]= 2.22045×10^-16
```

This prints as 1. also.

```
In[11]:= 1. + $MachineEpsilon / 2
Out[11]= 1.
```

In this case, however, the result is not distinguished from 1. to machine precision.

```
In[12]:= % // InputForm
Out[12]//InputForm= 1.
```

Subtracting 1 from the result yields 0.

```
In[13]:= % - 1.
Out[13]= 0.
```

Machine numbers have not only limited precision, but also limited magnitude. If you generate a number which lies outside the range specified by $MinMachineNumber and $MaxMachineNumber, *Mathematica* will automatically convert the number to arbitrary-precision form.

This is the maximum machine-precision number which can be handled on the computer system used for this example.

```
In[14]:= $MaxMachineNumber
Out[14]= 1.79769×10^308
```

Mathematica automatically converts any result larger than $MaxMachineNumber to arbitrary precision.

```
In[15]:= 2 $MaxMachineNumber
Out[15]= 3.595386269724631×10^308
```

Here is another computation whose result is outside the range of machine-precision numbers.

```
In[16]:= Exp[1000.]
Out[16]= 1.970071114017×10^434
```

Interval Arithmetic

`Interval[{`*min*`,`*max*`}]`	the interval from *min* to *max*
`Interval[{`min_1`,`max_1`},{`min_2`,`max_2`},…]`	the union of intervals from min_1 to max_1, min_2 to max_2, …

Representations of real intervals.

This represents all numbers between -2 and $+5$.

```
In[1]:= Interval[{-2, 5}]
Out[1]= Interval[{-2, 5}]
```

The square of any number between -2 and $+5$ is always between 0 and 25.

```
In[2]:= Interval[{-2, 5}]^2
Out[2]= Interval[{0, 25}]
```

Taking the reciprocal gives two distinct intervals.

```
In[3]:= 1 / Interval[{-2, 5}]
Out[3]= Interval[{{-∞, -1/2}, {1/5, ∞}}]
```

`Abs` folds the intervals back together again.

```
In[4]:= Abs[%]
Out[4]= Interval[{{1/5, ∞}}]
```

You can use intervals in many kinds of functions.

```
In[5]:= Solve[3 x + 2 == Interval[{-2, 5}], x]
Out[5]= {{x → Interval[{{-4/3, 1}}]}}
```

Some functions automatically generate intervals.

```
In[6]:= Limit[Sin[1 / x], x -> 0]
Out[6]= Interval[{-1, 1}]
```

IntervalUnion[$interval_1$, $interval_2$, ...]	find the union of several intervals
IntervalIntersection[$interval_1$, $interval_2$, ...]	find the intersection of several intervals
IntervalMemberQ[$interval$, x]	test whether the point x lies within an interval
IntervalMemberQ[$interval_1$, $interval_2$]	test whether $interval_2$ lies completely within $interval_1$

Operations on intervals.

This finds the overlap of the two intervals.

```
In[7]:= IntervalIntersection[Interval[{3, 7}], Interval[{-2, 5}]]
Out[7]= Interval[{3, 5}]
```

You can use Max and Min to find the end points of intervals.

```
In[8]:= Max[%]
Out[8]= 5
```

This finds out which of a list of intervals contains the point 7.

```
In[9]:= IntervalMemberQ[Table[Interval[{i, i + 1}], {i, 1, 20, 3}], 7]
Out[9]= {False, False, True, False, False, False, False}
```

You can use intervals not only with exact quantities but also with approximate numbers. Even with machine-precision numbers, *Mathematica* always tries to do rounding in such a way as to preserve the validity of results.

This shows explicitly the interval treated by *Mathematica* as the machine-precision number 0.

```
In[10]:= Interval[0.]
Out[10]= Interval[{-2.22507×10^-308, 2.22507×10^-308}]
```

This shows the corresponding interval around `100.`, shifted back to zero.

```
In[11]:= Interval[100.] - 100
Out[11]= Interval[{-1.42109×10^-14, 1.42109×10^-14}]
```

The same kind of thing works with numbers of any precision.

```
In[12]:= Interval[N[Pi, 50]] - Pi
Out[12]= Interval[{-0.×10^-50, 0.×10^-50}]
```

With ordinary machine-precision arithmetic, this computation gives an incorrect result.

```
In[13]:= Sin[N[Pi]]
Out[13]= 1.22461×10^-16
```

The interval generated here, however, includes the correct value of 0.

```
In[14]:= Sin[Interval[N[Pi]]]
Out[14]= Interval[{-3.21629×10^-16, 5.6655×10^-16}]
```

Indeterminate and Infinite Results

If you type in an expression like `0 / 0`, *Mathematica* prints a message, and returns the result `Indeterminate`.

```
In[1]:= 0 / 0
```

Power::infy: Infinite expression $\frac{1}{0}$ encountered. ≫

∞::indet: Indeterminate expression 0 ComplexInfinity encountered. ≫

```
Out[1]= Indeterminate
```

An expression like `0 / 0` is an example of an *indeterminate numerical result*. If you type in `0 / 0`, there is no way for *Mathematica* to know what answer you want. If you got `0 / 0` by taking the limit of x/x as $x \to 0$, then you might want the answer `1`. On the other hand, if you got `0 / 0` instead as the limit of $2x/x$, then you probably want the answer `2`. The expression `0 / 0` on its own does not contain enough information to choose between these and other cases. As a result, its value must be considered indeterminate.

Whenever an indeterminate result is produced in an arithmetic computation, *Mathematica* prints a warning message, and then returns `Indeterminate` as the result of the computation. If

you ever try to use `Indeterminate` in an arithmetic computation, you always get the result `Indeterminate`. A single indeterminate expression effectively "poisons" any arithmetic computation. (The symbol `Indeterminate` plays a role in *Mathematica* similar to the "not a number" object in the IEEE Floating Point Standard.)

The usual laws of arithmetic simplification are suspended in the case of `Indeterminate`.

```
In[2]:= Indeterminate - Indeterminate
Out[2]= Indeterminate
```

`Indeterminate` "poisons" any arithmetic computation, and leads to an indeterminate result.

```
In[3]:= 2 Indeterminate - 7
Out[3]= Indeterminate
```

When you do arithmetic computations inside *Mathematica* programs, it is often important to be able to tell whether indeterminate results were generated in the computations. You can do this by using the function `Check` discussed in "Messages" to test whether any warning messages associated with indeterminate results were produced.

You can use `Check` inside a program to test whether warning messages are generated in a computation.

```
In[4]:= Check[(7 - 7) / (8 - 8), meaningless]

        Power::infy: Infinite expression 1/0 encountered. >>

        ∞::indet: Indeterminate expression 0 ComplexInfinity encountered. >>

Out[4]= meaningless
```

`Indeterminate`	an indeterminate numerical result
`Infinity`	a positive infinite quantity
`-Infinity`	a negative infinite quantity (`DirectedInfinity[-1]`)
`DirectedInfinity[r]`	an infinite quantity with complex direction *r*
`ComplexInfinity`	an infinite quantity with an undetermined direction
`DirectedInfinity[]`	equivalent to `ComplexInfinity`

Indeterminate and infinite quantities.

There are many situations where it is convenient to be able to do calculations with infinite quantities. The symbol `Infinity` in *Mathematica* represents a positive infinite quantity. You can

use it to specify such things as limits of sums and integrals. You can also do some arithmetic calculations with it.

Here is an integral with an infinite limit.

```
In[5]:= Integrate[1 / x^3, {x, 1, Infinity}]
```

$$\text{Out[5]}= \frac{1}{2}$$

Mathematica knows that $1/\infty = 0$.

```
In[6]:= 1 / Infinity
Out[6]= 0
```

If you try to find the difference between two infinite quantities, you get an indeterminate result.

```
In[7]:= Infinity - Infinity

        ∞::indet : Indeterminate expression -∞ + ∞ encountered. >>
Out[7]= Indeterminate
```

There are a number of subtle points that arise in handling infinite quantities. One of them concerns the "direction" of an infinite quantity. When you do an infinite integral, you typically think of performing the integration along a path in the complex plane that goes to infinity in some direction. In this case, it is important to distinguish different versions of infinity that correspond to different directions in the complex plane. $+\infty$ and $-\infty$ are two examples, but for some purposes one also needs $i\,\infty$ and so on.

In *Mathematica*, infinite quantities can have a "direction", specified by a complex number. When you type in the symbol `Infinity`, representing a positive infinite quantity, this is converted internally to the form `DirectedInfinity[1]`, which represents an infinite quantity in the $+1$ direction. Similarly, `-Infinity` becomes `DirectedInfinity[-1]`, and `I Infinity` becomes `DirectedInfinity[I]`. Although the `DirectedInfinity` form is always used internally, the standard output format for `DirectedInfinity[`*r*`]` is *r* `Infinity`.

`Infinity` is converted internally to `DirectedInfinity[1]`.

```
In[8]:= Infinity // FullForm
Out[8]//FullForm= DirectedInfinity[1]
```

Although the notion of a "directed infinity" is often useful, it is not always available. If you type in `1 / 0`, you get an infinite result, but there is no way to determine the "direction" of the infinity

Mathematica represents the result of `1 / 0` as `DirectedInfinity[]`. In standard output form, this undirected infinity is printed out as `ComplexInfinity`.

`1 / 0` gives an undirected form of infinity.

```
In[9]:= 1 / 0

        Power::infy: Infinite expression 1/0 encountered. >>

Out[9]= ComplexInfinity
```

Controlling Numerical Evaluation

`NHoldAll`	prevent any arguments of a function from being affected by `N`
`NHoldFirst`	prevent the first argument from being affected
`NHoldRest`	prevent all but the first argument from being affected

Attributes for controlling numerical evaluation.

Usually `N` goes inside functions and gets applied to each of their arguments.

```
In[1]:= N[f[2 / 3, Pi]]
Out[1]= f[0.666667, 3.14159]
```

This tells *Mathematica* not to apply `N` to the first argument of `f`.

```
In[2]:= SetAttributes[f, NHoldFirst]
```

Now the first argument of `f` is left in its exact form.

```
In[3]:= N[f[2 / 3, Pi]]

Out[3]= f[2/3, 3.14159]
```

Algebraic Calculations

Symbolic Computation

One of the important features of *Mathematica* is that it can do *symbolic*, as well as *numerical* calculations. This means that it can handle algebraic formulas as well as numbers.

Here is a typical numerical computation.

```
In[1]:= 3 + 62 - 1
Out[1]= 64
```

This is a symbolic computation.

```
In[2]:= 3 x - x + 2
Out[2]= 2 + 2 x
```

Numerical computation	$62 + 3 - 1 \longrightarrow 64$
Symbolic computation	$3x + 2x - x + 2 \longrightarrow 2$

Numerical and symbolic computations.

You can type any algebraic expression into *Mathematica*.

```
In[3]:= -1 + 2 x + x^3
Out[3]= -1 + 2 x + x^3
```

Mathematica automatically carries out basic algebraic simplifications. Here it combines x^2 and $-4x^2$ to get $-3x^2$.

```
In[4]:= x^2 + x - 4 x^2
Out[4]= x - 3 x^2
```

You can type in any algebraic expression, using the operators listed in "Arithmetic". You can use spaces to denote multiplication. Be careful not to forget the space in `x y`. If you type in `xy` with no space, *Mathematica* will interpret this as a single symbol, with the name `xy`, not as a product of the two symbols `x` and `y`.

Mathematica rearranges and combines terms using the standard rules of algebra.

```
In[5]:= x y + 2 x^2 y + y^2 x^2 - 2 y x
Out[5]= -x y + 2 x^2 y + x^2 y^2
```

Here is another algebraic expression.

```
In[6]:= (x + 2 y + 1) (x - 2)^2
Out[6]= (-2 + x)^2 (1 + x + 2 y)
```

The function `Expand` multiplies out products and powers.

```
In[7]:= Expand[%]
Out[7]= 4 - 3 x^2 + x^3 + 8 y - 8 x y + 2 x^2 y
```

`Factor` does essentially the inverse of `Expand`.

```
In[8]:= Factor[%]
Out[8]= (-2 + x)^2 (1 + x + 2 y)
```

When you type in more complicated expressions, it is important that you put parentheses in the right places. Thus, for example, you have to give the expression x^{4y} in the form `x^(4 y)`. If you leave out the parentheses, you get $x^4 y$ instead. It never hurts to put in too many parentheses, but to find out exactly when you need to use parentheses, look at "Operator Input Forms".

Here is a more complicated formula, requiring several parentheses.

```
In[9]:= Sqrt[2] / 9801 (4 n)! (1103 + 26 390 n) / (n!^4 396^(4 n))
Out[9]= 2^(1/2 - 8 n) 99^(-2-4 n) (1103 + 26 390 n) (4 n)! / (n!)^4
```

When you type in an expression, *Mathematica* automatically applies its large repertoire of rules for transforming expressions. These rules include the standard rules of algebra, such as $x - x = 0$, together with much more sophisticated rules involving higher mathematical functions.

Mathematica uses standard rules of algebra to replace $\left(\sqrt{1+x}\right)^4$ by $(1+x)^2$.

```
In[10]:= Sqrt[1 + x]^4
Out[10]= (1 + x)^2
```

Mathematica knows no rules for this expression, so it leaves the expression in the original form you gave.

```
In[11]:= Log[1 + Cos[x]]
Out[11]= Log[1 + Cos[x]]
```

The notion of transformation rules is a very general one. In fact, you can think of the whole of *Mathematica* as simply a system for applying a collection of transformation rules to many different kinds of expressions.

The general principle that *Mathematica* follows is simple to state. It takes any expression you input, and gets results by applying a succession of transformation rules, stopping when it knows no more transformation rules that can be applied.

- Take any expression, and apply transformation rules until the result no longer changes.

The fundamental principle of *Mathematica*.

Values for Symbols

When *Mathematica* transforms an expression such as `x + x` into `2 x`, it is treating the variable `x` in a purely symbolic or formal fashion. In such cases, `x` is a symbol which can stand for any expression.

Often, however, you need to replace a symbol like `x` with a definite "value". Sometimes this value will be a number; often it will be another expression.

To take an expression such as `1 + 2 x` and replace the symbol `x` that appears in it with a definite value, you can create a *Mathematica* transformation rule, and then apply this rule to the expression. To replace `x` with the value `3`, you would create the transformation rule `x -> 3`. You must type `->` as a pair of characters, with no space in between. You can think of `x -> 3` as being a rule in which "`x` goes to `3`".

To apply a transformation rule to a particular *Mathematica* expression, you type *expr* `/.` *rule*. The "replacement operator" `/.` is typed as a pair of characters, with no space in between.

This uses the transformation rule `x -> 3` in the expression `1 + 2 x`.

```
In[1]:= 1 + 2 x /. x -> 3
Out[1]= 7
```

You can replace x with any expression. Here every occurrence of x is replaced by 2 – y.

```
In[2]:= 1 + x + x^2 /. x -> 2 - y
Out[2]= 3 + (2 - y)^2 - y
```

Here is a transformation rule. *Mathematica* treats it like any other symbolic expression.

```
In[3]:= x -> 3 + y
Out[3]= x → 3 + y
```

This applies the transformation rule on the previous line to the expression x^2 – 9.

```
In[4]:= x^2 - 9 /. %
Out[4]= -9 + (3 + y)^2
```

expr/.*x*->*value*	replace *x* by *value* in the expression *expr*
expr/.{*x*->*xval*,*y*->*yval*}	perform several replacements

Replacing symbols by values in expressions.

You can apply rules together by putting the rules in a list.

```
In[5]:= (x + y) (x - y)^2 /. {x -> 3, y -> 1 - a}
Out[5]= (4 - a) (2 + a)^2
```

The replacement operator /. allows you to apply transformation rules to a particular expression. Sometimes, however, you will want to define transformation rules that should *always* be applied. For example, you might want to replace x with 3 whenever x occurs.

As discussed in "Defining Variables", you can do this by *assigning* the value 3 to x using x = 3. Once you have made the assignment x = 3, x will always be replaced by 3, whenever it appears.

This assigns the value 3 to x.

```
In[6]:= x = 3
Out[6]= 3
```

Now x will automatically be replaced by 3 wherever it appears.

```
In[7]:= x^2 - 1
Out[7]= 8
```

This assigns the expression 1 + a to be the value of x.

```
In[8]:= x = 1 + a
Out[8]= 1 + a
```

Now x is replaced by 1 + a.

```
In[9]:= x^2 - 1
Out[9]= -1 + (1 + a)^2
```

You can define the value of a symbol to be any expression, not just a number. You should realize that once you have given such a definition, the definition will continue to be used whenever the symbol appears, until you explicitly change or remove the definition. For most people, forgetting to remove values you have assigned to symbols is the single most common source of mistakes in using *Mathematica*.

x=*value*	define a value for *x* which will always be used
x=.	remove any value defined for *x*

Assigning values to symbols.

The symbol x still has the value you assigned to it.

```
In[10]:= x + 5 - 2 x
Out[10]= 6 + a - 2 (1 + a)
```

This removes the value you assigned to x.

```
In[11]:= x =.
```

Now x has no value defined, so it can be used as a purely symbolic variable.

```
In[12]:= x + 5 - 2 x
Out[12]= 5 - x
```

A symbol such as x can serve many different purposes in *Mathematica*, and in fact, much of the flexibility of *Mathematica* comes from being able to mix these purposes at will. However, you need to keep some of the different uses of x straight in order to avoid making mistakes. The most important distinction is between the use of x as a name for another expression, and as a symbolic variable that stands only for itself.

Traditional programming languages that do not support symbolic computation allow variables to be used only as names for objects, typically numbers, that have been assigned as values for them. In *Mathematica*, however, `x` can also be treated as a purely formal variable, to which various transformation rules can be applied. Of course, if you explicitly give a definition, such as `x = 3`, then `x` will always be replaced by `3`, and can no longer serve as a formal variable.

You should understand that explicit definitions such as `x = 3` have a global effect. On the other hand, a replacement such as *expr* `/. x -> 3` affects only the specific expression *expr*. It is usually much easier to keep things straight if you avoid using explicit definitions except when absolutely necessary.

You can always mix replacements with assignments. With assignments, you can give names to expressions in which you want to do replacements, or to rules that you want to use to do the replacements.

This assigns a value to the symbol `t`.

```
In[13]:= t = 1 + x^2
Out[13]= 1 + x^2
```

This finds the value of `t`, and then replaces `x` by `2` in it.

```
In[14]:= t /. x -> 2
Out[14]= 5
```

This finds the value of `t` for a different value of `x`.

```
In[15]:= t /. x -> 5 a
Out[15]= 1 + 25 a^2
```

This finds the value of `t` when `x` is replaced by `Pi`, and then evaluates the result numerically.

```
In[16]:= t /. x -> Pi // N
Out[16]= 10.8696
```

Transforming Algebraic Expressions

There are often many different ways to write the same algebraic expression. As one example, the expression $(1+x)^2$ can be written as $1+2x+x^2$. *Mathematica* provides a large collection of functions for converting between different forms of algebraic expressions.

`Expand[`*expr*`]`	multiply out products and powers, writing the result as a sum of terms
`Factor[`*expr*`]`	write *expr* as a product of minimal factors

Two common functions for transforming algebraic expressions.

`Expand` gives the "expanded form", with products and powers multiplied out.

```
In[1]:= Expand[(1 + x)^2]
Out[1]= 1 + 2 x + x^2
```

`Factor` recovers the original form.

```
In[2]:= Factor[%]
Out[2]= (1 + x)^2
```

It is easy to generate complicated expressions with `Expand`.

```
In[3]:= Expand[(1 + x + 3 y)^4]
Out[3]= 1 + 4 x + 6 x^2 + 4 x^3 + x^4 + 12 y + 36 x y + 36 x^2 y + 12 x^3 y + 54 y^2 + 108 x y^2 + 54 x^2 y^2 + 108 y^3 + 108 x y^3 + 81 y^4
```

`Factor` often gives you simpler expressions.

```
In[4]:= Factor[%]
Out[4]= (1 + x + 3 y)^4
```

There are some cases, though, where `Factor` can give you more complicated expressions.

```
In[5]:= Factor[x^10 - 1]
Out[5]= (-1 + x) (1 + x) (1 - x + x^2 - x^3 + x^4) (1 + x + x^2 + x^3 + x^4)
```

In this case, `Expand` gives the "simpler" form.

```
In[6]:= Expand[%]
Out[6]= -1 + x^10
```

Simplifying Algebraic Expressions

There are many situations where you want to write a particular algebraic expression in the simplest possible form. Although it is difficult to know exactly what one means in all cases by

the "simplest form", a worthwhile practical procedure is to look at many different forms of an expression, and pick out the one that involves the smallest number of parts.

`Simplify[`*expr*`]`	try to find the simplest form of *expr* by applying various standard algebraic transformations
`FullSimplify[`*expr*`]`	try to find the simplest form by applying a wide range of transformations

Simplifying algebraic expressions.

`Simplify` writes $x^2 + 2x + 1$ in factored form.

```
In[1]:= Simplify[x^2 + 2 x + 1]
Out[1]= (1 + x)^2
```

`Simplify` leaves $x^{10} - 1$ in expanded form, since for this expression, the factored form is larger.

```
In[2]:= Simplify[x^10 - 1]
Out[2]= -1 + x^10
```

You can often use `Simplify` to "clean up" complicated expressions that you get as the results of computations.

Here is the integral of $1/(x^4 - 1)$. Integrals are discussed in more detail in "Integration".

```
In[3]:= Integrate[1 / (x^4 - 1), x]
Out[3]= -ArcTan[x]/2 + 1/4 Log[-1 + x] - 1/4 Log[1 + x]
```

Differentiating the result from `Integrate` should give back your original expression. In this case, as is common, you get a more complicated version of the expression.

```
In[4]:= D[%, x]
Out[4]= 1/(4 (-1 + x)) - 1/(4 (1 + x)) - 1/(2 (1 + x^2))
```

`Simplify` succeeds in getting back the original, simpler, form of the expression.

```
In[5]:= Simplify[%]
Out[5]= 1/(-1 + x^4)
```

Simplify is set up to try various standard algebraic transformations on the expressions you give. Sometimes, however, it can take more sophisticated transformations to make progress in finding the simplest form of an expression.

FullSimplify tries a much wider range of transformations, involving not only algebraic functions, but also many other kinds of functions.

Simplify does nothing to this expression.

```
In[6]:= Simplify[Gamma[x] Gamma[1 - x]]
Out[6]= Gamma[1 - x] Gamma[x]
```

FullSimplify, however, transforms it to a simpler form.

```
In[7]:= FullSimplify[Gamma[x] Gamma[1 - x]]
Out[7]= π Csc[π x]
```

For fairly small expressions, FullSimplify will often succeed in making some remarkable simplifications. But for larger expressions, it can become unmanageably slow.

The reason for this is that to do its job, FullSimplify effectively has to try combining every part of an expression with every other, and for large expressions the number of cases that it has to consider can be astronomically large.

Simplify also has a difficult task to do, but it is set up to avoid some of the most time-consuming transformations that are tried by FullSimplify. For simple algebraic calculations, therefore, you may often find it convenient to apply Simplify quite routinely to your results.

In more complicated calculations, however, even Simplify, let alone FullSimplify, may end up needing to try a very large number of different forms, and therefore taking a long time. In such cases, you typically need to do more controlled simplification, and use your knowledge of the form you want to get to guide the process.

Putting Expressions into Different Forms

Complicated algebraic expressions can usually be written in many different ways. *Mathematica* provides a variety of functions for converting expressions from one form to another.

In many applications, the most common of these functions are Expand, Factor and Simplify.

However, particularly when you have rational expressions that contain quotients, you may need to use other functions.

`Expand[`*expr*`]`	multiply out products and powers
`ExpandAll[`*expr*`]`	apply `Expand` everywhere
`Factor[`*expr*`]`	reduce to a product of factors
`Together[`*expr*`]`	put all terms over a common denominator
`Apart[`*expr*`]`	separate into terms with simple denominators
`Cancel[`*expr*`]`	cancel common factors between numerators and denominators
`Simplify[`*expr*`]`	try a sequence of algebraic transformations and give the smallest form of *expr* found

Functions for transforming algebraic expressions.

Here is a rational expression that can be written in many different forms.

```
In[1]:= e = (x - 1)^2 (2 + x) / ((1 + x) (x - 3)^2)
```

$$\text{Out[1]}= \frac{(-1+x)^2\,(2+x)}{(-3+x)^2\,(1+x)}$$

`Expand` expands out the numerator, but leaves the denominator in factored form.

```
In[2]:= Expand[e]
```

$$\text{Out[2]}= \frac{2}{(-3+x)^2\,(1+x)} - \frac{3x}{(-3+x)^2\,(1+x)} + \frac{x^3}{(-3+x)^2\,(1+x)}$$

`ExpandAll` expands out everything, including the denominator.

```
In[3]:= ExpandAll[e]
```

$$\text{Out[3]}= \frac{2}{9+3x-5x^2+x^3} - \frac{3x}{9+3x-5x^2+x^3} + \frac{x^3}{9+3x-5x^2+x^3}$$

`Together` collects all the terms together over a common denominator.

```
In[4]:= Together[%]
```

$$\text{Out[4]}= \frac{2-3x+x^3}{(-3+x)^2\,(1+x)}$$

`Apart` breaks the expression apart into terms with simple denominators.

```
In[5]:= Apart[%]
```

$$\text{Out[5]}= 1 + \frac{5}{(-3+x)^2} + \frac{19}{4\,(-3+x)} + \frac{1}{4\,(1+x)}$$

`Factor` factors everything, in this case reproducing the original form.

```
In[6]:= Factor[%]
```

$$\text{Out[6]}= \frac{(-1+x)^2\,(2+x)}{(-3+x)^2\,(1+x)}$$

According to `Simplify`, this is the simplest way to write the original expression.

```
In[7]:= Simplify[e]
```

$$\text{Out[7]}= \frac{(-1+x)^2\,(2+x)}{(-3+x)^2\,(1+x)}$$

Getting expressions into the form you want is something of an art. In most cases, it is best simply to experiment, trying different transformations until you get what you want. Often you will be able to use palettes in the front end to do this.

When you have an expression with a single variable, you can choose to write it as a sum of terms, a product, and so on. If you have an expression with several variables, there is an even wider selection of possible forms. You can, for example, choose to group terms in the expression so that one or another of the variables is "dominant".

`Collect[`*expr*`,`*x*`]`	group together powers of *x*
`FactorTerms[`*expr*`,`*x*`]`	pull out factors that do not depend on *x*

Rearranging expressions in several variables.

Here is an algebraic expression in two variables.

```
In[8]:= v = Expand[(3 + 2 x)^2 (x + 2 y)^2]
```

$$\text{Out[8]}= 9x^2 + 12x^3 + 4x^4 + 36xy + 48x^2y + 16x^3y + 36y^2 + 48xy^2 + 16x^2y^2$$

This groups together terms in `v` that involve the same power of `x`.

```
In[9]:= Collect[v, x]
```

$$\text{Out[9]}= 4x^4 + 36y^2 + x^3\,(12 + 16y) + x^2\left(9 + 48y + 16y^2\right) + x\left(36y + 48y^2\right)$$

This groups together powers of y.

```
In[10]:= Collect[v, y]
Out[10]= 9 x^2 + 12 x^3 + 4 x^4 + (36 x + 48 x^2 + 16 x^3) y + (36 + 48 x + 16 x^2) y^2
```

This factors out the piece that does not depend on y.

```
In[11]:= FactorTerms[v, y]
Out[11]= (9 + 12 x + 4 x^2) (x^2 + 4 x y + 4 y^2)
```

As we have seen, even when you restrict yourself to polynomials and rational expressions, there are many different ways to write any particular expression. If you consider more complicated expressions, involving, for example, higher mathematical functions, the variety of possible forms becomes still greater. As a result, it is totally infeasible to have a specific function built into *Mathematica* to produce each possible form. Rather, *Mathematica* allows you to construct arbitrary sets of transformation rules for converting between different forms. Many *Mathematica* packages include such rules; the details of how to construct them for yourself are given in "Transformation Rules and Definitions".

There are nevertheless a few additional built-in *Mathematica* functions for transforming expressions.

`TrigExpand[`*expr*`]`	expand out trigonometric expressions into a sum of terms
`TrigFactor[`*expr*`]`	factor trigonometric expressions into products of terms
`TrigReduce[`*expr*`]`	reduce trigonometric expressions using multiple angles
`TrigToExp[`*expr*`]`	convert trigonometric functions to exponentials
`ExpToTrig[`*expr*`]`	convert exponentials to trigonometric functions
`FunctionExpand[`*expr*`]`	expand out special and other functions
`ComplexExpand[`*expr*`]`	perform expansions assuming that all variables are real
`PowerExpand[`*expr*`]`	transform $(x\,y)^{\,p}$ into $x^p\,y^p$, etc.

Some other functions for transforming expressions.

This expands out the trigonometric expression, writing it so that all functions have argument x.

```
In[12]:= TrigExpand[Tan[x] Cos[2 x]]
Out[12]= 3/2 Cos[x] Sin[x] - Tan[x]/2 - 1/2 Sin[x]^2 Tan[x]
```

This uses trigonometric identities to generate a factored form of the expression.

```
In[13]:= TrigFactor[%]
```

$$\text{Out[13]= } 2\,\text{Sin}\left[\frac{\pi}{4}-x\right]\,\text{Sin}\left[\frac{\pi}{4}+x\right]\,\text{Tan}[x]$$

This reduces the expression by using multiple angles.

```
In[14]:= TrigReduce[%]
```

$$\text{Out[14]= } -\frac{1}{2}\,\text{Sec}[x]\,(\text{Sin}[x]-\text{Sin}[3\,x])$$

This expands the sine assuming that `x` and `y` are both real.

```
In[15]:= ComplexExpand[Sin[x + I y]]
Out[15]= Cosh[y] Sin[x] + i Cos[x] Sinh[y]
```

This does the expansion allowing `x` and `y` to be complex.

```
In[16]:= ComplexExpand[Sin[x + I y], {x, y}]
Out[16]= -Cosh[Im[x] + Re[y]] Sin[Im[y] - Re[x]] + i Cos[Im[y] - Re[x]] Sinh[Im[x] + Re[y]]
```

The transformations on expressions done by functions like `Expand` and `Factor` are always correct, whatever values the symbolic variables in the expressions may have. Sometimes, however, it is useful to perform transformations that are only correct for some possible values of symbolic variables. One such transformation is performed by `PowerExpand`.

Mathematica does not automatically expand out non-integer powers of products.

```
In[17]:= Sqrt[x y]
```

$$\text{Out[17]= } \sqrt{x\,y}$$

`PowerExpand` does the expansion.

```
In[18]:= PowerExpand[%]
```

$$\text{Out[18]= } \sqrt{x}\,\sqrt{y}$$

Simplifying with Assumptions

`Simplify[`*expr*`,`*assum*`]`	simplify *expr* with assumptions

Simplifying with assumptions.

Mathematica does not automatically simplify this, since it is only true for some values of `x`.

```
In[1]:= Simplify[Sqrt[x^2]]
```

$$\text{Out[1]}= \sqrt{x^2}$$

$\sqrt{x^2}$ is equal to x for $x \geq 0$, but not otherwise.

```
In[2]:= {Sqrt[4^2], Sqrt[(-4)^2]}
Out[2]= {4, 4}
```

This tells `Simplify` to make the assumption `x > 0`, so that simplification can proceed.

```
In[3]:= Simplify[Sqrt[x^2], x > 0]
Out[3]= x
```

No automatic simplification can be done on this expression.

```
In[4]:= 2 a + 2 Sqrt[a - Sqrt[-b]] Sqrt[a + Sqrt[-b]]
```

$$\text{Out[4]}= 2\,a + 2\sqrt{a - \sqrt{-b}}\ \sqrt{a + \sqrt{-b}}$$

If a and b are assumed to be positive, the expression can however be simplified.

```
In[5]:= Simplify[%, a > 0 && b > 0]
```

$$\text{Out[5]}= 2\left(a + \sqrt{a^2 + b}\right)$$

Here is a simple example involving trigonometric functions.

```
In[6]:= Simplify[ArcSin[Sin[x]], -Pi / 2 < x < Pi / 2]
Out[6]= x
```

`Element[x, dom]`	state that x is an element of the domain dom
`Element[{x1, x2, ...}, dom]`	state that all the x_i are elements of the domain dom
`Reals`	real numbers
`Integers`	integers
`Primes`	prime numbers

Some domains used in assumptions.

This simplifies $\sqrt{x^2}$ assuming that x is a real number.

```
In[7]:= Simplify[Sqrt[x^2], Element[x, Reals]]
Out[7]= Abs[x]
```

This simplifies the sine assuming that n is an integer.

```
In[8]:= Simplify[Sin[x + 2 n Pi], Element[n, Integers]]
Out[8]= Sin[x]
```

With the assumptions given, Fermat's little theorem can be used.

```
In[9]:= Simplify[Mod[a^p, p], Element[a, Integers] && Element[p, Primes]]
Out[9]= Mod[a, p]
```

This uses the fact that $\sin(x)$, but not $\arcsin(x)$, is real when x is real.

```
In[10]:= Simplify[Re[{Sin[x], ArcSin[x]}], Element[x, Reals]]
Out[10]= {Sin[x], Re[ArcSin[x]]}
```

Picking Out Pieces of Algebraic Expressions

`Coefficient[expr, form]`	coefficient of $form$ in $expr$
`Exponent[expr, form]`	maximum power of $form$ in $expr$
`Part[expr, n]` or `expr[[n]]`	n^{th} term of $expr$

Functions to pick out pieces of polynomials.

Here is an algebraic expression.

```
In[1]:= e = Expand[(1 + 3 x + 4 y^2)^2]
```

$$\text{Out[1]}= 1 + 6\,x + 9\,x^2 + 8\,y^2 + 24\,x\,y^2 + 16\,y^4$$

This gives the coefficient of x in e.

```
In[2]:= Coefficient[e, x]
```

$$\text{Out[2]}= 6 + 24\,y^2$$

`Exponent[`*expr*`, `*y*`]` gives the highest power of *y* that appears in *expr*.

```
In[3]:= Exponent[e, y]
```

$$\text{Out[3]}= 4$$

This gives the fourth term in e.

```
In[4]:= Part[e, 4]
```

$$\text{Out[4]}= 8\,y^2$$

You may notice that the function `Part[`*expr*`, `*n*`]` used to pick out the n^{th} term in a sum is the same as the function described in "Manipulating Elements of Lists" for picking out elements in lists. This is no coincidence. In fact, as discussed in "Manipulating Expressions like Lists," every *Mathematica* expression can be manipulated structurally much like a list. However, as discussed in "Manipulating Expressions like Lists," you must be careful, because *Mathematica* often shows algebraic expressions in a form that is different from the way it treats them internally.

`Coefficient` works even with polynomials that are not explicitly expanded out.

```
In[5]:= Coefficient[(1 + 3 x + 4 y^2)^2, x]
```

$$\text{Out[5]}= 6 + 24\,y^2$$

`Numerator[`*expr*`]`	numerator of *expr*
`Denominator[`*expr*`]`	denominator of *expr*

Functions to pick out pieces of rational expressions.

Here is a rational expression.

```
In[6]:= r = (1 + x) / (2 (2 - y))
```

$$\text{Out[6]}= \frac{1 + x}{2\,(2 - y)}$$

`Denominator` picks out the denominator.

```
In[7]:= Denominator[%]
Out[7]= 2 (2 - y)
```

`Denominator` gives `1` for expressions that are not explicit quotients.

```
In[8]:= Denominator[1 / x + 2 / y]
Out[8]= 1
```

Controlling the Display of Large Expressions

When you do symbolic calculations, it is quite easy to end up with extremely complicated expressions. Often, you will not even want to *see* the complete result of a computation.

If you end your input with a semicolon, *Mathematica* will do the computation you asked for, but will not display the result. You can nevertheless use `%` or `Out[n]` to refer to the result.

By default, the *Mathematica* front end will display any outputs which are excessively large in a shortened form inside an interface which allows you to refine the display of the output.

Mathematica shows this output with 5138 of the terms omitted.

```
In[1]:= Expand[(x + 2 y + 1)^100]
```

Out[1]=

A very large output was generated. Here is a sample of it:

$$1 + 100\,x + 4950\,x^2 + 161\,700\,x^3 + 3\,921\,225\,x^4 + 75\,287\,520\,x^5 + 1\,192\,052\,400\,x^6 + \text{<<5138>>} +$$
$$1\,568\,717\,617\,782\,433\,884\,352\,170\,216\,652\,800\,y^{98} + 3\,137\,435\,235\,564\,867\,768\,704\,340\,433\,305\,600\,x\,y^{98} +$$
$$1\,568\,717\,617\,782\,433\,884\,352\,170\,216\,652\,800\,x^2\,y^{98} + 63\,382\,530\,011\,411\,470\,074\,835\,160\,268\,800\,y^{99} +$$
$$63\,382\,530\,011\,411\,470\,074\,835\,160\,268\,800\,x\,y^{99} + 1\,267\,650\,600\,228\,229\,401\,496\,703\,205\,376\,y^{100}$$

Show Less | Show More | Show Full Output | Set Size Limit...

The **Show Less** and **Show More** buttons allow you to decrease or increase the level of detail to which *Mathematica* shows the expression. The **Show Full Output** button removes the interface entirely and displays the full result, but the result may take considerable time to display. The default threshold size at which this feature starts working may be set using the **Set Size Limit** option, which opens the **Preferences** dialog to the panel with the appropriate setting.

The large output suppression feature is implemented using the *Mathematica* function `Short`. You can use `Short` directly for finer control over the display of expressions. You can also use it for outputs which are not large enough to be suppressed by the default suppression scheme.

Ending your input with ; stops *Mathematica* from displaying the complicated result of the computation.

```
In[2]:= Expand[(x + 5 y + 10) ^ 8];
```

You can still refer to the result as %. // Short displays a one-line outline of the result. The << *n* >> stands for *n* terms that have been left out.

```
In[3]:= % // Short
```

$$\text{Out[3]//Short=}\quad 100\,000\,000 + 80\,000\,000\,x + 28\,000\,000\,x^2 + \text{<<39>>} + 6\,250\,000\,y^7 + 625\,000\,x\,y^7 + 390\,625\,y^8$$

This shows a three-line version of the expression. More parts are now visible.

```
In[4]:= Short[%, 3]
```

$$\begin{aligned}\text{Out[4]//Short=}\quad & 100\,000\,000 + 80\,000\,000\,x + 28\,000\,000\,x^2 + 5\,600\,000\,x^3 + 700\,000\,x^4 + \\ & 56\,000\,x^5 + 2800\,x^6 + 80\,x^7 + x^8 + \text{<<28>>} + 5\,250\,000\,x^2\,y^5 + 175\,000\,x^3\,y^5 + \\ & 43\,750\,000\,y^6 + 8\,750\,000\,x\,y^6 + 437\,500\,x^2\,y^6 + 6\,250\,000\,y^7 + 625\,000\,x\,y^7 + 390\,625\,y^8\end{aligned}$$

This gives the total number of terms in the sum.

```
In[5]:= Length[%]
Out[5]= 45
```

command;	execute *command*, but do not print the result
expr // Short	show a one-line outline form of *expr*
Short[*expr*, *n*]	show an *n*-line outline of *expr*

Some ways to shorten your output.

Using Symbols to Tag Objects

There are many ways to use symbols in *Mathematica*. Here we use symbols as "tags" for different types of objects.

Working with physical units gives one simple example. When you specify the length of an object, you want to give not only a number, but also the units in which the length is measured. In standard notation, you might write a length as 12 meters.

You can imitate this notation almost directly in *Mathematica*. You can for example simply use a symbol meters to indicate the units of your measurement.

The symbol `meters` here acts as a tag, which indicates the units used.

```
In[1]:= 12 meters
Out[1]= 12 meters
```

You can add lengths like this.

```
In[2]:= % + 5.3 meters
Out[2]= 17.3 meters
```

This gives a speed.

```
In[3]:= % / (25 seconds)
                0.692 meters
Out[3]= ------------------------
                  seconds
```

This converts to a speed in feet per second.

```
In[4]:= % /. meters -> 3.28084 feet
                2.27034 feet
Out[4]= ------------------------
                  seconds
```

There is in fact a *Mathematica* package that allows you to work with units. The package defines many symbols that represent standard types of units.

Load the *Mathematica* package for handling units.

```
In[5]:= << Units`
```

The package uses standardized names for units.

```
In[6]:= 12 Meter / Second
                12 Meter
Out[6]= ------------------------
                  Second
```

The function `Convert[`*expr*`, `*units*`]` converts to the specified units.

```
In[7]:= Convert[%, Mile / Hour]
```

$$\text{Out[7]}= \frac{37\,500\ \text{Mile}}{1397\ \text{Hour}}$$

Usually you have to give prefixes for units as separate words.

```
In[8]:= Convert[3 Kilo Meter / Hour, Inch / Minute]
```

$$\text{Out[8]}= \frac{250\,000\ \text{Inch}}{127\ \text{Minute}}$$

Algebraic Manipulation

Structural Operations on Polynomials

`Expand[`*poly*`]`	expand out products and powers
`Factor[`*poly*`]`	factor completely
`FactorTerms[`*poly*`]`	pull out any overall numerical factor
`FactorTerms[`*poly*`,{`*x*`,`*y*`,...}]`	pull out any overall factor that does not depend on *x*, *y*, ...
`Collect[`*poly*`,`*x*`]`	arrange a polynomial as a sum of powers of *x*
`Collect[`*poly*`,{`*x*`,`*y*`,...}]`	arrange a polynomial as a sum of powers of *x*, *y*, ...

Structural operations on polynomials.

Here is a polynomial in one variable.

```
In[1]:= (2 + 4 x^2)^2 (x - 1)^3
Out[1]= (-1 + x)^3 (2 + 4 x^2)^2
```

`Expand` expands out products and powers, writing the polynomial as a simple sum of terms.

```
In[2]:= t = Expand[%]
Out[2]= -4 + 12 x - 28 x^2 + 52 x^3 - 64 x^4 + 64 x^5 - 48 x^6 + 16 x^7
```

`Factor` performs complete factoring of the polynomial.

```
In[3]:= Factor[t]
Out[3]= 4 (-1 + x)^3 (1 + 2 x^2)^2
```

`FactorTerms` pulls out the overall numerical factor from `t`.

```
In[4]:= FactorTerms[t]
Out[4]= 4 (-1 + 3 x - 7 x^2 + 13 x^3 - 16 x^4 + 16 x^5 - 12 x^6 + 4 x^7)
```

There are several ways to write any polynomial. The functions `Expand`, `FactorTerms` and `Factor` give three common ways. `Expand` writes a polynomial as a simple sum of terms, with all products expanded out. `FactorTerms` pulls out common factors from each term. `Factor` does complete factoring, writing the polynomial as a product of terms, each of as low degree as possible.

When you have a polynomial in more than one variable, you can put the polynomial in different forms by essentially choosing different variables to be "dominant". `Collect[poly, x]` takes a polynomial in several variables and rewrites it as a sum of terms containing different powers of the "dominant variable" x.

Here is a polynomial in two variables.

```
In[5]:= Expand[(1 + 2 x + y)^3]
```

$$1 + 6\,x + 12\,x^2 + 8\,x^3 + 3\,y + 12\,x\,y + 12\,x^2\,y + 3\,y^2 + 6\,x\,y^2 + y^3$$

`Collect` reorganizes the polynomial so that `x` is the "dominant variable".

```
In[6]:= Collect[%, x]
```

$$1 + 8\,x^3 + 3\,y + 3\,y^2 + y^3 + x^2\,(12 + 12\,y) + x\,\left(6 + 12\,y + 6\,y^2\right)$$

If you specify a list of variables, `Collect` will effectively write the expression as a polynomial in these variables.

```
In[7]:= Collect[Expand[(1 + x + 2 y + 3 z)^3], {x, y}]
```

$$1 + x^3 + 8\,y^3 + 9\,z + 27\,z^2 + 27\,z^3 + x^2\,(3 + 6\,y + 9\,z) + y^2\,(12 + 36\,z) + y\,\left(6 + 36\,z + 54\,z^2\right) + x\,\left(3 + 12\,y^2 + 18\,z + 27\,z^2 + y\,(12 + 36\,z)\right)$$

`Expand[`*poly*`,`*patt*`]`	expand out *poly* avoiding those parts which do not contain terms matching *patt*

Controlling polynomial expansion.

This avoids expanding parts which do not contain `x`.

```
In[8]:= Expand[(x + 1)^2 (y + 1)^2, x]
```

$$(1 + y)^2 + 2\,x\,(1 + y)^2 + x^2\,(1 + y)^2$$

This avoids expanding parts which do not contain objects matching `b[_]`.

```
In[9]:= Expand[(a[1] + a[2] + 1)^2 (1 + b[1])^2, b[_]]
```

$$(1 + a[1] + a[2])^2 + 2\,(1 + a[1] + a[2])^2\,b[1] + (1 + a[1] + a[2])^2\,b[1]^2$$

`PowerExpand[`*expr*`]`	expand out $(a\,b)^c$ and $\left(a^b\right)^c$ in *expr*
`PowerExpand[`*expr*`,Assumptions->`*assum*`]`	expand out *expr* assuming *assum*

Expanding powers and logarithms.

Mathematica does not automatically expand out expressions of the form (a b)^c except when c is an integer. In general it is only correct to do this expansion if a and b are positive reals. Nevertheless, the function PowerExpand does the expansion, effectively assuming that a and b are indeed positive reals.

Mathematica does not automatically expand out this expression.

```
In[10]:= (x y) ^ n
Out[10]= (x y)^n
```

PowerExpand does the expansion, effectively assuming that x and y are positive reals.

```
In[11]:= PowerExpand[%]
Out[11]= x^n y^n
```

Log is not automatically expanded out.

```
In[12]:= Log[%]
Out[12]= Log[x^n y^n]
```

PowerExpand does the expansion.

```
In[13]:= PowerExpand[%]
Out[13]= n Log[x] + n Log[y]
```

PowerExpand returns a result correct for the given assumptions.

```
In[14]:= PowerExpand[%%, Assumptions → {x > 0, n < 0}]
Out[14]= 2 i π Floor[1/2 - Im[n Log[x]]/(2 π) - Im[n Log[y]]/(2 π)] + n Log[x] + n Log[y]
```

Collect[*poly*, *patt*]	collect separately terms involving each object that matches *patt*
Collect[*poly*, *patt*, *h*]	apply *h* to each final coefficient obtained

Ways of collecting terms.

Here is an expression involving various functions f.

```
In[15]:= t = 3 + x f[1] + x^2 f[1] + y f[2]^2 + z f[2]^2
Out[15]= 3 + x f[1] + x^2 f[1] + y f[2]^2 + z f[2]^2
```

This collects terms that match `f[_]`.

```
In[16]:= Collect[t, f[_]]
Out[16]= 3 + (x + x^2) f[1] + (y + z) f[2]^2
```

This applies `Factor` to each coefficient obtained.

```
In[17]:= Collect[t, f[_], Factor]
Out[17]= 3 + x (1 + x) f[1] + (y + z) f[2]^2
```

`HornerForm[`*expr*`,`*x*`]`	puts *expr* into Horner form with respect to *x*

Horner form.

Horner form is a way of arranging a polynomial that allows numerical values to be computed more efficiently by minimizing the number of multiplications.

This gives the Horner form of a polynomial.

```
In[18]:= HornerForm[1 + 2 x + 3 x^2 + 4 x^3, x]
Out[18]= 1 + x (2 + x (3 + 4 x))
```

Finding the Structure of a Polynomial

`PolynomialQ[`*expr*`,`*x*`]`	test whether *expr* is a polynomial in *x*
`PolynomialQ[`*expr*`,{`x_1`,`x_2`,...}]`	test whether *expr* is a polynomial in the x_i
`Variables[`*poly*`]`	a list of the variables in *poly*
`Exponent[`*poly*`,`*x*`]`	the maximum exponent with which *x* appears in *poly*
`Coefficient[`*poly*`,`*expr*`]`	the coefficient of *expr* in *poly*
`Coefficient[`*poly*`,`*expr*`,`*n*`]`	the coefficient of $expr^n$ in *poly*
`Coefficient[`*poly*`,`*expr*`,0]`	the term in *poly* independent of *expr*
`CoefficientList[`*poly*`,{`x_1`,`x_2`,...}]`	generate an array of the coefficients of the x_i in *poly*
`CoefficientRules[`*poly*`,{`x_1`,`x_2`,...}]`	get exponent vectors and coefficients of monomials

Finding the structure of polynomials written in expanded form.

Here is a polynomial in two variables.

```
In[1]:= t = (1 + x)^3 (1 - y - x)^2
Out[1]= (1 + x)^3 (1 - x - y)^2
```

This is the polynomial in expanded form.

```
In[2]:= Expand[t]
```

$$\text{Out[2]}= 1 + x - 2\,x^2 - 2\,x^3 + x^4 + x^5 - 2\,y - 4\,x\,y + 4\,x^3\,y + 2\,x^4\,y + y^2 + 3\,x\,y^2 + 3\,x^2\,y^2 + x^3\,y^2$$

`PolynomialQ` reports that `t` is a polynomial in `x`.

```
In[3]:= PolynomialQ[t, x]
Out[3]= True
```

This expression, however, is not a polynomial in `x`.

```
In[4]:= PolynomialQ[x + Sin[x], x]
Out[4]= False
```

`Variables` gives a list of the variables in the polynomial `t`.

```
In[5]:= Variables[t]
Out[5]= {x, y}
```

This gives the maximum exponent with which `x` appears in the polynomial `t`. For a polynomial in one variable, `Exponent` gives the degree of the polynomial.

```
In[6]:= Exponent[t, x]
Out[6]= 5
```

`Coefficient[`*poly*`,` *expr*`]` gives the total coefficient with which *expr* appears in *poly*. In this case, the result is a sum of two terms.

```
In[7]:= Coefficient[t, x^2]
```

$$\text{Out[7]}= -2 + 3\,y^2$$

This is equivalent to `Coefficient[t, x^2]`.

```
In[8]:= Coefficient[t, x, 2]
```

$$\text{Out[8]}= -2 + 3\,y^2$$

This picks out the coefficient of x^0 in `t`.

```
In[9]:= Coefficient[t, x, 0]
```

$$\text{Out[9]}= 1 - 2\,y + y^2$$

`CoefficientList` gives a list of the coefficients of each power of x, starting with x^0.

```
In[10]:= CoefficientList[1 + 3 x^2 + 4 x^4, x]
Out[10]= {1, 0, 3, 0, 4}
```

For multivariate polynomials, `CoefficientList` gives an array of the coefficients for each power of each variable.

```
In[11]:= CoefficientList[t, {x, y}]
Out[11]= {{1, -2, 1}, {1, -4, 3}, {-2, 0, 3}, {-2, 4, 1}, {1, 2, 0}, {1, 0, 0}}
```

`CoefficientRules` includes only those monomials that have nonzero coefficients.

```
In[12]:= CoefficientRules[t, {x, y}]
Out[12]= {{5, 0} → 1, {4, 1} → 2, {4, 0} → 1, {3, 2} → 1, {3, 1} → 4, {3, 0} → -2, {2, 2} → 3,
  {2, 0} → -2, {1, 2} → 3, {1, 1} → -4, {1, 0} → 1, {0, 2} → 1, {0, 1} → -2, {0, 0} → 1}
```

It is important to notice that the functions in this tutorial will often work even on polynomials that are not explicitly given in expanded form.

Many of the functions also work on expressions that are not strictly polynomials.

Without giving specific integer values to `a`, `b` and `c`, this expression cannot strictly be considered a polynomial.

```
In[13]:= x^a + x^b + y^c
Out[13]= x^a + x^b + y^c
```

`Exponent[`*expr*`, x]` still gives the maximum exponent of `x` in *expr*, but here has to write the result in symbolic form.

```
In[14]:= Exponent[%, x]
Out[14]= Max[0, a, b]
```

Structural Operations on Rational Expressions

For ordinary polynomials, `Factor` and `Expand` give the most important forms. For rational expressions, there are many different forms that can be useful.

`ExpandNumerator[`*expr*`]`	expand numerators only
`ExpandDenominator[`*expr*`]`	expand denominators only
`Expand[`*expr*`]`	expand numerators, dividing the denominator into each term
`ExpandAll[`*expr*`]`	expand numerators and denominators completely

Different kinds of expansion for rational expressions.

Here is a rational expression.

```
In[1]:= t = (1 + x) ^ 2 / (1 - x) + 3 x ^ 2 / (1 + x) ^ 2 + (2 - x) ^ 2
```

$$\text{Out[1]}= (2 - x)^2 + \frac{3 x^2}{(1 + x)^2} + \frac{(1 + x)^2}{1 - x}$$

`ExpandNumerator` writes the numerator of each term in expanded form.

```
In[2]:= ExpandNumerator[t]
```

$$\text{Out[2]}= 4 - 4 x + x^2 + \frac{3 x^2}{(1 + x)^2} + \frac{1 + 2 x + x^2}{1 - x}$$

`Expand` expands the numerator of each term, and divides all the terms by the appropriate denominators.

```
In[3]:= Expand[t]
```

$$\text{Out[3]}= 4 + \frac{1}{1 - x} - 4 x + \frac{2 x}{1 - x} + x^2 + \frac{x^2}{1 - x} + \frac{3 x^2}{(1 + x)^2}$$

`ExpandDenominator` expands out the denominator of each term.

```
In[4]:= ExpandDenominator[t]
```

$$\text{Out[4]}= (2 - x)^2 + \frac{(1 + x)^2}{1 - x} + \frac{3 x^2}{1 + 2 x + x^2}$$

`ExpandAll` does all possible expansions in the numerator and denominator of each term.

```
In[5]:= ExpandAll[t]
```

$$\text{Out[5]}= 4 + \frac{1}{1 - x} - 4 x + \frac{2 x}{1 - x} + x^2 + \frac{x^2}{1 - x} + \frac{3 x^2}{1 + 2 x + x^2}$$

`ExpandAll[`*expr*`,`*patt*`]`, etc.	avoid expanding parts which contain no terms matching *patt*

Controlling expansion.

This avoids expanding the term which does not contain z.

```
In[6]:= ExpandAll[(x + 1) ^ 2 / y ^ 2 + (z + 1) ^ 2 / z ^ 2, z]
```

$$\text{Out[6]}= 1 + \frac{(1+x)^2}{y^2} + \frac{1}{z^2} + \frac{2}{z}$$

`Together[`*expr*`]`	combine all terms over a common denominator
`Apart[`*expr*`]`	write an expression as a sum of terms with simple denominators
`Cancel[`*expr*`]`	cancel common factors between numerators and denominators
`Factor[`*expr*`]`	perform a complete factoring

Structural operations on rational expressions.

Here is a rational expression.

```
In[7]:= u = (-4 x + x ^ 2) / (-x + x ^ 2) + (-4 + 3 x + x ^ 2) / (-1 + x ^ 2)
```

$$\text{Out[7]}= \frac{-4x + x^2}{-x + x^2} + \frac{-4 + 3x + x^2}{-1 + x^2}$$

`Together` puts all terms over a common denominator.

```
In[8]:= Together[u]
```

$$\text{Out[8]}= \frac{2\left(-4 + x^2\right)}{(-1 + x)(1 + x)}$$

You can use `Factor` to factor the numerator and denominator of the resulting expression.

```
In[9]:= Factor[%]
```

$$\text{Out[9]}= \frac{2(-2 + x)(2 + x)}{(-1 + x)(1 + x)}$$

`Apart` writes the expression as a sum of terms, with each term having as simple a denominator as possible.

```
In[10]:= Apart[u]
```

$$\text{Out[10]}= 2 - \frac{3}{-1 + x} + \frac{3}{1 + x}$$

`Cancel` cancels any common factors between numerators and denominators.

```
In[11]:= Cancel[u]
```

$$\text{Out[11]}= \frac{-4+x}{-1+x}+\frac{4+x}{1+x}$$

`Factor` first puts all terms over a common denominator, then factors the result.

```
In[12]:= Factor[%]
```

$$\text{Out[12]}= \frac{2\,(-2+x)\,(2+x)}{(-1+x)\,(1+x)}$$

In mathematical terms, `Apart` decomposes a rational expression into "partial fractions".

In expressions with several variables, you can use `Apart[`*expr*`,` *var*`]` to do partial fraction decompositions with respect to different variables.

Here is a rational expression in two variables.

```
In[13]:= v = (x^2 + y^2) / (x + x y)
```

$$\text{Out[13]}= \frac{x^2+y^2}{x+x\,y}$$

This gives the partial fraction decomposition with respect to `x`.

```
In[14]:= Apart[v, x]
```

$$\text{Out[14]}= \frac{x}{1+y}+\frac{y^2}{x\,(1+y)}$$

Here is the partial fraction decomposition with respect to `y`.

```
In[15]:= Apart[v, y]
```

$$\text{Out[15]}= -\frac{1}{x}+\frac{y}{x}+\frac{1+x^2}{x\,(1+y)}$$

Algebraic Operations on Polynomials

For many kinds of practical calculations, the only operations you will need to perform on polynomials are essentially the structural ones already discussed.

If you do more advanced algebra with polynomials, however, you will have to use the algebraic operations discussed in this tutorial.

You should realize that most of the operations discussed in this tutorial work only on ordinary polynomials, with integer exponents and rational-number coefficients for each term.

`PolynomialQuotient[`$poly_1$`,`$poly_2$`,`x`]`	find the result of dividing the polynomial $poly_1$ in x by $poly_2$, dropping any remainder term
`PolynomialRemainder[`$poly_1$`,`$poly_2$`,`x`]`	find the remainder from dividing the polynomial $poly_1$ in x by $poly_2$
`PolynomialQuotientRemainder[`$poly_1$`,`$poly_2$`,`x`]`	give the quotient and remainder in a list
`PolynomialMod[`$poly$`,`m`]`	reduce the polynomial $poly$ modulo m
`PolynomialGCD[`$poly_1$`,`$poly_2$`]`	find the greatest common divisor of two polynomials
`PolynomialLCM[`$poly_1$`,`$poly_2$`]`	find the least common multiple of two polynomials
`PolynomialExtendedGCD[`$poly_1$`,`$poly_2$`]`	find the extended greatest common divisor of two polynomials
`Resultant[`$poly_1$`,`$poly_2$`,`x`]`	find the resultant of two polynomials
`Subresultants[`$poly_1$`,`$poly_2$`,`x`]`	find the principal subresultant coefficients of two polynomials
`Discriminant[`$poly$`,`x`]`	find the discriminant of the polynomial $poly$
`GroebnerBasis[{`$poly_1$`,`$poly_2$`,…},{`x_1`,`x_2`,…}]`	find the Gröbner basis for the polynomials $poly_i$
`GroebnerBasis[{`$poly_1$`,`$poly_2$`,…},{`x_1`,`x_2`,…},{`y_1`,`y_2`,…}]`	find the Gröbner basis eliminating the y_i
`PolynomialReduce[`$poly$`,{`$poly_1$`,`$poly_2$`,…},{`x_1`,`x_2`,…}]`	find a minimal representation of $poly$ in terms of the $poly_i$

Reduction of polynomials.

Given two polynomials $p(x)$ and $q(x)$, one can always uniquely write $\frac{p(x)}{q(x)} = a(x) + \frac{b(x)}{q(x)}$, where the degree of $b(x)$ is less than the degree of $q(x)$. `PolynomialQuotient` gives the quotient $a(x)$, and `PolynomialRemainder` gives the remainder $b(x)$.

This gives the remainder from dividing x^2 by $1 + x$.

```
In[1]:= PolynomialRemainder[x^2, x + 1, x]
Out[1]= 1
```

Here is the quotient of x^2 and $x+1$, with the remainder dropped.

```
In[2]:= PolynomialQuotient[x^2, x + 1, x]
Out[2]= -1 + x
```

This gives back the original expression.

```
In[3]:= Simplify[(x + 1) % + %%]
Out[3]= x^2
```

Here the result depends on whether the polynomials are considered to be in `x` or `y`.

```
In[4]:= {PolynomialRemainder[x + y, x - y, x], PolynomialRemainder[x + y, x - y, y]}
Out[4]= {2 y, 2 x}
```

`PolynomialMod` is essentially the analog for polynomials of the function `Mod` for integers. When the modulus m is an integer, `PolynomialMod[`*poly*`,` m`]` simply reduces each coefficient in *poly* modulo the integer m. If m is a polynomial, then `PolynomialMod[`*poly*`,` m`]` effectively tries to get a polynomial with as low a degree as possible by subtracting from *poly* appropriate multiples $q\,m$ of m. The multiplier q can itself be a polynomial, but its degree is always less than the degree of *poly*. `PolynomialMod` yields a final polynomial whose degree and leading coefficient are both as small as possible.

This reduces x^2 modulo $x+1$. The result is simply the remainder from dividing the polynomials.

```
In[5]:= PolynomialMod[x^2, x + 1]
Out[5]= 1
```

In this case, `PolynomialMod` and `PolynomialRemainder` do not give the same result.

```
In[6]:= {PolynomialMod[x^2, a x + 1], PolynomialRemainder[x^2, a x + 1, x]}
Out[6]= {x^2, 1/a^2}
```

The main difference between `PolynomialMod` and `PolynomialRemainder` is that while the former works simply by multiplying and subtracting polynomials, the latter uses division in getting its results. In addition, `PolynomialMod` allows reduction by several moduli at the same time. A typical case is reduction modulo both a polynomial and an integer.

This reduces the polynomial $x^2 + 1$ modulo both $x + 1$ and 2.

```
In[7]:= PolynomialMod[x^2 + 1, {x + 1, 2}]
Out[7]= 0
```

`PolynomialGCD[`$poly_1$`,` $poly_2$`]` finds the highest degree polynomial that divides the $poly_i$ exactly. It gives the analog for polynomials of the integer function `GCD`.

`PolynomialGCD` gives the greatest common divisor of the two polynomials.

```
In[8]:= PolynomialGCD[(1 - x)^2 (1 + x) (2 + x), (1 - x) (2 + x) (3 + x)]
Out[8]= (-1 + x) (2 + x)
```

`PolynomialExtendedGCD` gives the extended greatest common divisor of the two polynomials.

```
In[9]:= {g, {r, s}} = PolynomialExtendedGCD[x^3 + 2 x^2 - x + 1, x^4 + x + 2, x]
```

$$\text{Out[9]= } \left\{1, \left\{\frac{29}{215} - \frac{26\,x}{215} - \frac{23\,x^2}{215} + \frac{21\,x^3}{215}, \frac{93}{215} - \frac{19\,x}{215} - \frac{21\,x^2}{215}\right\}\right\}$$

The returned polynomials r and s can be used to represent the GCD in terms of the original polynomials.

```
In[10]:= r (x^3 + 2 x^2 - x + 1) + s (x^4 + x + 2) // Expand
Out[10]= 1
```

The function `Resultant[`$poly_1$`,` $poly_2$`,` x`]` is used in a number of classical algebraic algorithms. The resultant of two polynomials a and b, both with leading coefficient one, is given by the product of all the differences $a_i - b_j$ between the roots of the polynomials. It turns out that for any pair of polynomials, the resultant is always a polynomial in their coefficients. By looking at when the resultant is zero, you can tell for what values of their parameters two polynomials have a common root. Two polynomials with leading coefficient one have k common roots if exactly the first k elements in the list `Subresultants[`$poly_1$`,` $poly_2$`,` x`]` are zero.

Here is the resultant with respect to y of two polynomials in x and y. The original polynomials have a common root in y only for values of x at which the resultant vanishes.

```
In[11]:= Resultant[(x - y)^2 - 2, y^2 - 3, y]
Out[11]= 1 - 10 x^2 + x^4
```

The function `Discriminant[poly, x]` is the product of the squares of the differences of its roots. It can be used to determine whether the polynomial has any repeated roots. The discriminant is equal to the resultant of the polynomial and its derivative, up to a factor independent of the variable.

This polynomial has a repeated root, so its discriminant vanishes.

```
In[12]:= Discriminant[(x - 1) ^ 2, x]
Out[12]= 0
```

This polynomial has distinct roots, so its discriminant is nonzero.

```
In[13]:= Discriminant[x ^ 4 - 1, x]
Out[13]= -256
```

Gröbner bases appear in many modern algebraic algorithms and applications. The function `GroebnerBasis[{poly1, poly2, ...}, {x1, x2, ...}]` takes a set of polynomials, and reduces this set to a canonical form from which many properties can conveniently be deduced. An important feature is that the set of polynomials obtained from `GroebnerBasis` always has exactly the same collection of common roots as the original set.

The $(x + y)^2$ is effectively redundant, and so does not appear in the Gröbner basis.

```
In[14]:= GroebnerBasis[{(x + y), (x + y) ^ 2}, {x, y}]
Out[14]= {x + y}
```

The polynomial `1` has no roots, showing that the original polynomials have no common roots.

```
In[15]:= GroebnerBasis[{x + y, x ^ 2 - 1, y ^ 2 - 2 x}, {x, y}]
Out[15]= {1}
```

The polynomials are effectively unwound here, and can now be seen to have exactly five common roots.

```
In[16]:= GroebnerBasis[{x y ^ 2 + 2 x y + x ^ 2 + 1, x y + y ^ 2 + 1}, {x, y}]
Out[16]= {-1 - y^2 + y^3 + y^4 + y^5, x + y^2 + y^3 + y^4}
```

`PolynomialReduce[poly, {p1, p2, ...}, {x1, x2, ...}]` yields a list $\{\{a_1, a_2, \ldots\}, b\}$ of polynomials with the property that b is minimal and $a_1\, p_1 + a_2\, p_2 + \ldots + b$ is exactly $poly$.

This writes $x^2 + y^2$ in terms of $x - y$ and $y + a$, leaving a remainder that depends only on a.

```
In[17]:= PolynomialReduce[x^2 + y^2, {x - y, y + a}, {x, y}]
Out[17]= {{x + y, -2 a + 2 y}, 2 a^2}
```

`Factor[`*poly*`]`	factor a polynomial
`FactorSquareFree[`*poly*`]`	write a polynomial as a product of powers of square-free factors
`FactorTerms[`*poly*`,`*x*`]`	factor out terms that do not depend on *x*
`FactorList[`*poly*`]`, `FactorSquareFreeList[`*poly*`]`, `FactorTermsList[`*poly*`]`	give results as lists of factors

Functions for factoring polynomials.

`Factor`, `FactorTerms` and `FactorSquareFree` perform various degrees of factoring on polynomials. `Factor` does full factoring over the integers. `FactorTerms` extracts the "content" of the polynomial. `FactorSquareFree` pulls out any multiple factors that appear.

Here is a polynomial, in expanded form.

```
In[18]:= t = Expand[2 (1 + x)^2 (2 + x) (3 + x)]
Out[18]= 12 + 34 x + 34 x^2 + 14 x^3 + 2 x^4
```

`FactorTerms` pulls out only the factor of 2 that does not depend on x.

```
In[19]:= FactorTerms[t, x]
Out[19]= 2 (6 + 17 x + 17 x^2 + 7 x^3 + x^4)
```

`FactorSquareFree` factors out the 2 and the term $(1 + x)$^2, but leaves the rest unfactored.

```
In[20]:= FactorSquareFree[t]
Out[20]= 2 (1 + x)^2 (6 + 5 x + x^2)
```

`Factor` does full factoring, recovering the original form.

```
In[21]:= Factor[t]
Out[21]= 2 (1 + x)^2 (2 + x) (3 + x)
```

Particularly when you write programs that work with polynomials, you will often find it convenient to pick out pieces of polynomials in a standard form. The function `FactorList` gives a list of all the factors of a polynomial, together with their exponents. The first element of the list is always the overall numerical factor for the polynomial.

The form that FactorList returns is the analog for polynomials of the form produced by FactorInteger for integers.

Here is a list of the factors of the polynomial in the previous set of examples. Each element of the list gives the factor, together with its exponent.

```
In[22]:= FactorList[t]
Out[22]= {{2, 1}, {1 + x, 2}, {2 + x, 1}, {3 + x, 1}}
```

Factor[*poly*, GaussianIntegers->True]	factor a polynomial, allowing coefficients that are Gaussian integers

Factoring polynomials with complex coefficients.

Factor and related functions usually handle only polynomials with ordinary integer or rational-number coefficients. If you set the option GaussianIntegers -> True, however, then Factor will allow polynomials with coefficients that are complex numbers with rational real and imaginary parts. This often allows more extensive factorization to be performed.

This polynomial is irreducible when only ordinary integers are allowed.

```
In[23]:= Factor[1 + x^2]
Out[23]= 1 + x^2
```

When Gaussian integer coefficients are allowed, the polynomial factors.

```
In[24]:= Factor[1 + x^2, GaussianIntegers -> True]
Out[24]= (-i + x) (i + x)
```

IrreduciblePolynomialQ[*poly*]	test whether *poly* is an irreducible polynomial over the rationals
IrreduciblePolynomialQ[*poly*, GaussianIntegers→True]	test whether *poly* is irreducible over the Gaussian rationals
IrreduciblePolynomialQ[*poly*, Extension→Automatic]	test irreducibility over the rationals extended by the algebraic number coefficients of *poly*

Irreducibility testing.

A polynomial is irreducible over a field `F` if it cannot be represented as a product of two nonconstant polynomials with coefficients in `F`.

This polynomial is irreducible over the rationals.

```
In[25]:= IrreduciblePolynomialQ[x^2 + 4]
Out[25]= True
```

Over the Gaussian rationals, the polynomial is reducible.

```
In[26]:= IrreduciblePolynomialQ[x^2 + 4, GaussianIntegers -> True]
Out[26]= False
```

By default, algebraic numbers are treated as independent variables.

```
In[27]:= IrreduciblePolynomialQ[x^2 + 2 Sqrt[2] x + 2]
Out[27]= True
```

Over the rationals extended by `Sqrt[2]`, the polynomial is reducible.

```
In[28]:= IrreduciblePolynomialQ[x^2 + 2 Sqrt[2] x + 2, Extension → Automatic]
Out[28]= False
```

`Cyclotomic[n,x]`	give the cyclotomic polynomial of order n in x

Cyclotomic polynomials.

Cyclotomic polynomials arise as "elementary polynomials" in various algebraic algorithms. The cyclotomic polynomials are defined by $C_n(x) = \prod_k \left(x - e^{2\pi i k/n}\right)$, where k runs over all positive integers less than n that are relatively prime to n.

This is the cyclotomic polynomial $C_6(x)$.

```
In[29]:= Cyclotomic[6, x]
Out[29]= 1 - x + x^2
```

$C_6(x)$ appears in the factors of $x^6 - 1$.

```
In[30]:= Factor[x^6 - 1]
Out[30]= (-1 + x) (1 + x) (1 - x + x^2) (1 + x + x^2)
```

Decompose[*poly*, *x*]	decompose *poly*, if possible, into a composition of a list of simpler polynomials

Decomposing polynomials.

Factorization is one important way of breaking down polynomials into simpler parts. Another, quite different, way is *decomposition*. When you factor a polynomial $P(x)$, you write it as a product $p_1(x)\, p_2(x) \ldots$ of polynomials $p_i(x)$. Decomposing a polynomial $Q(x)$ consists of writing it as a *composition* of polynomials of the form $q_1(q_2(\ldots(x)\ldots))$.

Here is a simple example of Decompose. The original polynomial $x^4 + x^2 + 1$ can be written as the polynomial $\bar{x}^2 + \bar{x} + 1$, where $\bar{x}$ is the polynomial x^2.

```
In[31]:= Decompose[x^4 + x^2 + 1, x]
Out[31]= {1 + x + x^2, x^2}
```

Here are two polynomial functions.

```
In[32]:= (q1[x_] = 1 - 2 x + x^4; q2[x_] = 5 x + x^3;)
```

This gives the composition of the two functions.

```
In[33]:= Expand[q1[q2[x]]]
Out[33]= 1 - 10 x - 2 x^3 + 625 x^4 + 500 x^6 + 150 x^8 + 20 x^10 + x^12
```

Decompose recovers the original functions.

```
In[34]:= Decompose[%, x]
Out[34]= {1 - 2 x + x^4, 5 x + x^3}
```

Decompose[*poly*, *x*] is set up to give a list of polynomials in x, which, if composed, reproduce the original polynomial. The original polynomial can contain variables other than x, but the sequence of polynomials that Decompose produces are all intended to be considered as functions of x.

Unlike factoring, the decomposition of polynomials is not completely unique. For example, the two sets of polynomials p_i and q_i, related by $q_1(x) = p_1(x - a)$ and $q_2(x) = p_2(x) + a$ give the same result on composition, so that $p_1(p_2(x)) = q_1(q_2(x))$. *Mathematica* follows the convention of absorbing any constant terms into the first polynomial in the list produced by Decompose.

`InterpolatingPolynomial[{`f_1`,`f_2`,...},`x`]`	give a polynomial in x which is equal to f_i when x is the integer i
`InterpolatingPolynomial[{{`x_1`,`f_1`},{`x_2`,`f_2`},...},`x`]`	give a polynomial in x which is equal to f_i when x is x_i

Generating interpolating polynomials.

This yields a quadratic polynomial which goes through the specified three points.

```
In[35]:= InterpolatingPolynomial[{{-1, 4}, {0, 2}, {1, 6}}, x]
Out[35]= 4 + (1 + x) (-2 + 3 x)
```

When `x` is `0`, the polynomial has value `2`.

```
In[36]:= % /. x -> 0
Out[36]= 2
```

Polynomials Modulo Primes

Mathematica can work with polynomials whose coefficients are in the finite field Z_p of integers modulo a prime p.

`PolynomialMod[`$poly$`,`p`]`	reduce the coefficients in a polynomial modulo p
`Expand[`$poly$`,Modulus->`p`]`	expand $poly$ modulo p
`Factor[`$poly$`,Modulus->`p`]`	factor $poly$ modulo p
`PolynomialGCD[`$poly_1$`,`$poly_2$`,Modulus->`p`]`	find the GCD of the $poly_i$ modulo p
`GroebnerBasis[`$polys$`,`$vars$`,Modulus->`p`]`	find the Gröbner basis modulo p

Functions for manipulating polynomials over finite fields.

Here is an ordinary polynomial.

```
In[1]:= Expand[(1 + x) ^ 6]
Out[1]= 1 + 6 x + 15 x^2 + 20 x^3 + 15 x^4 + 6 x^5 + x^6
```

This reduces the coefficients modulo 2.

```
In[2]:= PolynomialMod[%, 2]
Out[2]= 1 + x^2 + x^4 + x^6
```

Here are the factors of the resulting polynomial over the integers.

```
In[3]:= Factor[%]
Out[3]= (1 + x^2) (1 + x^4)
```

If you work modulo 2, further factoring becomes possible.

```
In[4]:= Factor[%, Modulus -> 2]
Out[4]= (1 + x)^6
```

Symmetric Polynomials

A *symmetric polynomial* in variables $x_1, \ldots, x_n$ is a polynomial that is invariant under arbitrary permutations of $x_1, \ldots, x_n$. Polynomials

$$s_1 = x_1 + x_2 + \ldots + x_n$$
$$s_2 = x_1 x_2 + x_1 x_3 + \ldots + x_{n-1} x_n$$
$$\ldots$$
$$s_n = x_1 x_2 \ldots x_n$$

are called *elementary symmetric polynomials* in variables $x_1, \ldots, x_n$.

The fundamental theorem of symmetric polynomials says that every symmetric polynomial in $x_1, \ldots, x_n$ can be represented as a polynomial in elementary symmetric polynomials in $x_1, \ldots, x_n$.

When the ordering of variables is fixed, an arbitrary polynomial f can be uniquely represented as a sum of a symmetric polynomial p, called the symmetric part of f, and a remainder q that does not contain descending monomials. A monomial $c\, x_1^{e_1} \ldots x_n^{e_n}$ is called descending iff $e_1 \geq \ldots \geq e_n$.

SymmetricPolynomial[k, {x_1, …, x_n}]	give the k^{th} elementary symmetric polynomial in the variables x_1, …, x_n
SymmetricReduction[f, {x_1, …, x_n}]	give a pair of polynomials $\{p, q\}$ in x_1, …, x_n such that $f == p + q$, where p is the symmetric part and q is the remainder
SymmetricReduction[f, {x_1, …, x_n}, {s_1, …, s_n}]	give the pair $\{p, q\}$ with the elementary symmetric polynomials in p replaced by s_1, …, s_n

Functions for symmetric polynomial computations.

Here is the elementary symmetric polynomial of degree three in four variables.

```
In[1]:= SymmetricPolynomial[3, {x, y, z, t}]
Out[1]= t x y + t x z + t y z + x y z
```

This writes the polynomial $(x + y)^2 + (x + z)^2 + (z + y)^2$ in terms of elementary symmetric polynomials. The input polynomial is symmetric, so the remainder is zero.

```
In[2]:= SymmetricReduction[(x + y)^2 + (x + z)^2 + (z + y)^2, {x, y, z}]
Out[2]= {2 (x + y + z)^2 - 2 (x y + x z + y z), 0}
```

Here the elementary symmetric polynomials in the symmetric part are replaced with variables s_1, s_2, s_3. The polynomial is not symmetric, so the remainder is not zero.

```
In[3]:= SymmetricReduction[x^5 + y^5 + z^4, {x, y, z}, {s1, s2, s3}]
Out[3]= {s1^5 - 5 s1^3 s2 + 5 s1 s2^2 + 5 s1^2 s3 - 5 s2 s3, z^4 - z^5}
```

SymmetricReduction can be applied to polynomials with symbolic coefficients.

```
In[4]:= SymmetricReduction[x^5 + y^5 + z^4 + a x^4 + b y^4 + c z^5, {x, y, z}, {s1, s2, s3}]
Out[4]= {a s1^4 + s1^5 - 4 a s1^2 s2 - 5 s1^3 s2 + 2 a s2^2 + 5 s1 s2^2 + 4 a s1 s3 + 5 s1^2 s3 - 5 s2 s3, (-a + b) y^4 + (1 - a) z^4 + (-1 + c) z^5}
```

Polynomials over Algebraic Number Fields

Functions like Factor usually assume that all coefficients in the polynomials they produce must involve only rational numbers. But by setting the option Extension you can extend the domain of coefficients that will be allowed.

`Factor[`*poly*`,Extension->{`a_1`,`a_2`,...}]`

factor *poly* allowing coefficients that are rational combinations of the a_i

Factoring polynomials over algebraic number fields.

Allowing only rational number coefficients, this polynomial cannot be factored.

```
In[1]:= Factor[1 + x^4]
Out[1]= 1 + x^4
```

With coefficients that can involve $\sqrt{2}$, the polynomial can now be factored.

```
In[2]:= Factor[1 + x^4, Extension -> {Sqrt[2]}]
Out[2]= -(-1 + Sqrt[2] x - x^2) (1 + Sqrt[2] x + x^2)
```

The polynomial can also be factored if one allows coefficients involving $\sqrt{-1}$.

```
In[3]:= Factor[1 + x^4, Extension -> {Sqrt[-1]}]
Out[3]= (-i + x^2) (i + x^2)
```

`GaussianIntegers -> True` is equivalent to `Extension -> Sqrt[-1]`.

```
In[4]:= Factor[1 + x^4, GaussianIntegers -> True]
Out[4]= (-i + x^2) (i + x^2)
```

If one allows coefficients that involve both $\sqrt{2}$ and $\sqrt{-1}$ the polynomial can be factored completely.

```
In[5]:= Factor[1 + x^4, Extension -> {Sqrt[2], Sqrt[-1]}]
Out[5]= 1/4 (Sqrt[2] - (1 + i) x) (Sqrt[2] - (1 - i) x) (Sqrt[2] + (1 - i) x) (Sqrt[2] + (1 + i) x)
```

`Expand` gives the original polynomial back again.

```
In[6]:= Expand[%]
Out[6]= 1 + x^4
```

`Factor[`*poly*`,Extension->Automatic]`

factor *poly* allowing algebraic numbers in *poly* to appear in coefficients

Factoring polynomials with algebraic number coefficients.

Here is a polynomial with a coefficient involving $\sqrt{2}$.

```
In[7]:= t = Expand[(Sqrt[2] + x)^2]
```

$$\text{Out[7]= } 2 + 2\sqrt{2}\,x + x^2$$

By default, `Factor` will not factor this polynomial.

```
In[8]:= Factor[t]
```

$$\text{Out[8]= } 2 + 2\sqrt{2}\,x + x^2$$

But now the field of coefficients is extended by including $\sqrt{2}$, and the polynomial is factored.

```
In[9]:= Factor[t, Extension -> Automatic]
```

$$\text{Out[9]= } \left(\sqrt{2} + x\right)^2$$

Other polynomial functions work much like `Factor`. By default, they treat algebraic number coefficients just like independent symbolic variables. But with the option `Extension -> Automatic` they perform operations on these coefficients.

By default, `Cancel` does not reduce these polynomials.

```
In[10]:= Cancel[t / (x^2 - 2)]
```

$$\text{Out[10]= } \frac{2 + 2\sqrt{2}\,x + x^2}{-2 + x^2}$$

But now it does.

```
In[11]:= Cancel[t / (x^2 - 2), Extension -> Automatic]
```

$$\text{Out[11]= } \frac{-\sqrt{2} - x}{\sqrt{2} - x}$$

By default, `PolynomialLCM` pulls out no common factors.

```
In[12]:= PolynomialLCM[t, x^2 - 2]
```

$$\text{Out[12]= } \left(-2 + x^2\right)\left(2 + 2\sqrt{2}\,x + x^2\right)$$

But now it does.

```
In[13]:= PolynomialLCM[t, x^2 - 2, Extension -> Automatic]
Out[13]= -2 √2 - 2 x + √2 x^2 + x^3
```

IrreduciblePolynomialQ[*poly*, Extension→Automatic]	test whether *poly* is an irreducible polynomial over the rationals extended by the coefficients of *poly*
IrreduciblePolynomialQ[*poly*, Extension->{a_1, a_2, …}]	test whether *poly* is irreducible over the rationals extended by the coefficients of *poly* and by a_1, a_2, …
IrreduciblePolynomialQ[*poly*, Extension→All]	test irreducibility over the field of all complex numbers

Irreducibility testing.

A polynomial is irreducible over a field F if it cannot be represented as a product of two nonconstant polynomials with coefficients in F.

By default, algebraic numbers are treated as independent variables.

```
In[14]:= IrreduciblePolynomialQ[x^2 + 2 Sqrt[2] x + 2]
Out[14]= True
```

Over the rationals extended by Sqrt[2], the polynomial is reducible.

```
In[15]:= IrreduciblePolynomialQ[x^2 + 2 Sqrt[2] x + 2, Extension → Automatic]
Out[15]= False
```

This polynomial is irreducible over the rationals.

```
In[16]:= IrreduciblePolynomialQ[x^2 - 3]
Out[16]= True
```

Over the rationals extended by Sqrt[3], the polynomial is reducible.

```
In[17]:= IrreduciblePolynomialQ[x^2 - 3, Extension → {Sqrt[3]}]
Out[17]= False
```

This polynomial is irreducible over the field of all complex numbers.

```
In[18]:= IrreduciblePolynomialQ[x^3 - 5 x y + 7, Extension → All]
Out[18]= True
```

Trigonometric Expressions

TrigExpand[*expr*]	expand trigonometric expressions out into a sum of terms
TrigFactor[*expr*]	factor trigonometric expressions into products of terms
TrigFactorList[*expr*]	give terms and their exponents in a list
TrigReduce[*expr*]	reduce trigonometric expressions using multiple angles

Functions for manipulating trigonometric expressions.

This expands out a trigonometric expression.

```
In[1]:= TrigExpand[Sin[2 x] Cos[2 y]]
```

$$\text{Out[1]}= 2\,\text{Cos}[x]\,\text{Cos}[y]^2\,\text{Sin}[x] - 2\,\text{Cos}[x]\,\text{Sin}[x]\,\text{Sin}[y]^2$$

This factors the expression.

```
In[2]:= TrigFactor[%]
```

$$\text{Out[2]}= 4\,\text{Cos}[x]\,\text{Sin}[x]\,\text{Sin}\left[\frac{\pi}{4} - y\right]\,\text{Sin}\left[\frac{\pi}{4} + y\right]$$

And this reduces the expression to a form that is linear in the trigonometric functions.

```
In[3]:= TrigReduce[%]
```

$$\text{Out[3]}= \frac{1}{2}\,(\text{Sin}[2x - 2y] + \text{Sin}[2x + 2y])$$

TrigExpand works on hyperbolic as well as circular functions.

```
In[4]:= TrigExpand[Tanh[x + y]]
```

$$\text{Out[4]}= \frac{\text{Cosh}[y]\,\text{Sinh}[x]}{\text{Cosh}[x]\,\text{Cosh}[y] + \text{Sinh}[x]\,\text{Sinh}[y]} + \frac{\text{Cosh}[x]\,\text{Sinh}[y]}{\text{Cosh}[x]\,\text{Cosh}[y] + \text{Sinh}[x]\,\text{Sinh}[y]}$$

TrigReduce reproduces the original form again.

```
In[5]:= TrigReduce[%]
Out[5]= Tanh[x + y]
```

Mathematica automatically uses functions like Tan whenever it can.

```
In[6]:= Sin[x]^2 / Cos[x]
Out[6]= Sin[x] Tan[x]
```

With TrigFactorList, however, you can see the parts of functions like Tan.

```
In[7]:= TrigFactorList[%]
Out[7]= {{1, 1}, {Sin[x], 2}, {Cos[x], -1}}
```

TrigToExp[*expr*]	write trigonometric functions in terms of exponentials
ExpToTrig[*expr*]	write exponentials in terms of trigonometric functions

Converting to and from exponentials.

TrigToExp writes trigonometric functions in terms of exponentials.

```
In[8]:= TrigToExp[Tan[x]]
```

$$\text{Out[8]}= \frac{i\left(e^{-ix}-e^{ix}\right)}{e^{-ix}+e^{ix}}$$

TrigToExp also works with hyperbolic functions.

```
In[9]:= TrigToExp[Tanh[x]]
```

$$\text{Out[9]}= \frac{-e^{-x}+e^{x}}{e^{-x}+e^{x}}$$

ExpToTrig does the reverse, getting rid of explicit complex numbers whenever possible.

```
In[10]:= ExpToTrig[%]
Out[10]= Tanh[x]
```

ExpToTrig deals with hyperbolic as well as circular functions.

```
In[11]:= ExpToTrig[Exp[x] - Exp[-x]]
Out[11]= 2 Sinh[x]
```

You can also use ExpToTrig on purely numerical expressions.

```
In[12]:= ExpToTrig[(-1)^(1 / 17)]
```

$$\text{Out[12]}= \text{Cos}\left[\frac{\pi}{17}\right]+i\,\text{Sin}\left[\frac{\pi}{17}\right]$$

Expressions Involving Complex Variables

Mathematica usually pays no attention to whether variables like x stand for real or complex numbers. Sometimes, however, you may want to make transformations which are appropriate only if particular variables are assumed to be either real or complex.

The function ComplexExpand expands out algebraic and trigonometric expressions, making definite assumptions about the variables that appear.

ComplexExpand[*expr*]	expand *expr* assuming that all variables are real
ComplexExpand[*expr*, {x_1, x_2, ...}]	expand *expr* assuming that the x_i are complex

Expanding complex expressions.

This expands the expression, assuming that x and y are both real.

```
In[1]:= ComplexExpand[Tan[x + I y]]
```

$$\text{Out[1]}= \frac{\text{Sin}[2\,x]}{\text{Cos}[2\,x]+\text{Cosh}[2\,y]}+\frac{i\,\text{Sinh}[2\,y]}{\text{Cos}[2\,x]+\text{Cosh}[2\,y]}$$

In this case, a is assumed to be real, but x is assumed to be complex, and is broken into explicit real and imaginary parts.

```
In[2]:= ComplexExpand[a + x^2, {x}]
Out[2]= a - Im[x]^2 + 2 i Im[x] Re[x] + Re[x]^2
```

With several complex variables, you quickly get quite complicated results.

```
In[3]:= ComplexExpand[Sin[x] Exp[y], {x, y}]
Out[3]= E^Re[y] Cos[Im[y]] Cosh[Im[x]] Sin[Re[x]] - E^Re[y] Cos[Re[x]] Sin[Im[y]] Sinh[Im[x]] +
         i (E^Re[y] Cosh[Im[x]] Sin[Im[y]] Sin[Re[x]] + E^Re[y] Cos[Im[y]] Cos[Re[x]] Sinh[Im[x]])
```

There are several ways to write a complex variable z in terms of real parameters. As above, for example, z can be written in the "Cartesian form" Re[z] + I Im[z]. But it can equally well be written in the "polar form" Abs[z] Exp[I Arg[z]].

The option TargetFunctions in ComplexExpand allows you to specify how complex variables should be written. TargetFunctions can be set to a list of functions from the set

{Re, Im, Abs, Arg, Conjugate, Sign}. ComplexExpand will try to give results in terms of whichever of these functions you request. The default is typically to give results in terms of Re and Im.

This gives an expansion in Cartesian form.

```
In[4]:= ComplexExpand[Re[z^2], {z}]
Out[4]= -Im[z]^2 + Re[z]^2
```

Here is an expansion in polar form.

```
In[5]:= ComplexExpand[Re[z^2], {z}, TargetFunctions -> {Abs, Arg}]
Out[5]= Abs[z]^2 Cos[Arg[z]]^2 - Abs[z]^2 Sin[Arg[z]]^2
```

Here is another form of expansion.

```
In[6]:= ComplexExpand[Re[z^2], {z}, TargetFunctions -> Conjugate]
Out[6]= z^2/2 + Conjugate[z]^2/2
```

Logical and Piecewise Functions

Nested logical and piecewise functions can be expanded out much like nested arithmetic functions. You can do this using LogicalExpand and PiecewiseExpand.

LogicalExpand[*expr*]	expand out logical functions in *expr*
PiecewiseExpand[*expr*]	expand out piecewise functions in *expr*
PiecewiseExpand[*expr*, *assum*]	expand out with the specified assumptions

Expanding out logical and piecewise functions.

LogicalExpand puts logical expressions into a standard *disjunctive normal form* (DNF), consisting of an OR of ANDs.

By default, *Mathematica* leaves this expression unchanged.

```
In[1]:= (a || b) && c
Out[1]= (a || b) && c
```

`LogicalExpand` expands this into an OR of ANDs.

```
In[2]:= LogicalExpand[%]
Out[2]= (a && c) || (b && c)
```

`LogicalExpand` works on all logical functions, always converting them into a standard OR of ANDs form. Sometimes the results are inevitably quite large.

`Xor` can be expressed as an OR of ANDs.

```
In[3]:= LogicalExpand[Xor[a, b, c]]
Out[3]= (a && b && c) || (a && ! b && ! c) || (b && ! a && ! c) || (c && ! a && ! b)
```

Any collection of nested conditionals can always in effect be flattened into a *piecewise normal form* consisting of a single `Piecewise` object. You can do this in *Mathematica* using `PiecewiseExpand`.

By default, *Mathematica* leaves this expression unchanged.

```
In[4]:= If[x > 0, If[x < 1, a, b], c]
Out[4]= If[x > 0, If[x < 1, a, b], c]
```

`PiecewiseExpand` flattens it into a single `Piecewise` object.

```
In[5]:= PiecewiseExpand[%]
Out[5]= ⎧ a   0 < x < 1
        ⎨ b   x ≥ 1
        ⎩ c   True
```

Functions like `Max` and `Abs`, as well as `Clip` and `UnitStep`, implicitly involve conditionals, and combinations of them can again be reduced to a single `Piecewise` object using `PiecewiseExpand`.

This gives a result as a single `Piecewise` object.

```
In[6]:= PiecewiseExpand[Max[Min[a, b], c]]
Out[6]= ⎧ a   a - c > 0 && a - b ≤ 0
        ⎨ b   a - b > 0 && b - c > 0
        ⎩ c   True
```

With `x` assumed real, this can also be written as a `Piecewise` object.

```
In[7]:= PiecewiseExpand[Abs[x], x ∈ Reals]
Out[7]= ⎧ -x   x < 0
        ⎩ x    True
```

Functions like `Floor`, `Mod` and `FractionalPart` can also be expressed in terms of `Piecewise` objects, though in principle they can involve an infinite number of cases.

Without a bound on `x`, this would yield an infinite number of cases.

```
In[8]:= PiecewiseExpand[Floor[x^2], 0 < x < 2]
```

$$\text{Out[8]}= \begin{cases} 1 & 1 \le x < \sqrt{2} \\ 2 & \sqrt{2} \le x < \sqrt{3} \\ 3 & x \ge \sqrt{3} \end{cases}$$

Mathematica by default limits the number of cases that *Mathematica* will explicitly generate in the expansion of any single piecewise function such as `Floor` at any stage in a computation. You can change this limit by resetting the value of `$MaxPiecewiseCases`.

Simplification

`Simplify[`*expr*`]`	try various algebraic and trigonometric transformations to simplify an expression
`FullSimplify[`*expr*`]`	try a much wider range of transformations

Simplifying expressions.

Mathematica does not automatically simplify an algebraic expression like this.

```
In[1]:= (1 - x) / (1 - x^2)
```

$$\text{Out[1]}= \frac{1 - x}{1 - x^2}$$

`Simplify` performs the simplification.

```
In[2]:= Simplify[%]
```

$$\text{Out[2]}= \frac{1}{1 + x}$$

`Simplify` performs standard algebraic and trigonometric simplifications.

```
In[3]:= Simplify[Sin[x]^2 + Cos[x]^2]
Out[3]= 1
```

It does not, however, do more sophisticated transformations that involve, for example, special functions.

```
In[4]:= Simplify[Gamma[1 + n] / n]
```

$$\text{Out[4]=}\ \frac{\text{Gamma}[1+n]}{n}$$

`FullSimplify` does perform such transformations.

```
In[5]:= FullSimplify[%]
Out[5]= Gamma[n]
```

`FullSimplify[`*expr*`,ExcludedForms->`*pattern*`]`

try to simplify *expr*, without touching subexpressions that match *pattern*

Controlling simplification.

Here is an expression involving trigonometric functions and square roots.

```
In[6]:= t = (1 - Sin[x]^2) Sqrt[Expand[(1 + Sqrt[2])^20]]
```

$$\text{Out[6]=}\ \sqrt{22\,619\,537 + 15\,994\,428\sqrt{2}}\ \left(1 - \text{Sin}[x]^2\right)$$

By default, `FullSimplify` will try to simplify everything.

```
In[7]:= FullSimplify[t]
```

$$\text{Out[7]=}\ \left(3363 + 2378\sqrt{2}\right)\text{Cos}[x]^2$$

This makes `FullSimplify` avoid simplifying the square roots.

```
In[8]:= FullSimplify[t, ExcludedForms -> Sqrt[_]]
```

$$\text{Out[8]=}\ \sqrt{22\,619\,537 + 15\,994\,428\sqrt{2}}\ \text{Cos}[x]^2$$

FullSimplify[*expr*, TimeConstraint->*t*]	try to simplify *expr*, working for at most *t* seconds on each transformation
FullSimplify[*expr*, TransformationFunctions->{f_1, f_2, ...}]	use only the functions f_i in trying to transform parts of *expr*
FullSimplify[*expr*, TransformationFunctions->{Automatic, f_1, f_2, ...}]	use built-in transformations as well as the f_i
Simplify[*expr*, ComplexityFunction->*c*] and FullSimplify[*expr*, ComplexityFunction->*c*]	simplify using *c* to determine what form is considered simplest

Further control of simplification.

In both Simplify and FullSimplify there is always an issue of what counts as the "simplest" form of an expression. You can use the option ComplexityFunction -> *c* to provide a function to determine this. The function will be applied to each candidate form of the expression, and the one that gives the smallest numerical value will be considered simplest.

With its default definition of simplicity, Simplify leaves this unchanged.

```
In[9]:= Simplify[4 Log[10]]
Out[9]= 4 Log[10]
```

This now tries to minimize the number of elements in the expression.

```
In[10]:= Simplify[4 Log[10], ComplexityFunction -> LeafCount]
Out[10]= Log[10 000]
```

Using Assumptions

Mathematica normally makes as few assumptions as possible about the objects you ask it to manipulate. This means that the results it gives are as general as possible. But sometimes these results are considerably more complicated than they would be if more assumptions were made.

`Refine[`*expr*`,`*assum*`]`	refine *expr* using assumptions
`Simplify[`*expr*`,`*assum*`]`	simplify with assumptions
`FullSimplify[`*expr*`,`*assum*`]`	full simplify with assumptions
`FunctionExpand[`*expr*`,`*assum*`]`	function expand with assumptions

Doing operations with assumptions.

`Simplify` by default does essentially nothing with this expression.

```
In[1]:= Simplify[1 / Sqrt[x] - Sqrt[1 / x]]
```

$$\text{Out[1]}= -\sqrt{\frac{1}{x}} + \frac{1}{\sqrt{x}}$$

The reason is that its value is quite different for different choices of x.

```
In[2]:= % /. x -> {-3, -2, -1, 1, 2, 3}
```

$$\text{Out[2]}= \left\{-\frac{2\,i}{\sqrt{3}},\ -i\sqrt{2},\ -2\,i,\ 0,\ 0,\ 0\right\}$$

With the assumption $x > 0$, `Simplify` can immediately reduce the expression to 0.

```
In[3]:= Simplify[1 / Sqrt[x] - Sqrt[1 / x], x > 0]
Out[3]= 0
```

Without making assumptions about x and y, nothing can be done.

```
In[4]:= FunctionExpand[Log[x y]]
Out[4]= Log[x y]
```

If x and y are both assumed positive, the log can be expanded.

```
In[5]:= FunctionExpand[Log[x y], x > 0 && y > 0]
Out[5]= Log[x] + Log[y]
```

By applying `Simplify` and `FullSimplify` with appropriate assumptions to equations and inequalities you can in effect establish a vast range of theorems.

Without making assumptions about x the truth or falsity of this equation cannot be determined.

```
In[6]:= Simplify[Abs[x] == x]
Out[6]= x == Abs[x]
```

Now Simplify can prove that the equation is true.

```
In[7]:= Simplify[Abs[x] == x, x > 0]
Out[7]= True
```

This establishes the standard result that the arithmetic mean is larger than the geometric one.

```
In[8]:= Simplify[(x + y) / 2 >= Sqrt[x y], x >= 0 && y >= 0]
Out[8]= True
```

This proves that $\mathrm{erf}(x)$ lies in the range $(0, 1)$ for all positive arguments.

```
In[9]:= FullSimplify[0 < Erf[x] < 1, x > 0]
Out[9]= True
```

Simplify and FullSimplify always try to find the simplest forms of expressions. Sometimes, however, you may just want *Mathematica* to follow its ordinary evaluation process, but with certain assumptions made. You can do this using Refine. The way it works is that Refine[*expr*, *assum*] performs the same transformations as *Mathematica* would perform automatically if the variables in *expr* were replaced by numerical expressions satisfying the assumptions *assum*.

There is no simpler form that Simplify can find.

```
In[10]:= Simplify[Log[x], x < 0]
Out[10]= Log[x]
```

Refine just evaluates Log[x] as it would for any explicit negative number x.

```
In[11]:= Refine[Log[x], x < 0]
Out[11]= i π + Log[-x]
```

An important class of assumptions is those which assert that some object is an element of a particular domain. You can set up such assumptions using $x \in dom$, where the $\in$ character can be entered as :el: or \[Element].

$x \in dom$ or Element[x, dom]	assert that x is an element of the domain dom
$\{x_1, x_2, \ldots\} \in dom$	assert that all the x_i are elements of the domain dom
$patt \in dom$	assert that any expression which matches $patt$ is an element of the domain dom

Asserting that objects are elements of domains.

This confirms that π is an element of the domain of real numbers.

```
In[12]:= Pi ∈ Reals
Out[12]= True
```

These numbers are all elements of the domain of algebraic numbers.

```
In[13]:= {1, Sqrt[2], 3 + Sqrt[5]} ∈ Algebraics
Out[13]= True
```

Mathematica knows that π is not an algebraic number.

```
In[14]:= Pi ∈ Algebraics
Out[14]= False
```

Current mathematics has not established whether $e + \pi$ is an algebraic number or not.

```
In[15]:= E + Pi ∈ Algebraics
Out[15]= ℯ + π ∈ Algebraics
```

This represents the assertion that the symbol `x` is an element of the domain of real numbers.

```
In[16]:= x ∈ Reals
Out[16]= x ∈ Reals
```

`Complexes`	the domain of complex numbers $\mathbb{C}$
`Reals`	the domain of real numbers $\mathbb{R}$
`Algebraics`	the domain of algebraic numbers $\mathbb{A}$
`Rationals`	the domain of rational numbers $\mathbb{Q}$
`Integers`	the domain of integers $\mathbb{Z}$
`Primes`	the domain of primes $\mathbb{P}$
`Booleans`	the domain of Booleans (`True` and `False`) $\mathbb{B}$

Domains supported by *Mathematica*.

If n is assumed to be an integer, $\sin(n\pi)$ is zero.

```
In[17]:= Simplify[Sin[n Pi], n ∈ Integers]
Out[17]= 0
```

This establishes the theorem $\cosh(x) \geq 1$ if x is assumed to be a real number.

```
In[18]:= Simplify[Cosh[x] >= 1, x ∈ Reals]
Out[18]= True
```

If you say that a variable satisfies an inequality, *Mathematica* will automatically assume that it is real.

```
In[19]:= Simplify[x ∈ Reals, x > 0]
Out[19]= True
```

By using `Simplify`, `FullSimplify` and `FunctionExpand` with assumptions you can access many of *Mathematica*'s vast collection of mathematical facts.

This uses the periodicity of the tangent function.

```
In[20]:= Simplify[Tan[x + Pi k], k ∈ Integers]
Out[20]= Tan[x]
```

The assumption $k / 2 \in$ `Integers` implies that k must be even.

```
In[21]:= Simplify[Tan[x + Pi k / 2], k / 2 ∈ Integers]
Out[21]= Tan[x]
```

Mathematica knows that $\log(x) < \exp(x)$ for positive x.

```
In[22]:= Simplify[Log[x] < Exp[x], x > 0]
Out[22]= True
```

`FullSimplify` accesses knowledge about special functions.

```
In[23]:= FullSimplify[Im[BesselJ[0, x]], x ∈ Reals]
Out[23]= 0
```

Mathematica knows about discrete mathematics and number theory as well as continuous mathematics.

This uses Wilson's theorem to simplify the result.

```
In[24]:= FunctionExpand[Mod[(p - 1)!, p], p ∈ Primes]
Out[24]= -1 + p
```

This uses the multiplicative property of the Euler phi function.

```
In[25]:= FunctionExpand[EulerPhi[m n], {m, n} ∈ Integers && GCD[m, n] == 1]
Out[25]= EulerPhi[m] EulerPhi[n]
```

In something like `Simplify[expr, assum]` or `Refine[expr, assum]` you explicitly give the assumptions you want to use. But sometimes you may want to specify one set of assumptions to use in a whole collection of operations. You can do this by using `Assuming`.

`Assuming[assum, expr]`	use assumptions *assum* in the evaluation of *expr*
`$Assumptions`	the default assumptions to use

Specifying assumptions with larger scopes.

This tells `Simplify` to use the default assumption $x > 0$.

```
In[26]:= Assuming[x > 0, Simplify[Sqrt[x^2]]]
Out[26]= x
```

This combines the two assumptions given.

```
In[27]:= Assuming[x > 0, Assuming[x ∈ Integers, Refine[Floor[Sqrt[x^2]]]]]
Out[27]= x
```

Functions like `Simplify` and `Refine` take the option `Assumptions`, which specifies what default assumptions they should use. By default, the setting for this option is `Assumptions :> $Assumptions`. The way `Assuming` then works is to assign a local value to `$Assumptions`, much as in `Block`.

In addition to `Simplify` and `Refine`, a number of other functions take `Assumptions` options, and thus can have assumptions specified for them by `Assuming`. Examples are `FunctionExpand`, `Integrate`, `Limit`, `Series`, `LaplaceTransform`.

The assumption is automatically used in `Integrate`.

```
In[28]:= Assuming[n > 0, 1 + Integrate[x^n, {x, 0, 1}]^2]
Out[28]= 1 + 1/(1 + n)^2
```

Manipulating Equations and Inequalities

Equations

"Defining Variables" discussed *assignments* such as $x = y$ which *set* x equal to y. Here we discuss *equations*, which *test* equality. The equation $x == y$ *tests* whether x is equal to y.

This *tests* whether 2 + 2 and 4 are equal. The result is the symbol `True`.

```
In[1]:= 2 + 2 == 4
Out[1]= True
```

It is very important that you do not confuse $x = y$ with $x == y$. While $x = y$ is an *imperative* statement that actually causes an assignment to be done, $x == y$ merely *tests* whether x and y are equal, and causes no explicit action. If you have used the C programming language, you will recognize that the notation for assignment and testing in *Mathematica* is the same as in C.

$x=y$	assigns x to have value y
$x==y$	tests whether x and y are equal

Assignments and tests.

This *assigns* `x` to have value `4`.

```
In[2]:= x = 4
Out[2]= 4
```

If you ask for `x`, you now get `4`.

```
In[3]:= x
Out[3]= 4
```

This *tests* whether `x` is equal to `4`. In this case, it is.

```
In[4]:= x == 4
Out[4]= True
```

`x` is equal to `4`, not `6`.

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

This removes the value assigned to `x`.

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

The tests we have used so far involve only numbers, and always give a definite answer, either `True` or `False`. You can also do tests on symbolic expressions.

Mathematica cannot get a definite result for this test unless you give `x` a specific numerical value.

```
In[7]:= x == 5
Out[7]= x == 5
```

If you replace `x` by the specific numerical value `4`, the test gives `False`.

```
In[8]:= % /. x -> 4
Out[8]= False
```

Even when you do tests on symbolic expressions, there are some cases where you can get definite results. An important one is when you test the equality of two expressions that are *identical*. Whatever the numerical values of the variables in these expressions may be, *Mathematica* knows that the expressions must always be equal.

The two expressions are *identical*, so the result is `True`, whatever the value of `x` may be.

```
In[9]:= 2 x + x^2 == 2 x + x^2
Out[9]= True
```

Mathematica does not try to tell whether these expressions are equal. In this case, using `Expand` would make them have the same form.

```
In[10]:= 2 x + x^2 == x (2 + x)
Out[10]= 2 x + x^2 == x (2 + x)
```

Expressions like `x == 4` represent *equations* in *Mathematica*. There are many functions in *Mathematica* for manipulating and solving equations.

This is an *equation* in *Mathematica*. "Solving Equations" discusses how to solve it for x.

```
In[11]:= x^2 + 2 x - 7 == 0
Out[11]= -7 + 2 x + x^2 == 0
```

You can assign a name to the equation.

```
In[12]:= eqn = %
Out[12]= -7 + 2 x + x^2 == 0
```

If you ask for eqn, you now get the equation.

```
In[13]:= eqn
Out[13]= -7 + 2 x + x^2 == 0
```

Solving Equations

An expression like x^2 + 2 x - 7 == 0 represents an *equation* in *Mathematica*. You will often need to *solve* equations like this, to find out for what values of x they are true.

This gives the two solutions to the quadratic equation $x^2 + 2x - 7 = 0$. The solutions are given as replacements for x.

```
In[1]:= Solve[x^2 + 2 x - 7 == 0, x]
Out[1]= {{x → -1 - 2 √2}, {x → -1 + 2 √2}}
```

Here are the numerical values of the solutions.

```
In[2]:= N[%]
Out[2]= {{x → -3.82843}, {x → 1.82843}}
```

You can get a list of the actual solutions for x by applying the rules generated by Solve to x using the replacement operator.

```
In[3]:= x /. %
Out[3]= {-3.82843, 1.82843}
```

You can equally well apply the rules to any other expression involving x.

```
In[4]:= x^2 + 3 x /. %%
Out[4]= {3.17157, 8.82843}
```

`Solve[`*lhs*`==`*rhs*`,`*x*`]`	solve an equation, giving a list of rules for *x*
x`/.`*solution*	use the list of rules to get values for *x*
expr`/.`*solution*	use the list of rules to get values for an expression

Finding and using solutions to equations.

`Solve` always tries to give you explicit *formulas* for the solutions to equations. However, it is a basic mathematical result that, for sufficiently complicated equations, explicit algebraic formulas in terms of radicals cannot be given. If you have an algebraic equation in one variable, and the highest power of the variable is at most four, then *Mathematica* can always give you formulas for the solutions. However, if the highest power is five or more, it may be mathematically impossible to give explicit algebraic formulas for all the solutions.

Mathematica can always solve algebraic equations in one variable when the highest power is less than five.

```
In[5]:= Solve[x^4 - 5 x^2 - 3 == 0, x]
```

$$\text{Out[5]= } \left\{\left\{x \to -\sqrt{\frac{5}{2}+\frac{\sqrt{37}}{2}}\right\}, \left\{x \to \sqrt{\frac{5}{2}+\frac{\sqrt{37}}{2}}\right\}, \left\{x \to -i\sqrt{\frac{1}{2}\left(-5+\sqrt{37}\right)}\right\}, \left\{x \to i\sqrt{\frac{1}{2}\left(-5+\sqrt{37}\right)}\right\}\right\}$$

It can solve some equations that involve higher powers.

```
In[6]:= Solve[x^6 == 1, x]
```

$$\text{Out[6]= } \left\{\{x \to -1\}, \{x \to 1\}, \left\{x \to -(-1)^{1/3}\right\}, \left\{x \to (-1)^{1/3}\right\}, \left\{x \to -(-1)^{2/3}\right\}, \left\{x \to (-1)^{2/3}\right\}\right\}$$

There are some equations, however, for which it is mathematically impossible to find explicit formulas for the solutions. *Mathematica* uses `Root` objects to represent the solutions in this case.

```
In[7]:= Solve[2 - 4 x + x^5 == 0, x]
```

$$\text{Out[7]= } \left\{\{x \to \text{Root}\left[2 - 4\,\#1 + \#1^5\,\&,\, 1\right]\}, \{x \to \text{Root}\left[2 - 4\,\#1 + \#1^5\,\&,\, 2\right]\}, \{x \to \text{Root}\left[2 - 4\,\#1 + \#1^5\,\&,\, 3\right]\}, \{x \to \text{Root}\left[2 - 4\,\#1 + \#1^5\,\&,\, 4\right]\}, \{x \to \text{Root}\left[2 - 4\,\#1 + \#1^5\,\&,\, 5\right]\}\right\}$$

Even though you cannot get explicit formulas, you can still evaluate the solutions numerically.

```
In[8]:= N[%]
Out[8]= {{x → -1.51851}, {x → 0.508499}, {x → 1.2436}, {x → -0.116792 - 1.43845 i}, {x → -0.116792 + 1.43845 i}}
```

In addition to being able to solve purely algebraic equations, *Mathematica* can also solve some equations involving other functions.

After printing a warning, *Mathematica* returns one solution to this equation.

```
In[9]:= Solve[Sin[x] == a, x]

    Solve::ifun: Inverse functions are being used by Solve, so some
        solutions may not be found; use Reduce for complete solution information. >>

Out[9]= {{x → ArcSin[a]}}
```

It is important to realize that an equation such as $\sin(x) = a$ actually has an infinite number of possible solutions, in this case differing by multiples of 2π. However, `Solve` by default returns just one solution, but prints a message telling you that other solutions may exist. You can use `Reduce` to get more information.

There is no explicit "closed form" solution for a transcendental equation like this.

```
In[10]:= Solve[Cos[x] == x, x]

    Solve::tdep:
      The equations appear to involve the variables to be solved for in an essentially non-algebraic way. >>

Out[10]= Solve[Cos[x] == x, x]
```

You can find an approximate numerical solution using `FindRoot`, and giving a starting value for `x`.

```
In[11]:= FindRoot[Cos[x] == x, {x, 0}]

Out[11]= {x → 0.739085}
```

`Solve` can also handle equations involving symbolic functions. In such cases, it again prints a warning, then gives results in terms of formal inverse functions.

Mathematica returns a result in terms of the formal inverse function of `f`.

```
In[12]:= Solve[f[x^2] == a, x]

    InverseFunction::ifun: Inverse functions are being used. Values may be lost for multivalued inverses. >>

Out[12]= {{x → -√(f^(-1)[a])}, {x → √(f^(-1)[a])}}
```

Solve[{*lhs*$_1$==*rhs*$_1$, *lhs*$_2$==*rhs*$_2$, ...}, {*x*, *y*, ...}]
solve a set of simultaneous equations for *x*, *y*, ...

Solving sets of simultaneous equations.

You can also use *Mathematica* to solve sets of simultaneous equations. You simply give the list of equations, and specify the list of variables to solve for.

Here is a list of two simultaneous equations, to be solved for the variables x and y.

```
In[13]:= Solve[{a x + y == 0, 2 x + (1 - a) y == 1}, {x, y}]
```

$$\text{Out[13]= } \left\{\left\{x \to -\frac{1}{-2+a-a^2},\ y \to -\frac{a}{2-a+a^2}\right\}\right\}$$

Here are some more complicated simultaneous equations. The two solutions are given as two lists of replacements for x and y.

```
In[14]:= Solve[{x^2 + y^2 == 1, x + 3 y == 0}, {x, y}]
```

$$\text{Out[14]= } \left\{\left\{x \to -\frac{3}{\sqrt{10}},\ y \to \frac{1}{\sqrt{10}}\right\},\ \left\{x \to \frac{3}{\sqrt{10}},\ y \to -\frac{1}{\sqrt{10}}\right\}\right\}$$

This uses the solutions to evaluate the expression x + y.

```
In[15]:= x + y /. %
```

$$\text{Out[15]= } \left\{-\sqrt{\frac{2}{5}},\ \sqrt{\frac{2}{5}}\right\}$$

Mathematica can solve any set of simultaneous *linear* or polynomial equations.

When you are working with sets of equations in several variables, it is often convenient to reorganize the equations by eliminating some variables between them.

This eliminates y between the two equations, giving a single equation for x.

```
In[16]:= Eliminate[{a x + y == 0, 2 x + (1 - a) y == 1}, y]
```

$$\text{Out[16]= } \left(2-a+a^2\right) x == 1$$

If you have several equations, there is no guarantee that there exists *any* consistent solution for a particular variable.

There is no consistent solution to these equations, so *Mathematica* returns `{}`, indicating that the set of solutions is empty.

```
In[17]:= Solve[{x == 1, x == 2}, x]
Out[17]= {}
```

There is also no consistent solution to these equations for almost all values of `a`.

```
In[18]:= Solve[{x == 1, x == a}, x]
Out[18]= {}
```

The general question of whether a set of equations has any consistent solution is quite a subtle one. For example, for most values of `a`, the equations `{x == 1, x == a}` are inconsistent, so there is no possible solution for `x`. However, if `a` is equal to `1`, then the equations *do* have a solution. `Solve` is set up to give you *generic* solutions to equations. It discards any solutions that exist only when special constraints between parameters are satisfied.

If you use `Reduce` instead of `Solve`, *Mathematica* will however keep *all* the possible solutions to a set of equations, including those that require special conditions on parameters.

This shows that the equations have a solution only when `a == 1`. The notation `a == 1 && x == 1` represents the requirement that *both* `a == 1` *and* `x == 1` should be `True`.

```
In[19]:= Reduce[{x == a, x == 1}, x]
Out[19]= a == 1 && x == 1
```

This gives the complete set of possible solutions to the equation. The answer is stated in terms of a combination of simpler equations. `&&` indicates equations that must simultaneously be true; `||` indicates alternatives.

```
In[20]:= Reduce[a x - b == 0, x]
Out[20]= (b == 0 && a == 0) || (a ≠ 0 && x == b/a)
```

This gives a more complicated combination of equations.

```
In[21]:= Reduce[a x^2 - b == 0, x]
Out[21]= (b == 0 && a == 0) || (a ≠ 0 && (x == -Sqrt[b]/Sqrt[a] || x == Sqrt[b]/Sqrt[a]))
```

This gives a symbolic representation of all solutions.

```
In[22]:= Reduce[Sin[x] == a, x]
Out[22]= C[1] ∈ Integers && (x == π - ArcSin[a] + 2 π C[1] || x == ArcSin[a] + 2 π C[1])
```

`Solve[`*lhs*`==`*rhs*`,`*x*`]`	solve an equation for *x*
`Solve[{`*lhs*$_1$`==`*rhs*$_1$`,`*lhs*$_2$`==`*rhs*$_2$`,…},{`*x*`,`*y*`,…}]`	solve a set of simultaneous equations for *x*, *y*, …
`Eliminate[{`*lhs*$_1$`==`*rhs*$_1$`,`*lhs*$_2$`==`*rhs*$_2$`,…},{`*x*`,…}]`	eliminate *x*, … in a set of simultaneous equations
`Reduce[{`*lhs*$_1$`==`*rhs*$_1$`,`*lhs*$_2$`==`*rhs*$_2$`,…},{`*x*`,`*y*`,…}]`	give a set of simplified equations, including all possible solutions

Functions for solving and manipulating equations.

`Reduce` also has powerful capabilities for handling equations specifically over real numbers or integers. "Equations and Inequalities over Domains" discusses this in more detail.

This reduces the equation assuming `x` and `y` are complex.

```
In[23]:= Reduce[x^2 + y^2 == 1, y]
Out[23]= y == -√(1 - x^2) || y == √(1 - x^2)
```

This includes the conditions for `x` and `y` to be real.

```
In[24]:= Reduce[x^2 + y^2 == 1, y, Reals]
Out[24]= -1 ≤ x ≤ 1 && (y == -√(1 - x^2) || y == √(1 - x^2))
```

This gives only the integer solutions.

```
In[25]:= Reduce[x^2 + y^2 == 1, y, Integers]
Out[25]= (x == -1 && y == 0) || (x == 0 && y == -1) || (x == 0 && y == 1) || (x == 1 && y == 0)
```

The Representation of Equations and Solutions

Mathematica treats equations as logical statements. If you type in an equation like `x^2 + 3 x == 2`, *Mathematica* interprets this as a logical statement which asserts that `x^2 + 3 x` is equal to `2`. If you have assigned an explicit value to `x`, say `x = 4`, then *Mathematica* can explicitly determine that the logical statement `x^2 + 3 x == 2` is `False`.

If you have not assigned any explicit value to `x`, however, *Mathematica* cannot work out whether `x^2 + 3 x == 2` is `True` or `False`. As a result, it leaves the equation in the symbolic form `x^2 + 3 x == 2`.

You can manipulate symbolic equations in *Mathematica* in many ways. One common goal is to rearrange the equations so as to "solve" for a particular set of variables.

Here is a symbolic equation.

```
In[1]:= x^2 + 3 x == 2
```

$$\text{Out[1]= } 3\,x + x^2 == 2$$

You can use the function `Reduce` to reduce the equation so as to give "solutions" for `x`. The result, like the original equation, can be viewed as a logical statement.

```
In[2]:= Reduce[%, x]
```

$$\text{Out[2]= } x == \frac{1}{2}\left(-3 - \sqrt{17}\right) \;||\; x == \frac{1}{2}\left(-3 + \sqrt{17}\right)$$

The quadratic equation `x^2 + 3 x == 2` can be thought of as an implicit statement about the value of `x`. As shown in the example above, you can use the function `Reduce` to get a more explicit statement about the value of `x`. The expression produced by `Reduce` has the form `x ==`r_1 `|| x ==`r_2. This expression is again a logical statement, which asserts that either `x` is equal to r_1, or `x` is equal to r_2. The values of `x` that are consistent with this statement are exactly the same as the ones that are consistent with the original quadratic equation. For many purposes, however, the form that `Reduce` gives is much more useful than the original equation.

You can combine and manipulate equations just like other logical statements. You can use logical connectives such as `||` and `&&` to specify alternative or simultaneous conditions. You can use functions like `LogicalExpand`, as well as `FullSimplify`, to simplify collections of equations.

For many purposes, you will find it convenient to manipulate equations simply as logical statements. Sometimes, however, you will actually want to use explicit solutions to equations in other calculations. In such cases, it is convenient to convert equations that are stated in the form *lhs* `==` *rhs* into transformation rules of the form *lhs* `->` *rhs*. Once you have the solutions to an equation in the form of explicit transformation rules, you can substitute the solutions into expressions by using the `/.` operator.

`Reduce` produces a logical statement about the values of `x` corresponding to the roots of the quadratic equation.

```
In[3]:= Reduce[x^2 + 3 x == 2, x]
```

$$\text{Out[3]= } x == \frac{1}{2}\left(-3 - \sqrt{17}\right) \;||\; x == \frac{1}{2}\left(-3 + \sqrt{17}\right)$$

`ToRules` converts the logical statement into an explicit list of transformation rules.

```
In[4]:= {ToRules[%]}
```

$$\text{Out[4]}= \left\{\left\{x \to \frac{1}{2}\left(-3-\sqrt{17}\right)\right\}, \left\{x \to \frac{1}{2}\left(-3+\sqrt{17}\right)\right\}\right\}$$

You can now use the transformation rules to substitute the solutions for `x` into expressions involving `x`.

```
In[5]:= x^2 + a x /. %
```

$$\text{Out[5]}= \left\{\frac{1}{4}\left(-3-\sqrt{17}\right)^2 + \frac{1}{2}\left(-3-\sqrt{17}\right) a, \frac{1}{4}\left(-3+\sqrt{17}\right)^2 + \frac{1}{2}\left(-3+\sqrt{17}\right) a\right\}$$

The function `Solve` produces transformation rules for solutions directly.

```
In[6]:= Solve[x^2 + 3 x == 2, x]
```

$$\text{Out[6]}= \left\{\left\{x \to \frac{1}{2}\left(-3-\sqrt{17}\right)\right\}, \left\{x \to \frac{1}{2}\left(-3+\sqrt{17}\right)\right\}\right\}$$

Equations in One Variable

The main equations that `Solve` and related *Mathematica* functions deal with are *polynomial equations*.

It is easy to solve a linear equation in `x`.

```
In[1]:= Solve[a x + b == c, x]
```

$$\text{Out[1]}= \left\{\left\{x \to \frac{-b+c}{a}\right\}\right\}$$

One can also solve quadratic equations just by applying a simple formula.

```
In[2]:= Solve[x^2 + a x + 2 == 0, x]
```

$$\text{Out[2]}= \left\{\left\{x \to \frac{1}{2}\left(-a-\sqrt{-8+a^2}\right)\right\}, \left\{x \to \frac{1}{2}\left(-a+\sqrt{-8+a^2}\right)\right\}\right\}$$

Mathematica can also find exact solutions to cubic equations. Here is the first solution to a comparatively simple cubic equation.

```
In[3]:= Solve[x^3 + 34 x + 1 == 0, x][[1]]
```

$$\text{Out[3]}= \left\{x \to -34\left(\frac{2}{3\left(-9+\sqrt{471\,729}\right)}\right)^{1/3} + \frac{\left(\frac{1}{2}\left(-9+\sqrt{471\,729}\right)\right)^{1/3}}{3^{2/3}}\right\}$$

For cubic and quartic equations the results are often complicated, but for all equations with degrees up to four *Mathematica* is always able to give explicit formulas for the solutions.

An important feature of these formulas is that they involve only *radicals*: arithmetic combinations of square roots, cube roots and higher roots.

It is a fundamental mathematical fact, however, that for equations of degree five or higher, it is no longer possible in general to give explicit formulas for solutions in terms of radicals.

There are some specific equations for which this is still possible, but in the vast majority of cases it is not.

This constructs a degree six polynomial.

```
In[4]:= Expand[Product[x^2 - 2 i, {i, 3}]]
```

$$\text{Out[4]=}\quad -48 + 44\,x^2 - 12\,x^4 + x^6$$

For a polynomial that factors in the way this one does, it is straightforward for `Solve` to find the roots.

```
In[5]:= Solve[% == 0, x]
```

$$\text{Out[5]=}\quad \left\{\{x \to -2\}, \{x \to 2\}, \left\{x \to -\sqrt{2}\right\}, \left\{x \to \sqrt{2}\right\}, \left\{x \to -\sqrt{6}\right\}, \left\{x \to \sqrt{6}\right\}\right\}$$

This constructs a polynomial of degree eight.

```
In[6]:= Expand[x^2 - 2 /. x -> x^2 - 3 /. x -> x^2 - 5]
```

$$\text{Out[6]=}\quad 482 - 440\,x^2 + 144\,x^4 - 20\,x^6 + x^8$$

The polynomial does not factor, but it can be decomposed into nested polynomials, so `Solve` can again find explicit formulas for the roots.

```
In[7]:= Solve[% == 0, x]
```

$$\text{Out[7]=}\quad \left\{\left\{x \to -\sqrt{5 - \sqrt{3 - \sqrt{2}}}\right\}, \left\{x \to \sqrt{5 - \sqrt{3 - \sqrt{2}}}\right\}, \left\{x \to -\sqrt{5 + \sqrt{3 - \sqrt{2}}}\right\}, \left\{x \to \sqrt{5 + \sqrt{3 - \sqrt{2}}}\right\},\right.$$
$$\left.\left\{x \to -\sqrt{5 - \sqrt{3 + \sqrt{2}}}\right\}, \left\{x \to \sqrt{5 - \sqrt{3 + \sqrt{2}}}\right\}, \left\{x \to -\sqrt{5 + \sqrt{3 + \sqrt{2}}}\right\}, \left\{x \to \sqrt{5 + \sqrt{3 + \sqrt{2}}}\right\}\right\}$$

`Root[`f`,`k`]`	the k^{th} root of the equation $f[x]$ == 0

Implicit representation for roots.

No explicit formulas for the solution to this equation can be given in terms of radicals, so *Mathematica* uses an implicit symbolic representation.

```
In[8]:= Solve[x^5 - x + 11 == 0, x]

Out[8]= {{x → Root[11 - #1 + #1^5 &, 1]}, {x → Root[11 - #1 + #1^5 &, 2]},
  {x → Root[11 - #1 + #1^5 &, 3]}, {x → Root[11 - #1 + #1^5 &, 4]}, {x → Root[11 - #1 + #1^5 &, 5]}}
```

This finds a numerical approximation to each root.

```
In[9]:= N[%]

Out[9]= {{x → -1.66149}, {x → -0.46194 - 1.565 i},
  {x → -0.46194 + 1.565 i}, {x → 1.29268 - 0.903032 i}, {x → 1.29268 + 0.903032 i}}
```

If what you want in the end is a numerical solution, it is usually much faster to use `NSolve` from the outset.

```
In[10]:= NSolve[x^5 - x + 11 == 0, x]

Out[10]= {{x → -1.66149}, {x → -0.46194 - 1.565 i},
  {x → -0.46194 + 1.565 i}, {x → 1.29268 - 0.903032 i}, {x → 1.29268 + 0.903032 i}}
```

`Root` objects provide an exact, though implicit, representation for the roots of a polynomial. You can work with them much as you would work with `Sqrt[2]` or any other expression that represents an exact numerical quantity.

Here is the `Root` object representing the first root of the polynomial discussed above.

```
In[11]:= r = Root[#^5 - # + 11 &, 1]

Out[11]= Root[11 - #1 + #1^5 &, 1]
```

This is a numerical approximation to its value.

```
In[12]:= N[r]

Out[12]= -1.66149
```

`Round` does an exact computation to find the closest integer to the root.

```
In[13]:= Round[r]

Out[13]= -2
```

If you substitute the root into the original polynomial, and then simplify the result, you get zero.

```
In[14]:= FullSimplify[x^5 - x + 11 /. x -> r]

Out[14]= 0
```

This finds the product of all the roots of the original polynomial.

```
In[15]:= FullSimplify[Product[Root[11 - # + #^5 &, k], {k, 5}]]
Out[15]= -11
```

The complex conjugate of the third root is the second root.

```
In[16]:= Conjugate[Root[11 - # + #^5 &, 3]]
Out[16]= Root[11 - #1 + #1^5 &, 2]
```

If the only symbolic parameter that exists in an equation is the variable that you are solving for, then all the solutions to the equation will just be numbers. But if there are other symbolic parameters in the equation, then the solutions will typically be functions of these parameters.

The solution to this equation can again be represented by `Root` objects, but now each `Root` object involves the parameter `a`.

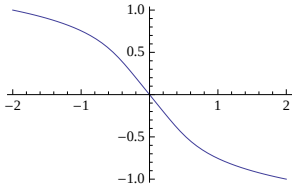
```
In[17]:= Solve[x^5 + x + a == 0, x]
Out[17]= {{x → Root[a + #1 + #1^5 &, 1]}, {x → Root[a + #1 + #1^5 &, 2]},
  {x → Root[a + #1 + #1^5 &, 3]}, {x → Root[a + #1 + #1^5 &, 4]}, {x → Root[a + #1 + #1^5 &, 5]}}
```

When `a` is replaced with 1, the `Root` objects can be simplified, and some are given as explicit radicals.

```
In[18]:= Simplify[% /. a -> 1]
Out[18]= {{x → Root[1 - #1^2 + #1^3 &, 1]}, {x → -1/2 i (-i + Sqrt[3])},
  {x → 1/2 i (i + Sqrt[3])}, {x → Root[1 - #1^2 + #1^3 &, 2]}, {x → Root[1 - #1^2 + #1^3 &, 3]}}
```

This shows the behavior of the first root as a function of `a`.

```
In[19]:= Plot[Root[#^5 + # + a &, 1], {a, -2, 2}]
```



This finds the derivative of the first root with respect to `a`.

```
In[20]:= D[Root[#^5 + # + a &, 1], a]
Out[20]= -1/(1 + 5 Root[a + #1 + #1^5 &, 1]^4)
```

If you give Solve any n^{th}-degree polynomial equation, then it will always return exactly n solutions, although some of these may be represented by Root objects. If there are degenerate solutions, then the number of times that each particular solution appears will be equal to its multiplicity.

Solve gives two identical solutions to this equation.

```
In[21]:= Solve[(x - 1) ^ 2 == 0, x]
Out[21]= {{x → 1}, {x → 1}}
```

Here are the first four solutions to a tenth-degree equation. The solutions come in pairs.

```
In[22]:= Take[Solve[(x^5 - x + 11) ^ 2 == 0, x], 4]
Out[22]= {{x → Root[11 - #1 + #1^5 &, 1]}, {x → Root[11 - #1 + #1^5 &, 1]},
  {x → Root[11 - #1 + #1^5 &, 2]}, {x → Root[11 - #1 + #1^5 &, 2]}}
```

Mathematica also knows how to solve equations which are not explicitly in the form of polynomials.

Here is an equation involving square roots.

```
In[23]:= Solve[Sqrt[x] + Sqrt[1 + x] == a, x]
Out[23]= {{x → (1 - 2 a^2 + a^4)/(4 a^2)}}
```

And here is one involving logarithms.

```
In[24]:= Solve[Log[x] + Log[1 - x] == a, x]
Out[24]= {{x → 1/2 (1 - Sqrt[1 - 4 E^a])}, {x → 1/2 (1 + Sqrt[1 - 4 E^a])}}
```

So long as it can reduce an equation to some kind of polynomial form, *Mathematica* will always be able to represent its solution in terms of Root objects. However, with more general equations, involving say transcendental functions, there is no systematic way to use Root objects, or even necessarily to find numerical approximations.

Here is a simple transcendental equation for x.

```
In[25]:= Solve[ArcSin[x] == a, x]
Out[25]= {{x → Sin[a]}}
```

There is no solution to this equation in terms of standard functions.

```
In[26]:= Solve[Cos[x] == x, x]

Solve::tdep :
  The equations appear to involve the variables to be solved for in an essentially non-algebraic way.

Out[26]= Solve[Cos[x] == x, x]
```

Mathematica can nevertheless find a numerical solution even in this case.

```
In[27]:= FindRoot[Cos[x] == x, {x, 0}]

Out[27]= {x → 0.739085}
```

Polynomial equations in one variable only ever have a finite number of solutions. But transcendental equations often have an infinite number. Typically the reason for this is that functions like `Sin` in effect have infinitely many possible inverses. With the default option setting `InverseFunctions -> True`, `Solve` will nevertheless assume that there is a definite inverse for any such function. `Solve` may then be able to return particular solutions in terms of this inverse function.

Mathematica returns a particular solution in terms of `ArcSin`, but prints a warning indicating that other solutions are lost.

```
In[28]:= Solve[Sin[x] == a, x]

Solve::ifun : Inverse functions are being used by Solve, so some
    solutions may not be found; use Reduce for complete solution information.

Out[28]= {{x → ArcSin[a]}}
```

Here the answer comes out in terms of `ProductLog`.

```
In[29]:= Solve[Exp[x] + x + 1 == 0, x]

InverseFunction::ifun : Inverse functions are being used. Values may be lost for multivalued inverses.

Solve::ifun : Inverse functions are being used by Solve, so some
    solutions may not be found; use Reduce for complete solution information.

Out[29]= {{x → -1 - ProductLog[1/E]}}
```

If you ask `Solve` to solve an equation involving an arbitrary function like `f`, it will by default try to construct a formal solution in terms of inverse functions.

`Solve` by default uses a formal inverse for the function `f`.

```
In[30]:= Solve[f[x] == a, x]
```

```
InverseFunction::ifun: Inverse functions are being used. Values may be lost for multivalued inverses.
```

```
Out[30]= {{x → f^(-1)[a]}}
```

This is the structure of the inverse function.

```
In[31]:= InputForm[%]
```

```
Out[31]//InputForm= {{x -> InverseFunction[f, 1, 1][a]}}
```

`InverseFunction[`f`]`	the inverse function of f
`InverseFunction[`f`,`k`,`n`]`	the inverse function of the n-argument function f with respect to its k^{th} argument

Inverse functions.

This returns an explicit inverse function.

```
In[32]:= InverseFunction[Tan]
```

```
Out[32]= ArcTan
```

Mathematica can do formal operations on inverse functions.

```
In[33]:= D[InverseFunction[f][x^2], x]
```

$$\text{Out[33]}= \frac{2\,x}{f'\left[f^{(-1)}\left[x^2\right]\right]}$$

While `Solve` can only give specific solutions to an equation, `Reduce` can give a representation of a whole solution set. For transcendental equations, it often ends up introducing new parameters, say with values ranging over all possible integers.

This is a complete representation of the solution set.

```
In[34]:= Reduce[Sin[x] == a, x]
```

```
Out[34]= C[1] ∈ Integers && (x == π - ArcSin[a] + 2 π C[1] || x == ArcSin[a] + 2 π C[1])
```

Here again is a representation of the general solution.

```
In[35]:= Reduce[Exp[x] + x + 1 == 0, x]
Out[35]= C[1] ∈ Integers && x == -1 - ProductLog[C[1], 1/e]
```

As discussed at more length in "Equations and Inequalities over Domains", `Reduce` allows you to restrict the domains of variables. Sometimes this will let you generate definite solutions to transcendental equations—or show that they do not exist.

With the domain of `x` restricted, this yields definite solutions.

```
In[36]:= Reduce[{Sin[x] == 1 / 2, Abs[x] < 4}, x]
Out[36]= x == -7 π/6 || x == π/6 || x == 5 π/6
```

With `x` constrained to be real, only one solution is possible.

```
In[37]:= Reduce[Exp[x] + x + 1 == 0, x, Reals]
Out[37]= x == -1 - ProductLog[1/e]
```

`Reduce` knows there can be no solution here.

```
In[38]:= Reduce[{Sin[x] == x, x > 1}, x]
Out[38]= False
```

Counting and Isolating Polynomial Roots

Counting Roots of Polynomials

`CountRoots[`*poly*`,`*x*`]`	give the number of real roots of the polynomial *poly* in *x*
`CountRoots[`*poly*`,{`*x*`,`*a*`,`*b*`}]`	give the number of roots of the polynomial *poly* in *x* with $\mathrm{Re}(a) \le \mathrm{Re}(r) \le \mathrm{Re}(b) \bigwedge \mathrm{Im}(a) \le \mathrm{Im}(r) \le \mathrm{Im}(b)$

Counting roots of polynomials.

`CountRoots` accepts polynomials with Gaussian rational coefficients. The root count includes multiplicities.

This gives the number of real roots of $(x^2-2)(x^2-3)(x^2-4)$.

```
In[1]:= CountRoots[(x^2 - 2) (x^2 - 3) (x^2 - 4), x]
Out[1]= 6
```

This counts the roots of $(x^2-2)(x^2-3)(x^2-4)$ in the closed interval $[1, 2]$.

```
In[2]:= CountRoots[(x^2 - 2) (x^2 - 3) (x^2 - 4), {x, 1, 2}]
Out[2]= 3
```

The roots of $(x^2+1)x^3$ in the vertical axis segment between 0 and $2i$ consist of a triple root at 0 and a single root at i.

```
In[3]:= CountRoots[(x^2 + 1) x^3, {x, 0, 2 i}]
Out[3]= 4
```

This counts 17^{th}-degree roots of unity in the closed unit square.

```
In[4]:= CountRoots[x^17 - 1, {x, 0, 1 + i}]
Out[4]= 5
```

The coefficients of the polynomial can be Gaussian rationals.

```
In[5]:= CountRoots[x^3 - i x + 3 i/4 - 1, {x, 0, 1 + i}]
Out[5]= 1
```

Isolating Intervals

A set $S \subseteq K$, where K is $\mathbb{R}$ or $\mathbb{C}$, is an *isolating set* for a root a of a polynomial f if a is the only root of f in S. Isolating roots of a polynomial means finding disjoint isolating sets for all the roots of the polynomial.

`RootIntervals[{`$poly_1$`,`$poly_2$`,...}]`	give a list of disjoint isolating intervals for the real roots of any of the $poly_i$, together with a list of which polynomials actually have each successive root
`RootIntervals[`$poly$`]`	give disjoint isolating intervals for real roots of a single polynomial

`RootIntervals[`*polys*`,Complexes]`	give disjoint isolating intervals or rectangles for complex roots of *polys*
`IsolatingInterval[`*a*`]`	give an isolating interval for the algebraic number *a*
`IsolatingInterval[`*a*`,`*dx*`]`	give an isolating interval of width at most *dx*

Functions for isolating roots of polynomials.

`RootIntervals` accepts polynomials with rational number coefficients.

For a real root r the returned isolating interval is a pair of rational numbers $\{a, b\}$, such that either $a < r < b$ or $a = b = r$. For a nonreal root r the isolating rectangle returned is a pair of Gaussian rational numbers $\{a, b\}$, such that $\text{Re}(a) < \text{Re}(r) < \text{Re}(b) \bigwedge \text{Im}(a) < \text{Im}(r) < \text{Im}(b)$ and either $\text{Im}(a) \geq 0$ or $\text{Im}(b) \leq 0$.

Here are isolating intervals for the real roots of f.

```
In[6]:= f = (x^2 - 2) (x^2 - 3) (x^2 - 4); RootIntervals[f]
```

$$\text{Out[6]}= \left\{\left\{\{-2, -2\}, \left\{-2, -\frac{3}{2}\right\}, \left\{-\frac{3}{2}, -1\right\}, \left\{1, \frac{3}{2}\right\}, \left\{\frac{3}{2}, 2\right\}, \{2, 2\}\right\}, \{\{1\}, \{1\}, \{1\}, \{1\}, \{1\}, \{1\}\}\right\}$$

The second list shows which interval contains a root of which polynomial.

```
In[7]:= RootIntervals[{f + 3, f + 5, f + 7}]
```

$$\text{Out[7]}= \left\{\left\{\left\{-\frac{5}{4}, -\frac{9}{8}\right\}, \left\{-\frac{9}{8}, -1\right\}, \{-1, 0\}, \{0, 1\}, \left\{1, \frac{9}{8}\right\}, \left\{\frac{9}{8}, \frac{5}{4}\right\}\right\}, \{\{1\}, \{2\}, \{3\}, \{3\}, \{2\}, \{1\}\}\right\}$$

This gives isolating intervals for all complex roots of $f + 3$.

```
In[8]:= RootIntervals[f + 3, Complexes]
```

$$\text{Out[8]}= \left\{\left\{\{-2, -1\}, \{1, 2\}, \left\{-7 - 7\,i, -\frac{7}{4}\right\}, \left\{-7, -\frac{7}{4} + 7\,i\right\}, \left\{-\frac{7}{4} - 7\,i, \frac{7}{2}\right\}, \left\{-\frac{7}{4}, \frac{7}{2} + 7\,i\right\}\right\}, \{\{1\}, \{1\}, \{1\}, \{1\}, \{1\}, \{1\}\}\right\}$$

Here are isolating intervals for the third- and fourth-degree roots of unity. The second interval contains a root common to both polynomials.

```
In[9]:= RootIntervals[{x^3 - 1, x^4 - 1}, Complexes]
```

$$\text{Out[9]}= \left\{\left\{\{-1, -1\}, \{0, 2\}, \left\{-\frac{3}{4} - \frac{3\,i}{2}, -\frac{3}{16} - \frac{3\,i}{4}\right\}, \left\{-\frac{3}{4} + \frac{3\,i}{4}, -\frac{3}{16} + \frac{3\,i}{2}\right\}, \left\{-\frac{3}{16} - \frac{3\,i}{2}, \frac{3}{8} - \frac{3\,i}{4}\right\}, \left\{-\frac{3}{16} + \frac{3\,i}{4}, \frac{3}{8} + \frac{3\,i}{2}\right\}\right\}, \{\{2\}, \{1, 2\}, \{1\}, \{1\}, \{2\}, \{2\}\}\right\}$$

Here is an isolating interval for a root of a polynomial of degree seven.

```
In[10]:= IsolatingInterval[Root[#^7 - 11 # + 3 &, 5]]
Out[10]= {-4, 4 i}
```

This gives an isolating interval of width at most 10^{-10}.

```
In[11]:= IsolatingInterval[Root[#^7 - 11 # + 3 &, 5], 10^-10]
```

$$\text{Out[11]}= \left\{-\frac{866\,877\,392\,461}{1\,099\,511\,627\,776}+\frac{355\,978\,878\,543\,i}{274\,877\,906\,944},\ -\frac{3\,467\,509\,569\,843}{4\,398\,046\,511\,104}+\frac{5\,695\,662\,056\,689\,i}{4\,398\,046\,511\,104}\right\}$$

All numbers in the interval have the first ten decimal digits in common.

```
In[12]:= N[%, 10]
Out[12]= {-0.788420396 + 1.295043616 i, -0.788420396 + 1.295043616 i}
```

Algebraic Numbers

Root [f,k]	the k^{th} root of the polynomial equation $f[x]$ == 0

The representation of algebraic numbers.

When you enter a Root object, the polynomial that appears in it is automatically reduced to a minimal form.

```
In[1]:= Root[24 - 2 # + 4 #^5 &, 1]
Out[1]= Root[12 - #1 + 2 #1^5 &, 1]
```

This extracts the pure function which represents the polynomial, and applies it to x.

```
In[2]:= First[%][x]
Out[2]= 12 - x + 2 x^5
```

Root objects are the way that *Mathematica* represents *algebraic numbers*. Algebraic numbers have the property that when you perform algebraic operations on them, you always get a single algebraic number as the result.

Here is the square root of an algebraic number.

```
In[3]:= Sqrt[Root[2 - # + #^5 &, 1]]
```

$$\text{Out[3]}= \sqrt{\text{Root}\left[2 - \#1 + \#1^5\ \&,\ 1\right]}$$

RootReduce reduces this to a single Root object.

```
In[4]:= RootReduce[%]
Out[4]= Root[2 - #1^2 + #1^10 &, 6]
```

Here is a more complicated expression involving an algebraic number.

```
In[5]:= Sqrt[2] + Root[2 - # + #^5 &, 1]^2
Out[5]= Sqrt[2] + Root[2 - #1 + #1^5 &, 1]^2
```

Again this can be reduced to a single Root object, albeit a fairly complicated one.

```
In[6]:= RootReduce[%]
Out[6]= Root[14 - 72 #1 + 25 #1^2 - 144 #1^3 - 88 #1^4 - 8 #1^5 + 62 #1^6 - 14 #1^8 + #1^10 &, 2]
```

RootReduce[*expr*]	attempt to reduce *expr* to a single Root object
ToRadicals[*expr*]	attempt to transform Root objects to explicit radicals

Operations on algebraic numbers.

In this simple case the Root object is automatically expressed in terms of radicals.

```
In[7]:= Root[#^2 - # - 1 &, 1]
Out[7]= 1/2 (1 - Sqrt[5])
```

When cubic polynomials are involved, Root objects are not automatically expressed in terms of radicals.

```
In[8]:= Root[#^3 - 2 &, 1]
Out[8]= Root[-2 + #1^3 &, 1]
```

ToRadicals attempts to express all Root objects in terms of radicals.

```
In[9]:= ToRadicals[%]
Out[9]= 2^(1/3)
```

If Solve and ToRadicals do not succeed in expressing the solution to a particular polynomial equation in terms of radicals, then it is a good guess that this fundamentally cannot be done. However, you should realize that there are some special cases in which a reduction to radicals is in principle possible, but *Mathematica* cannot find it. The simplest example is the equation

$x^5 + 20x + 32 = 0$, but here the solution in terms of radicals is very complicated. The equation $x^6 - 9x^4 - 4x^3 + 27x^2 - 36x - 23$ is another example, where now $x = 2^{\frac{1}{3}} + 3^{\frac{1}{2}}$ is a solution.

This gives a `Root` object involving a degree-six polynomial.

```
In[10]:= RootReduce[2^(1 / 3) + Sqrt[3]]
Out[10]= Root[-23 - 36 #1 + 27 #1^2 - 4 #1^3 - 9 #1^4 + #1^6 &, 2]
```

Even though a simple form in terms of radicals does exist, `ToRadicals` does not find it.

```
In[11]:= ToRadicals[%]
Out[11]= Root[-23 - 36 #1 + 27 #1^2 - 4 #1^3 - 9 #1^4 + #1^6 &, 2]
```

Beyond degree four, most polynomials do not have roots that can be expressed at all in terms of radicals. However, for degree five it turns out that the roots can always be expressed in terms of elliptic or hypergeometric functions. The results, however, are typically much too complicated to be useful in practice.

`RootSum[`*f*`,` *form*`]`	the sum of *form*`[`*x*`]` for all *x* satisfying the polynomial equation *f*`[`*x*`] == 0`
`Normal[`*expr*`]`	the form of *expr* with `RootSum` replaced by explicit sums of `Root` objects

Sums of roots.

This computes the sum of the reciprocals of the roots of $1 + 2x + x^5$.

```
In[12]:= RootSum[(1 + 2 # + #^5) &, (1 / #) &]
Out[12]= -2
```

Now no explicit result can be given in terms of radicals.

```
In[13]:= RootSum[(1 + 2 # + #^5) &, (# Log[1 + #]) &]
Out[13]= RootSum[1 + 2 #1 + #1^5 &, Log[1 + #1] #1 &]
```

This expands the `RootSum` into an explicit sum involving `Root` objects.

```
In[14]:= Normal[%]
Out[14]= Log[1 + Root[1 + 2 #1 + #1^5 &, 1]] Root[1 + 2 #1 + #1^5 &, 1] +
          Log[1 + Root[1 + 2 #1 + #1^5 &, 2]] Root[1 + 2 #1 + #1^5 &, 2] +
          Log[1 + Root[1 + 2 #1 + #1^5 &, 3]] Root[1 + 2 #1 + #1^5 &, 3] +
          Log[1 + Root[1 + 2 #1 + #1^5 &, 4]] Root[1 + 2 #1 + #1^5 &, 4] +
          Log[1 + Root[1 + 2 #1 + #1^5 &, 5]] Root[1 + 2 #1 + #1^5 &, 5]
```

RootApproximant[x]	converts the number x to one of the "simplest" algebraic numbers that approximates it well
RootApproximant[x,n]	finds an algebraic number of degree at most n that approximates x

This recovers $\sqrt{2}$ from a numerical approximation.

```
In[15]:= RootApproximant[N[Sqrt[2]]]
```

Out[15]= $\sqrt{2}$

In this case, the result has degree at most 4.

```
In[16]:= RootApproximant[N[Sqrt[2] + Sqrt[3]], 4]
```

```
Out[16]= Root[1 - 10 #1^2 + #1^4 &, 4]
```

This confirms that the Root expression does correspond to $\sqrt{2} + \sqrt{3}$.

```
In[17]:= RootReduce[Sqrt[2] + Sqrt[3]]
```

```
Out[17]= Root[1 - 10 #1^2 + #1^4 &, 4]
```

Simultaneous Equations

You can give Solve a list of simultaneous equations to solve. Solve can find explicit solutions for a large class of simultaneous polynomial equations.

Here is a simple linear equation with two unknowns.

```
In[1]:= Solve[{a x + b y == 1, x - y == 2}, {x, y}]
```

Out[1]= $\left\{\left\{x \to -\frac{-1-2\,b}{a+b},\ y \to -\frac{-1+2\,a}{a+b}\right\}\right\}$

Here is a more complicated example. The result is a list of solutions, with each solution consisting of a list of transformation rules for the variables.

```
In[2]:= Solve[{x^2 + y^2 == 1, x + y == a}, {x, y}]
```

Out[2]= $\left\{\left\{x \to \frac{1}{2}\left(a - \sqrt{2 - a^2}\right),\ y \to \frac{1}{2}\left(a + \sqrt{2 - a^2}\right)\right\},\ \left\{x \to \frac{a}{2} + \frac{\sqrt{2 - a^2}}{2},\ y \to \frac{1}{2}\left(a - \sqrt{2 - a^2}\right)\right\}\right\}$

You can use the list of solutions with the `/.` operator.

```
In[3]:= x^3 + y^4 /. % /. a -> 0.7
Out[3]= {0.846577, 0.901873}
```

Even when `Solve` cannot find explicit solutions, it often can "unwind" simultaneous equations to produce a symbolic result in terms of `Root` objects.

```
In[4]:= First[Solve[{x^2 + y^3 == x y, x + y + x y == 1}, {x, y}]]
```

$$\text{Out[4]= } \left\{x \to \frac{1}{2}\left(1 - \text{Root}\left[1 - 3\,\#1 + \#1^2 + 2\,\#1^3 + 2\,\#1^4 + \#1^5\,\&,\, 1\right]^2 - \text{Root}\left[1 - 3\,\#1 + \#1^2 + 2\,\#1^3 + 2\,\#1^4 + \#1^5\,\&,\, 1\right]^3 - \text{Root}\left[1 - 3\,\#1 + \#1^2 + 2\,\#1^3 + 2\,\#1^4 + \#1^5\,\&,\, 1\right]^4\right),\ y \to \text{Root}\left[1 - 3\,\#1 + \#1^2 + 2\,\#1^3 + 2\,\#1^4 + \#1^5\,\&,\, 1\right]\right\}$$

You can then use `N` to get a numerical result.

```
In[5]:= N[%]
Out[5]= {x → -3.4875, y → -1.80402}
```

The variables that you use in `Solve` do not need to be single symbols. Often when you set up large collections of simultaneous equations, you will want to use expressions like $a[i]$ as variables.

Here is a list of three equations for the `a[i]`.

```
In[6]:= Table[2 a[i] + a[i - 1] == a[i + 1], {i, 3}]
Out[6]= {a[0] + 2 a[1] == a[2], a[1] + 2 a[2] == a[3], a[2] + 2 a[3] == a[4]}
```

This solves for some of the `a[i]`.

```
In[7]:= Solve[%, {a[1], a[2], a[3]}]
```

$$\text{Out[7]= } \left\{\left\{a[1] \to \frac{1}{12}\,(-5\,a[0] + a[4]),\ a[2] \to \frac{1}{6}\,(a[0] + a[4]),\ a[3] \to \frac{1}{12}\,(-a[0] + 5\,a[4])\right\}\right\}$$

`Solve[`*eqns*`,{`x_1`,`x_2`,...}]`	solve *eqns* for the specific objects x_i
`Solve[`*eqns*`]`	try to solve *eqns* for all the objects that appear in them

Solving simultaneous equations.

If you do not explicitly specify objects to solve for, `Solve` will try to solve for all the variables.

```
In[8]:= Solve[{x + y == 1, x - 3 y == 2}]
```

$$\text{Out[8]= } \left\{\left\{x \to \frac{5}{4},\ y \to -\frac{1}{4}\right\}\right\}$$

- Solve[{lhs_1==rhs_1, lhs_2==rhs_2, ...}, $vars$]
- Solve[lhs_1==rhs_1&&lhs_2==rhs_2&&..., $vars$]
- Solve[{lhs_1, lhs_2, ...}=={rhs_1, rhs_2, ...}, $vars$]

Ways to present simultaneous equations to Solve.

If you construct simultaneous equations from matrices, you typically get equations between lists of expressions.

```
In[9]:= {{3, 1}, {2, -5}}.{x, y} == {7, 8}
Out[9]= {3 x + y, 2 x - 5 y} == {7, 8}
```

Solve converts equations involving lists to lists of equations.

```
In[10]:= Solve[%, {x, y}]
Out[10]= {{x → 43/17, y → -10/17}}
```

You can use LogicalExpand to do the conversion explicitly.

```
In[11]:= LogicalExpand[%%]
Out[11]= 2 x - 5 y == 8 && 3 x + y == 7
```

In some kinds of computations, it is convenient to work with arrays of coefficients instead of explicit equations. You can construct such arrays from equations by using CoefficientArrays.

Generic and Non-Generic Solutions

If you have an equation like 2 x == 0, it is perfectly clear that the only possible solution is x -> 0. However, if you have an equation like a x == 0, things are not so clear. If a is not equal to zero, then x -> 0 is again the only solution. However, if a is in fact equal to zero, then *any* value of x is a solution. You can see this by using Reduce.

Solve implicitly assumes that the parameter a does not have the special value 0.

```
In[1]:= Solve[a x == 0, x]
Out[1]= {{x → 0}}
```

Reduce, on the other hand, gives you all the possibilities, without assuming anything about the value of a.

```
In[2]:= Reduce[a x == 0, x]
Out[2]= a == 0 || x == 0
```

A basic difference between Reduce and Solve is that Reduce gives *all* the possible solutions to a set of equations, while Solve gives only the *generic* ones. Solutions are considered "generic" if they involve conditions only on the variables that you explicitly solve for, and not on other parameters in the equations. Reduce and Solve also differ in that Reduce always returns combinations of equations, while Solve gives results in the form of transformation rules.

Solve[*eqns*, *vars*]	find generic solutions to equations
Reduce[*eqns*, *vars*]	reduce equations, maintaining all solutions

Solving equations.

This is the solution to an arbitrary linear equation given by Solve.

```
In[3]:= Solve[a x + b == 0, x]
Out[3]= {{x → -b/a}}
```

Reduce gives the full version, which includes the possibility a == b == 0. In reading the output, note that && has higher precedence than ||.

```
In[4]:= Reduce[a x + b == 0, x]
Out[4]= (b == 0 && a == 0) || (a ≠ 0 && x == -b/a)
```

Here is the full solution to a general quadratic equation. There are three alternatives. If a is nonzero, then there are two solutions for x, given by the standard quadratic formula. If a is zero, however, the equation reduces to a linear one. Finally, if a, b and c are all zero, there is no restriction on x.

```
In[5]:= Reduce[a x^2 + b x + c == 0, x]
Out[5]= (a ≠ 0 && (x == (-b - Sqrt[b^2 - 4 a c])/(2 a) || x == (-b + Sqrt[b^2 - 4 a c])/(2 a))) ||
         (a == 0 && b ≠ 0 && x == -c/b) || (c == 0 && b == 0 && a == 0)
```

When you have several simultaneous equations, Reduce can show you under what conditions the equations have solutions. Solve shows you whether there are any generic solutions.

This shows there can never be any solution to these equations.

```
In[6]:= Reduce[{x == 1, x == 2}, x]
Out[6]= False
```

There is a solution to these equations, but only when a has the special value 1.

```
In[7]:= Reduce[{x == 1, x == a}, x]
Out[7]= a == 1 && x == 1
```

The solution is not generic, and is rejected by Solve.

```
In[8]:= Solve[{x == 1, x == a}, x]
Out[8]= {}
```

But if a is constrained to have value 1, then Solve again returns a solution.

```
In[9]:= Solve[{x == 1, x == a, a == 1}, x]
Out[9]= {{x → 1}}
```

This equation is true for any value of x.

```
In[10]:= Reduce[x == x, x]
Out[10]= True
```

This is the kind of result Solve returns when you give an equation that is always true.

```
In[11]:= Solve[x == x, x]
Out[11]= {{}}
```

When you work with systems of linear equations, you can use Solve to get generic solutions, and Reduce to find out for what values of parameters solutions exist.

Here is a matrix whose i, j^{th} element is $i + j$.

```
In[12]:= m = Table[i + j, {i, 3}, {j, 3}]
Out[12]= {{2, 3, 4}, {3, 4, 5}, {4, 5, 6}}
```

The matrix has determinant zero.

```
In[13]:= Det[m]
Out[13]= 0
```

This makes a set of three simultaneous equations.

```
In[14]:= eqn = m.{x, y, z} == {a, b, c}
Out[14]= {2 x + 3 y + 4 z, 3 x + 4 y + 5 z, 4 x + 5 y + 6 z} == {a, b, c}
```

`Solve` reports that there are no generic solutions.

```
In[15]:= Solve[eqn, {x, y, z}]
Out[15]= {}
```

`Reduce`, however, shows that there *would* be a solution if the parameters satisfied the special condition a == 2 b – c.

```
In[16]:= Reduce[eqn, {x, y, z}]
Out[16]= a == 2 b - c && y == -6 b + 5 c - 2 x && z == 5 b - 4 c + x
```

For nonlinear equations, the conditions for the existence of solutions can be much more complicated.

Here is a very simple pair of nonlinear equations.

```
In[17]:= eqn = {x y == a, x^2 y^2 == b}
Out[17]= {x y == a, x^2 y^2 == b}
```

`Solve` shows that the equations have no generic solutions.

```
In[18]:= Solve[eqn, {x, y}]
Out[18]= {}
```

`Reduce` gives the complete conditions for a solution to exist.

```
In[19]:= Reduce[eqn, {x, y}]
Out[19]= (b == 0 && a == 0 && x == 0) || ((a == -Sqrt[b] || a == Sqrt[b]) && x ≠ 0 && y == a/x)
```

Eliminating Variables

When you write down a set of simultaneous equations in *Mathematica*, you are specifying a collection of constraints between variables. When you use `Solve`, you are finding values for some of the variables in terms of others, subject to the constraints represented by the equations.

`Solve[`*eqns*`,`*vars*`,`*elims*`]`	find solutions for *vars*, eliminating the variables *elims*
`Eliminate[`*eqns*`,`*elims*`]`	rearrange equations to eliminate the variables *elims*

Eliminating variables.

Here are two equations involving `x`, `y` and the "parameters" `a` and `b`.

```
In[1]:= eqn = {x + y == 6 a + 3 b, y == 9 a + 2 x}
Out[1]= {x + y == 6 a + 3 b, y == 9 a + 2 x}
```

If you solve for both `x` and `y`, you get results in terms of `a` and `b`.

```
In[2]:= Solve[eqn, {x, y}]
Out[2]= {{x → -a + b, y → 7 a + 2 b}}
```

Similarly, if you solve for `x` and `a`, you get results in terms of `y` and `b`.

```
In[3]:= Solve[eqn, {x, a}]
Out[3]= {{x → 1/7 (9 b - y), a → 1/7 (-2 b + y)}}
```

If you only want to solve for `x`, however, you have to specify whether you want to eliminate `y` or `a` or `b`. This eliminates `y`, and so gives the result in terms of `a` and `b`.

```
In[4]:= Solve[eqn, x, y]
Out[4]= {{x → -a + b}}
```

If you eliminate `a`, then you get a result in terms of `y` and `b`.

```
In[5]:= Solve[eqn, x, a]
Out[5]= {{x → 1/7 (9 b - y)}}
```

In some cases, you may want to construct explicitly equations in which variables have been eliminated. You can do this using `Eliminate`.

This combines the two equations in the list `eqn`, by eliminating the variable `a`.

```
In[6]:= Eliminate[eqn, a]
Out[6]= 9 b - y == 7 x
```

This is what you get if you eliminate `y` instead of `a`.

```
In[7]:= Eliminate[eqn, y]
Out[7]= b - x == a
```

As a more sophisticated example of `Eliminate`, consider the problem of writing $x^5 + y^5$ in terms of the "symmetric polynomials" $x + y$ and $x\,y$.

To solve the problem, we simply have to write `f` in terms of `a` and `b`, eliminating the original variables `x` and `y`.

```
In[8]:= Eliminate[{f == x^5 + y^5, a == x + y, b == x y}, {x, y}]
Out[8]= f == a^5 - 5 a^3 b + 5 a b^2
```

In dealing with sets of equations, it is common to consider some of the objects that appear as true "variables", and others as "parameters". In some cases, you may need to know for what values of parameters a particular relation between the variables is *always* satisfied.

`SolveAlways[`*eqns*`,`*vars*`]`	solve for the values of parameters for which the *eqns* are satisfied for all values of the *vars*

Solving for parameters that make relations always true.

This finds the values of parameters that make the equation hold for all `x`.

```
In[9]:= SolveAlways[a + b x + c x^2 == (1 + x)^2, x]
Out[9]= {{a → 1, b → 2, c → 1}}
```

This equates two series.

```
In[10]:= Series[a Cos[x] + b Cos[2 x] + Cos[3 x], {x, 0, 3}] == Series[Cosh[x], {x, 0, 3}]
```

$$\text{Out[10]= } (1 + a + b) + \left(-\frac{9}{2} - \frac{a}{2} - 2\,b\right) x^2 + O[x]^4 == 1 + \frac{x^2}{2} + O[x]^4$$

This finds values of the undetermined coefficients.

```
In[11]:= SolveAlways[%, x]
```

$$\text{Out[11]}= \left\{\left\{a \to \frac{10}{3},\ b \to -\frac{10}{3}\right\}\right\}$$

Relational and Logical Operators

$x==y$	equal (also input as $x \equiv y$)
$x!=y$	unequal (also input as $x \neq y$)
$x>y$	greater than
$x>=y$	greater than or equal to (also input as $x \geq y$)
$x<y$	less than
$x<=y$	less than or equal to (also input as $x \leq y$)
$x==y==z$	all equal
$x!=y!=z$	all unequal (distinct)
$x>y>z$, etc.	strictly decreasing, etc.

Relational operators.

This tests whether `10` is less than `7`. The result is `False`.

```
In[1]:= 10 < 7
Out[1]= False
```

Not all of these numbers are unequal, so this gives `False`.

```
In[2]:= 3 != 2 != 3
Out[2]= False
```

You can mix `<` and `<=`.

```
In[3]:= 3 < 5 <= 6
Out[3]= True
```

Since both of the quantities involved are numeric, *Mathematica* can determine that this is true.

```
In[4]:= Pi^E < E^Pi
Out[4]= True
```

Mathematica does not know whether this is true or false.

```
In[5]:= x > y
Out[5]= x > y
```

`!`p	not (also input as $\neg p$)
p`&&`q`&&`...	and (also input as $p \wedge q \wedge \ldots$)
p`||`q`||`...	or (also input as $p \vee q \vee \ldots$)
`Xor[`p`,`q`,`...`]`	exclusive or (also input as $p \veebar q \veebar \ldots$)
`Nand[`p`,`q`,`...`]` and `Nor[`p`,`q`,`...`]`	nand and nor (also input as $\barwedge$ and $\barvee$)
`If[`p`,`*then*`,`*else*`]`	give *then* if p is `True`, and *else* if p is `False`
`LogicalExpand[`*expr*`]`	expand out logical expressions

Logical operations.

Both tests give `True`, so the result is `True`.

```
In[6]:= 7 > 4 && 2 != 3
Out[6]= True
```

You should remember that the logical operations `==`, `&&` and `||` are all *double characters* in *Mathematica*. If you have used a programming language such as C, you will be familiar with this notation.

Mathematica does not know whether this is true or false.

```
In[7]:= p && q
Out[7]= p && q
```

Mathematica leaves this expression unchanged.

```
In[8]:= (p || q) && ! (r || s)
Out[8]= (p || q) && ! (r || s)
```

You can use `LogicalExpand` to expand out the terms.

```
In[9]:= LogicalExpand[%]
Out[9]= (p && ! r && ! s) || (q && ! r && ! s)
```

Solving Logical Combinations of Equations

When you give a list of equations to `Solve`, it assumes that you want all the equations to be satisfied simultaneously. It is also possible to give `Solve` more complicated logical combinations of equations.

`Solve` assumes that the equations `x + y == 1` and `x - y == 2` are simultaneously valid.

```
In[1]:= Solve[{x + y == 1, x - y == 2}, {x, y}]
```

$$\text{Out[1]= } \left\{\left\{x \to \frac{3}{2},\ y \to -\frac{1}{2}\right\}\right\}$$

Here is an alternative form, using the logical connective `&&` explicitly.

```
In[2]:= Solve[x + y == 1 && x - y == 2, {x, y}]
```

$$\text{Out[2]= } \left\{\left\{x \to \frac{3}{2},\ y \to -\frac{1}{2}\right\}\right\}$$

This specifies that *either* `x + y == 1` *or* `x - y == 2`. `Solve` gives two solutions for `x`, corresponding to these two possibilities.

```
In[3]:= Solve[x + y == 1 || x - y == 2, x]
Out[3]= {{x → 1 - y}, {x → 2 + y}}
```

`Solve` gives three solutions to this equation.

```
In[4]:= Solve[x^3 == x, x]
Out[4]= {{x → -1}, {x → 0}, {x → 1}}
```

If you explicitly include the assertion that `x != 0`, one of the previous solutions is suppressed.

```
In[5]:= Solve[x^3 == x && x != 0, x]
Out[5]= {{x → -1}, {x → 1}}
```

Here is a slightly more complicated example. Note that the precedence of `||` is lower than the precedence of `&&`, so the equation is interpreted as `(x^3 == x && x != 1) || x^2 == 2`, not `x^3 == x && (x != 1 || x^2 == 2)`.

```
In[6]:= Solve[x^3 == x && x != 1 || x^2 == 2, x]
```

$$\text{Out[6]= } \left\{\{x \to -1\},\ \{x \to 0\},\ \left\{x \to -\sqrt{2}\right\},\ \left\{x \to \sqrt{2}\right\}\right\}$$

When you use `Solve`, the final results you get are in the form of transformation rules. If you use `Reduce` or `Eliminate`, on the other hand, then your results are logical statements, which you can manipulate further.

This gives a logical statement representing the solutions of the equation `x^2 == x`.

```
In[7]:= Reduce[x^2 == x, x]
Out[7]= x == 0 || x == 1
```

This finds values of `x` which satisfy `x^5 == x` but do not satisfy the statement representing the solutions of `x^2 == x`.

```
In[8]:= Reduce[x^5 == x && !%, x]
Out[8]= x == -1 || x == -i || x == i
```

The logical statements produced by `Reduce` can be thought of as representations of the solution set for your equations. The logical connectives `&&`, `||` and so on then correspond to operations on these sets.

$eqns_1$ `||` $eqns_2$	union of solution sets
$eqns_1$ `&&` $eqns_2$	intersection of solution sets
`!` $eqns$	complement of a solution set
`Implies[`$eqns_1$`,`$eqns_2$`]`	the part of $eqns_1$ that contains $eqns_2$

Operations on solution sets.

You may often find it convenient to use special notations for logical connectives, as discussed in "Operators".

The input uses special notations for `Implies` and `Or`.

```
In[9]:= Reduce[x^2 == 1 ⇒ (x == 1 ⋁ x == -1), x]
Out[9]= True
```

Inequalities

Just as the *equation* `x^2 + 3 x == 2` asserts that `x^2 + 3 x` is equal to `2`, so also the *inequality* `x^2 + 3 x > 2` asserts that `x^2 + 3 x` is greater than `2`. In *Mathematica*, `Reduce` works not only on equations, but also on inequalities.

Reduce[{$ineq_1$, $ineq_2$, …}, x]	reduce a collection of inequalities in x

Manipulating univariate inequalities.

This pair of inequalities reduces to a single inequality.

```
In[1]:= Reduce[{0 < x < 2, 1 < x < 4}, x]
Out[1]= 1 < x < 2
```

These inequalities can never simultaneously be satisfied.

```
In[2]:= Reduce[{x < 1, x > 3}, x]
Out[2]= False
```

When applied to an equation, Reduce[eqn, x] tries to get a result consisting of simple equations for x of the form $x == r_1$, … . When applied to an inequality, Reduce[$ineq$, x] does the exactly analogous thing, and tries to get a result consisting of simple inequalities for x of the form $l_1 < x < r_1$, … .

This reduces a quadratic equation to two simple equations for x.

```
In[3]:= Reduce[x^2 + 3 x == 2, x]
Out[3]= x == 1/2 (-3 - Sqrt[17]) || x == 1/2 (-3 + Sqrt[17])
```

This reduces a quadratic inequality to two simple inequalities for x.

```
In[4]:= Reduce[x^2 + 3 x > 2, x]
Out[4]= x < 1/2 (-3 - Sqrt[17]) || x > 1/2 (-3 + Sqrt[17])
```

You can think of the result generated by Reduce[$ineq$, x] as representing a series of intervals, described by inequalities. Since the graph of a polynomial of degree n can go up and down as many as n times, a polynomial inequality of degree n can give rise to as many as $n/2 + 1$ distinct intervals.

This inequality yields three distinct intervals.

```
In[5]:= Reduce[(x - 1) (x - 2) (x - 3) (x - 4) > 0, x]
Out[5]= x < 1 || 2 < x < 3 || x > 4
```

The ends of the intervals are at roots and poles.

```
In[6]:= Reduce[1 < (x^2 + 3 x) / (x + 1) < 2, x]
```

$$\text{Out[6]= } -1-\sqrt{2}<x<-2 \;||\; -1+\sqrt{2}<x<1$$

Solving this inequality requires introducing `ProductLog`.

```
In[7]:= Reduce[x - 2 < Log[x] < x, x]
```

```
Reduce::ztest: Unable to decide whether numeric quantities
{-2 - Log[-ProductLog[-Power[<<2>>]]] - ProductLog[-1/E^2], -2 - Log[-ProductLog[-1, -Power[<<2>>]
]] - ProductLog[-1, -1/E^2]} are equal to zero. Assuming they are.
```

$$\text{Out[7]= } -\text{ProductLog}\left[-\frac{1}{e^2}\right]<x<-\text{ProductLog}\left[-1,-\frac{1}{e^2}\right]$$

Transcendental functions like $\sin(x)$ have graphs that go up and down infinitely many times, so that infinitely many intervals can be generated.

The second inequality allows only finitely many intervals.

```
In[8]:= Reduce[{Sin[x] > 0, 0 < x < 20}, x]
```

$$\text{Out[8]= } 0<x<\pi \;||\; 2\pi<x<3\pi \;||\; 4\pi<x<5\pi \;||\; 6\pi<x<20$$

This is how `Reduce` represents infinitely many intervals.

```
In[9]:= Reduce[{Sin[x] > 0, 0 < x}, x]
```

$$\text{Out[9]= } C[1]\in\text{Integers}\;\&\&\;(0<x<\pi \;||\; (C[1]\geq 1\;\&\&\;2\pi C[1]<x<\pi+2\pi C[1]))$$

Fairly simple inputs can give fairly complicated results.

```
In[10]:= Reduce[{Sin[x]^2 + Sin[3 x] > 0, x^2 + 2 < 20}, x]
```

$$\text{Out[10]= } -3\sqrt{2}<x<-\pi \;||\; 2\,\text{ArcTan}\left[\frac{1}{3}\left(-4-\sqrt{7}\right)\right]<x<2\,\text{ArcTan}\left[\frac{1}{3}\left(-4+\sqrt{7}\right)\right] \;||$$
$$0<x<\frac{\pi}{2} \;||\; \frac{\pi}{2}<x<\pi \;||\; 2\pi+2\,\text{ArcTan}\left[\frac{1}{3}\left(-4-\sqrt{7}\right)\right]<x<3\sqrt{2}$$

If you have inequalities that involve `<=` as well as `<`, there may be isolated points where the inequalities can be satisfied. `Reduce` represents such points by giving equations.

This inequality can be satisfied at just two isolated points.

```
In[11]:= Reduce[(x^2 - 3 x + 1)^2 <= 0, x]
```

$$\text{Out[11]= } x == \frac{1}{2}\left(3 - \sqrt{5}\right) \;||\; x == \frac{1}{2}\left(3 + \sqrt{5}\right)$$

This yields both intervals and isolated points.

```
In[12]:= Reduce[{Max[Sin[2 x], Cos[3 x]] <= 0, 0 < x < 10}, x]
```

$$\text{Out[12]= } x == \frac{\pi}{2} \;||\; \frac{5\pi}{6} \le x \le \pi \;||\; \frac{3\pi}{2} \le x \le \frac{11\pi}{6} \;||\; x == \frac{5\pi}{2} \;||\; \frac{17\pi}{6} \le x \le 3\pi$$

Reduce[{$ineq_1$, $ineq_2$, ...}, {x_1, x_2, ...}]	reduce a collection of inequalities in several variables

Multivariate inequalities.

For inequalities involving several variables, Reduce in effect yields nested collections of interval specifications, in which later variables have bounds that depend on earlier variables.

This represents the unit disk as nested inequalities for x and y.

```
In[13]:= Reduce[x^2 + y^2 < 1, {x, y}]
```

$$\text{Out[13]= } -1 < x < 1 \;\&\&\; -\sqrt{1 - x^2} < y < \sqrt{1 - x^2}$$

In geometrical terms, any linear inequality divides space into two halves. Lists of linear inequalities thus define polyhedra, sometimes bounded, sometimes not. Reduce represents such polyhedra in terms of nested inequalities. The corners of the polyhedra always appear among the endpoints of these inequalities.

This defines a triangular region in the plane.

```
In[14]:= Reduce[{x > 0, y > 0, x + y < 1}, {x, y}]
```

$$\text{Out[14]= } 0 < x < 1 \;\&\&\; 0 < y < 1 - x$$

Even a single triangle may need to be described as two components.

```
In[15]:= Reduce[{x > y - 1, y > 0, x + y < 1}, {x, y}]
```

$$\text{Out[15]= } (-1 < x \le 0 \;\&\&\; 0 < y < 1 + x) \;||\; (0 < x < 1 \;\&\&\; 0 < y < 1 - x)$$

Lists of inequalities in general represent regions of overlap between geometrical objects. Often the description of these can be quite complicated.

This represents the part of the unit disk on one side of a line.

```
In[16]:= Reduce[{x^2 + y^2 < 1, x + 3 y > 2}, {x, y}]
```

$$\text{Out[16]}= \frac{1}{10}\left(2-3\sqrt{6}\right) < x < \frac{1}{10}\left(2+3\sqrt{6}\right) \;\&\&\; \frac{2-x}{3} < y < \sqrt{1-x^2}$$

Here is the intersection between two disks.

```
In[17]:= Reduce[{(x - 1)^2 + y^2 < 2, x^2 + y^2 < 2}, {x, y}]
```

$$\text{Out[17]}= \left(1-\sqrt{2} < x \le \frac{1}{2} \;\&\&\; -\sqrt{1+2x-x^2} < y < \sqrt{1+2x-x^2}\right) \;||\; \left(\frac{1}{2} < x < \sqrt{2} \;\&\&\; -\sqrt{2-x^2} < y < \sqrt{2-x^2}\right)$$

If the disks are too far apart, there is no intersection.

```
In[18]:= Reduce[{(x - 4)^2 + y^2 < 2, x^2 + y^2 < 2}, {x, y}]
Out[18]= False
```

Here is an example involving a transcendental inequality.

```
In[19]:= Reduce[{Sin[x y] > 1 / 2, x^2 + y^2 < 3 / 2}, {x, y}]
```

$$\text{Out[19]}= \left(-\sqrt{\frac{3}{4}+\frac{1}{12}\sqrt{81-4\pi^2}} < x < -\frac{1}{2}\sqrt{\frac{1}{3}\left(9-\sqrt{81-4\pi^2}\right)} \;\&\&\; -\frac{\sqrt{3-2x^2}}{\sqrt{2}} < y < \frac{\pi}{6x}\right) \;||$$
$$\left(\frac{1}{2}\sqrt{\frac{1}{3}\left(9-\sqrt{81-4\pi^2}\right)} < x < \sqrt{\frac{3}{4}+\frac{1}{12}\sqrt{81-4\pi^2}} \;\&\&\; \frac{\pi}{6x} < y < \frac{\sqrt{3-2x^2}}{\sqrt{2}}\right)$$

If you have inequalities that involve parameters, `Reduce` automatically handles the different cases that can occur, just as it does for equations.

The form of the intervals depends on the value of `a`.

```
In[20]:= Reduce[(x - 1) (x - a) > 0, x]
Out[20]= (a ≤ 1 && (x < a || x > 1)) || (a > 1 && (x < 1 || x > a))
```

One gets a hyperbolic or an elliptical region, depending on the value of a.

```
In[21]:= Reduce[x^2 + a y^2 < 1, {x, y}]
```

$$\text{Out[21]= } y \in \text{Reals} \;\&\&\; \left(\left(a < 0 \;\&\&\; \left(\left(x \le -1 \;\&\&\; \left(y < -\sqrt{\frac{1-x^2}{a}} \;||\; y > \sqrt{\frac{1-x^2}{a}}\right)\right) \;||\; -1 < x < 1 \;||\; \left(x \ge 1 \;\&\&\; \left(y < -\sqrt{\frac{1-x^2}{a}} \;||\; y > \sqrt{\frac{1-x^2}{a}}\right)\right)\right)\right) \;||\; (a == 0 \;\&\&\; -1 < x < 1) \;||\; \left(a > 0 \;\&\&\; -1 < x < 1 \;\&\&\; -\sqrt{\frac{1-x^2}{a}} < y < \sqrt{\frac{1-x^2}{a}}\right)\right)$$

`Reduce` tries to give you a complete description of the region defined by a set of inequalities. Sometimes, however, you may just want to find individual instances of values of variables that satisfy the inequalities. You can do this using `FindInstance`.

`FindInstance[`*ineqs*`,{`x_1`,`x_2`,...}]`	try to find an instance of the x_i satisfying *ineqs*
`FindInstance[`*ineqs*`,`*vars*`,`*n*`]`	try to find *n* instances

Finding individual points that satisfy inequalities.

This finds a specific instance that satisfies the inequalities.

```
In[22]:= FindInstance[{Sin[x y] > 1 / 2, x^2 + y^2 < 3 / 2}, {x, y}]
```

$$\text{Out[22]= } \left\{\left\{x \to -\frac{88}{151},\; y \to -\frac{543}{566}\right\}\right\}$$

This shows that there is no way to satisfy the inequalities.

```
In[23]:= FindInstance[{Sin[x y] > 1 / 2, x^2 + y^2 < 1 / 4}, {x, y}]
Out[23]= {}
```

`FindInstance` is in some ways an analog for inequalities of `Solve` for equations. For like `Solve`, it returns a list of rules giving specific values for variables. But while for equations these values can generically give an accurate representation of all solutions, for inequalities they can only correspond to isolated sample points within the regions described by the inequalities.

Every time you call `FindInstance` with specific input, it will give the same output. And when there are instances that correspond to special, limiting, points of some kind, it will preferentially return these. But in general, the distribution of instances returned by `FindInstance` will typi-

cally seem somewhat random. Each instance is, however, in effect a constructive proof that the inequalities you have given can in fact be satisfied.

If you ask for one point in the unit disk, `FindInstance` gives the origin.

```
In[24]:= FindInstance[x^2 + y^2 <= 1, {x, y}]
Out[24]= {{x → 0, y → 0}}
```

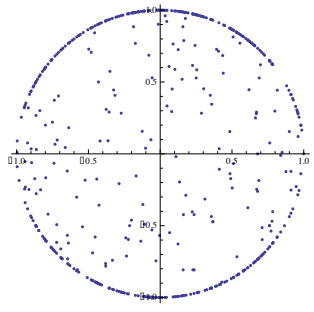
This finds 500 points in the unit disk.

```
In[25]:= FindInstance[x^2 + y^2 <= 1, {x, y}, 500];
```

Their distribution seems somewhat random.

```
In[26]:= ListPlot[{x, y} /. %, AspectRatio -> Automatic]
```

Out[26]=



Equations and Inequalities over Domains

Mathematica normally assumes that variables which appear in equations can stand for arbitrary complex numbers. But when you use `Reduce`, you can explicitly tell *Mathematica* that the variables stand for objects in more restricted domains.

`Reduce[`*expr*`,`*vars*`,`*dom*`]`	reduce *eqns* over the domain *dom*
`Complexes`	complex numbers $\mathbb{C}$
`Reals`	real numbers $\mathbb{R}$
`Integers`	integers $\mathbb{Z}$

Solving over domains.

`Reduce` by default assumes that `x` can be complex, and gives all five complex solutions.

```
In[1]:= Reduce[x^6 - x^4 - 4 x^2 + 4 == 0, x]
Out[1]= x == -1 || x == 1 || x == -√2 || x == -i √2 || x == i √2 || x == √2
```

But here it assumes that x is real, and gives only the real solutions.

```
In[2]:= Reduce[x^6 - x^4 - 4 x^2 + 4 == 0, x, Reals]
Out[2]= x == -1 || x == 1 || x == -Sqrt[2] || x == Sqrt[2]
```

And here it assumes that x is an integer, and gives only the integer solutions.

```
In[3]:= Reduce[x^6 - x^4 - 4 x^2 + 4 == 0, x, Integers]
Out[3]= x == -1 || x == 1
```

A single polynomial equation in one variable will always have a finite set of discrete solutions. And in such a case one can think of `Reduce[`*eqns*`,` *vars*`,` *dom*`]` as just filtering the solutions by selecting the ones that happen to lie in the domain *dom*.

But as soon as there are more variables, things can become more complicated, with solutions to equations corresponding to parametric curves or surfaces in which the values of some variables can depend on the values of others. Often this dependence can be described by some collection of equations or inequalities, but the form of these can change significantly when one goes from one domain to another.

This gives solutions over the complex numbers as simple formulas.

```
In[4]:= Reduce[x^2 + y^2 == 1, {x, y}]
Out[4]= y == -Sqrt[1 - x^2] || y == Sqrt[1 - x^2]
```

To represent solutions over the reals requires introducing an inequality.

```
In[5]:= Reduce[x^2 + y^2 == 1, {x, y}, Reals]
Out[5]= -1 ≤ x ≤ 1 && (y == -Sqrt[1 - x^2] || y == Sqrt[1 - x^2])
```

Over the integers, the solution can be represented as equations for discrete points.

```
In[6]:= Reduce[x^2 + y^2 == 1, {x, y}, Integers]
Out[6]= (x == -1 && y == 0) || (x == 0 && y == -1) || (x == 0 && y == 1) || (x == 1 && y == 0)
```

If your input involves only equations, then `Reduce` will by default assume that all variables are complex. But if your input involves inequalities, then `Reduce` will assume that any algebraic variables appearing in them are real, since inequalities can only compare real quantities.

Since the variables appear in an inequality, they are assumed to be real.

```
In[7]:= Reduce[{x + y + z == 1, x^2 + y^2 + z^2 < 1}, {x, y, z}]
```

$$\text{Out[7]}= -\frac{1}{3} < x < 1 \;\&\&\; \frac{1-x}{2} - \frac{1}{2}\sqrt{1+2x-3x^2} < y < \frac{1-x}{2} + \frac{1}{2}\sqrt{1+2x-3x^2} \;\&\&\; z == 1-x-y$$

`Complexes`	*polynomial* != 0, x_i == `Root[...]`
`Reals`	`Root[...]` < x_i < `Root[...]`, x_i == `Root[...]`
`Integers`	arbitrarily complicated

Schematic building blocks for solutions to polynomial equations and inequalities.

For systems of polynomials over real and complex domains, the solutions always consist of a finite number of components, within which the values of variables are given by algebraic numbers or functions.

Here the components are distinguished by equations and inequations on `x`.

```
In[8]:= Reduce[x y^3 + y == 1, {x, y}, Complexes]
```

```
Out[8]= (x == 0 && y == 1) ||
  (x ≠ 0 && (y == Root[-1 + #1 + x #1^3 &, 1] || y == Root[-1 + #1 + x #1^3 &, 2] || y == Root[-1 + #1 + x #1^3 &, 3]))
```

And here the components are distinguished by inequalities on `x`.

```
In[9]:= Reduce[x y^3 + y == 1, {x, y}, Reals]
```

```
Out[9]= (x < -4/27 && y == Root[-1 + #1 + x #1^3 &, 1]) || (x == -4/27 && (y == -3 || y == 3/2)) || (-4/27 < x < 0 &&
    (y == Root[-1 + #1 + x #1^3 &, 1] || y == Root[-1 + #1 + x #1^3 &, 2] || y == Root[-1 + #1 + x #1^3 &, 3])) ||
  (x ≥ 0 && y == Root[-1 + #1 + x #1^3 &, 1])
```

While in principle `Reduce` can always find the complete solution to any collection of polynomial equations and inequalities with real or complex variables, the results are often very complicated, with the number of components typically growing exponentially as the number of variables increases.

With 3 variables, the solution here already involves 8 components.

```
In[10]:= Reduce[x^2 == y^2 == z^2 == 1, {x, y, z}]
```

```
Out[10]= (z == -1 && y == -1 && x == -1) || (z == -1 && y == -1 && x == 1) ||
  (z == -1 && y == 1 && x == -1) || (z == -1 && y == 1 && x == 1) || (z == 1 && y == -1 && x == -1) ||
  (z == 1 && y == -1 && x == 1) || (z == 1 && y == 1 && x == -1) || (z == 1 && y == 1 && x == 1)
```

As soon as one introduces functions like `Sin` or `Exp`, even equations in single real or complex variables can have solutions with an infinite number of components. `Reduce` labels these components by introducing additional parameters. By default, the n^{th} parameter in a given solution will be named `C[`n`]`. In general you can specify that it should be named f`[`n`]` by giving the option setting `GeneratedParameters ->` f.

The components here are labeled by the integer parameter c_1.

```
In[11]:= Reduce[Exp[x] == 2, x, GeneratedParameters -> (Subscript[c, #] &)]
Out[11]= c_1 ∈ Integers && x == Log[2] + 2 i π c_1
```

`Reduce` can handle equations not only over real and complex variables, but also over integers. Solving such *Diophantine equations* can often be a very difficult problem.

Describing the solution to this equation over the reals is straightforward.

```
In[12]:= Reduce[x y == 8, {x, y}, Reals]
Out[12]= (x < 0 || x > 0) && y == 8/x
```

The solution over the integers involves the divisors of 8.

```
In[13]:= Reduce[x y == 8, {x, y}, Integers]
Out[13]= (x == -8 && y == -1) || (x == -4 && y == -2) || (x == -2 && y == -4) || (x == -1 && y == -8) ||
          (x == 1 && y == 8) || (x == 2 && y == 4) || (x == 4 && y == 2) || (x == 8 && y == 1)
```

Solving an equation like this effectively requires factoring a large number.

```
In[14]:= Reduce[{x y == 7 777 777, x > y > 0}, {x, y}, Integers]
Out[14]= (x == 4649 && y == 1673) || (x == 32 543 && y == 239) || (x == 1 111 111 && y == 7) || (x == 7 777 777 && y == 1)
```

`Reduce` can solve any system of linear equations or inequalities over the integers. With m linear equations in n variables, $n-m$ parameters typically need to be introduced. But with inequalities, a much larger number of parameters may be needed.

Three parameters are needed here, even though there are only two variables.

```
In[15]:= Reduce[{3 x - 2 y > 1, x > 0, y > 0}, {x, y}, Integers]
Out[15]= (C[1] | C[2] | C[3]) ∈ Integers && C[1] ≥ 0 && C[2] ≥ 0 &&
          C[3] ≥ 0 && ((x == 2 + 2 C[1] + C[2] + C[3] && y == 2 + 3 C[1] + C[2]) ||
            (x == 2 + 2 C[1] + C[2] + C[3] && y == 1 + 3 C[1] + C[2]))
```

With two variables, `Reduce` can solve any quadratic equation over the integers. The result can be a Fibonacci-like sequence, represented in terms of powers of quadratic irrationals.

Here is the solution to a Pell equation.

```
In[16]:= Reduce[{x^2 == 13 y^2 + 1, x > 0, y > 0}, {x, y}, Integers]
```

$$\text{Out[16]= } C[1] \in \text{Integers} \&\& C[1] \geq 1 \&\& x == \frac{1}{2}\left(\left(649 - 180\sqrt{13}\right)^{C[1]} + \left(649 + 180\sqrt{13}\right)^{C[1]}\right) \&\& y == -\frac{\left(649 - 180\sqrt{13}\right)^{C[1]} - \left(649 + 180\sqrt{13}\right)^{C[1]}}{2\sqrt{13}}$$

The actual values for specific `C[1]` as integers, as they should be.

```
In[17]:= FullSimplify[% /. Table[{C[1] -> i}, {i, 4}]]
Out[17]= {x == 649 && y == 180, x == 842 401 && y == 233 640,
  x == 1 093 435 849 && y == 303 264 540, x == 1 419 278 889 601 && y == 393 637 139 280}
```

`Reduce` can handle many specific classes of equations over the integers.

Here `Reduce` finds the solution to a Thue equation.

```
In[18]:= Reduce[x^3 - 4 x y^2 + y^3 == 1, {x, y}, Integers]
Out[18]= (x == -2 && y == 1) || (x == 0 && y == 1) || (x == 1 && y == 0) ||
  (x == 1 && y == 4) || (x == 2 && y == 1) || (x == 508 && y == 273)
```

Changing the right-hand side to `3`, the equation now has no solution.

```
In[19]:= Reduce[x^3 - 4 x y^2 + y^3 == 3, {x, y}, Integers]
Out[19]= False
```

Equations over the integers sometimes have seemingly quite random collections of solutions. And even small changes in equations can often lead them to have no solutions at all.

For polynomial equations over real and complex numbers, there is a definite *decision procedure* for determining whether or not any solution exists. But for polynomial equations over the integers, the unsolvability of Hilbert's tenth problem demonstrates that there can never be any such general procedure.

For specific classes of equations, however, procedures can be found, and indeed many are implemented in `Reduce`. But handling different classes of equations can often seem to require whole different branches of number theory, and quite different kinds of computations. And in fact it is known that there are *universal* integer polynomial equations, for which filling in some variables can make solutions for other variables correspond to the output of absolutely any possible program. This then means that for such equations there can never in general be any closed-form solution built from fixed elements like algebraic functions.

If one includes functions like `Sin`, then even for equations involving real and complex numbers the same issues can arise.

`Reduce` here effectively has to solve an equation over the integers.

```
In[20]:= Reduce[Sin[Pi x]^2 + Sin[Pi y]^2 + (x^2 + y^2 - 25)^2 == 0, {x, y}, Reals]
Out[20]= (x == -5 && y == 0) || (x == -4 && (y == -3 || y == 3)) || (x == -3 && (y == -4 || y == 4)) ||
  (x == 0 && (y == -5 || y == 5)) || (x == 3 && (y == -4 || y == 4)) || (x == 4 && (y == -3 || y == 3)) || (x == 5 && y == 0)
```

`Reduce[`*eqns*`,`*vars*`,Modulus->`*n*`]`	find solutions modulo *n*

Handling equations involving integers modulo *n*.

Since there are only ever a finite number of possible solutions for integer equations modulo *n*, `Reduce` can systematically find them.

This finds all solutions modulo 4.

```
In[21]:= Reduce[x^5 == y^4 + x y + 1, {x, y}, Modulus -> 4]
Out[21]= (x == 1 && y == 0) || (x == 1 && y == 3) || (x == 2 && y == 1) || (x == 2 && y == 3) || (x == 3 && y == 2) || (x == 3 && y == 3)
```

`Reduce` can also handle equations that involve several different moduli.

Here is an equation involving two different moduli.

```
In[22]:= Reduce[Mod[2 x + 1, 5] == Mod[x, 7] && 0 < x < 50, x]
Out[22]= x == 4 || x == 7 || x == 15 || x == 23 || x == 31 || x == 39 || x == 42
```

`Reduce[`*expr*`,`*vars*`,`*dom*`]`	specify a default domain for all variables
`Reduce[{`*expr*$_1$`,…,`*x*$_1$∈*dom*$_1$`,…},`*vars*`]`	explicitly specify individual domains for variables

Different ways to specify domains for variables.

This assumes that `x` is an integer, but `y` is a real.

```
In[23]:= Reduce[{x^2 + 2 y^2 == 1, x ∈ Integers, y ∈ Reals}, {x, y}]
Out[23]= (x == -1 && y == 0) || (x == 0 && (y == -1/Sqrt[2] || y == 1/Sqrt[2])) || (x == 1 && y == 0)
```

`Reduce` normally treats complex variables as single objects. But in dealing with functions that are not analytic or have branch cuts, it sometimes has to break them into pairs of real variables `Re[`*z*`]` and `Im[`*z*`]`.

The result involves separate real and imaginary parts.

```
In[24]:= Reduce[Abs[z] == 1, z]
Out[24]= -1 ≤ Re[z] ≤ 1 && (Im[z] == -√(1 - Re[z]^2) || Im[z] == √(1 - Re[z]^2))
```

Here again there is a separate condition on the imaginary part.

```
In[25]:= Reduce[Log[z] == a, {a, z}]
Out[25]= -π < Im[a] ≤ π && z == e^a
```

`Reduce` by default assumes that variables that appear algebraically in inequalities are real. But you can override this by explicitly specifying `Complexes` as the default domain. It is often useful in such cases to be able to specify that certain variables are still real.

`Reduce` by default assumes that `x` is a real.

```
In[26]:= Reduce[x^2 < 1, x]
Out[26]= -1 < x < 1
```

This forces `Reduce` to consider the case where `x` can be complex.

```
In[27]:= Reduce[x^2 < 1, x, Complexes]
Out[27]= (-1 < Re[x] < 0 && Im[x] == 0) || Re[x] == 0 || (0 < Re[x] < 1 && Im[x] == 0)
```

Since `x` does not appear algebraically, `Reduce` immediately assumes that it can be complex.

```
In[28]:= Reduce[Abs[x] < 1, x]
Out[28]= -1 < Re[x] < 1 && -√(1 - Re[x]^2) < Im[x] < √(1 - Re[x]^2)
```

Here `x` is a real, but `y` can be complex.

```
In[29]:= Reduce[{Abs[y] < Abs[x], x ∈ Reals}, {x, y}]
Out[29]= (x < 0 && -√x^2 < Re[y] < √x^2 && -√(x^2 - Re[y]^2) < Im[y] < √(x^2 - Re[y]^2)) ||
  (x > 0 && -√x^2 < Re[y] < √x^2 && -√(x^2 - Re[y]^2) < Im[y] < √(x^2 - Re[y]^2))
```

`FindInstance[`*expr*`,{`x_1`,`x_2`,...},`*dom*`]`	try to find an instance of the x_i in *dom* satisfying *expr*
`FindInstance[`*expr*`,`*vars*`,`*dom*`,`*n*`]`	try to find *n* instances
`Complexes`	the domain of complex numbers $\mathbb{C}$

`Reals`	the domain of real numbers $\mathbb{R}$
`Integers`	the domain of integers $\mathbb{Z}$
`Booleans`	the domain of Booleans (`True` and `False`) $\mathbb{B}$

Finding particular solutions in domains.

`Reduce` always returns a complete representation of the solution to a system of equations or inequalities. Sometimes, however, you may just want to find particular sample solutions. You can do this using `FindInstance`.

If `FindInstance[`*expr*`,` *vars*`,` *dom*`]` returns `{}` then this means that *Mathematica* has effectively proved that *expr* cannot be satisfied for any values of variables in the specified domain. When *expr* can be satisfied, `FindInstance` will normally pick quite arbitrarily among values that do this, as discussed for inequalities in "Inequalities: Manipulating Equations and Inequalities".

Particularly for integer equations, `FindInstance` can often find particular solutions to equations even when `Reduce` cannot find a complete solution. In such cases it usually returns one of the smallest solutions to the equations.

This finds the smallest integer point on an elliptic curve.

```
In[30]:= FindInstance[{x^2 == y^3 + 12, x > 0, y > 0}, {x, y}, Integers]
Out[30]= {{x → 47, y → 13}}
```

One feature of `FindInstance` is that it also works with Boolean expressions whose variables can have values `True` or `False`. You can use `FindInstance` to determine whether a particular expression is *satisfiable*, so that there is some choice of truth values for its variables that makes the expression `True`.

This expression cannot be satisfied for any choice of `p` and `q`.

```
In[31]:= FindInstance[p && ! (p || ! q), {p, q}, Booleans]
Out[31]= {}
```

But this can.

```
In[32]:= FindInstance[p && ! (! p || ! q), {p, q}, Booleans]
Out[32]= {{p → True, q → True}}
```

The Representation of Solution Sets

Any combination of equations or inequalities can be thought of as implicitly defining a region in some kind of space. The fundamental function of Reduce is to turn this type of implicit description into an explicit one.

An implicit description in terms of equations or inequalities is sufficient if one just wants to test whether a point specified by values of variables is in the region. But to understand the structure of the region, or to generate points in it, one typically needs a more explicit description, of the kind obtained from Reduce.

Here are inequalities that implicitly define a semicircular region.

```
In[1]:= semi = x > 0 && x^2 + y^2 < 1
```

$$\text{Out[1]= } x > 0 \;\&\&\; x^2 + y^2 < 1$$

This shows that the point $(1/2, 1/2)$ lies in the region.

```
In[2]:= semi /. {x -> 1 / 2, y -> 1 / 2}
```

```
Out[2]= True
```

Reduce gives a more explicit representation of the region.

```
In[3]:= Reduce[semi, {x, y}]
```

$$\text{Out[3]= } 0 < x < 1 \;\&\&\; -\sqrt{1 - x^2} < y < \sqrt{1 - x^2}$$

If we pick a value for x consistent with the first inequality, we then immediately get an explicit inequality for y.

```
In[4]:= % /. x -> 1 / 2
```

$$\text{Out[4]= } -\frac{\sqrt{3}}{2} < y < \frac{\sqrt{3}}{2}$$

Reduce[*expr*, {x_1, x_2, ...}] is set up to describe regions by first giving fixed conditions for x_1, then giving conditions for x_2 that depend on x_1, then conditions for x_3 that depend on x_1 and x_2, and so on. This structure has the feature that it allows one to pick points by successively choosing values for each of the x_i in turn—in much the same way as when one uses iterators in functions like Table.

This gives a representation for the region in which one first picks a value for `y`, then `x`.

```
In[5]:= Reduce[semi, {y, x}]
```

$$\text{Out[5]= } -1 < y < 1 \;\&\&\; 0 < x < \sqrt{1 - y^2}$$

In some simple cases the region defined by a system of equations or inequalities will end up having only one component. In such cases, the output from `Reduce` will be of the form e_1 `&&` e_2 … where each of the e_i is an equation or inequality involving variables up to x_i.

In most cases, however, there will be several components, represented by output containing forms such as u_1 `||` u_2 `||` …. `Reduce` typically tries to minimize the number of components used in describing a region. But in some cases multiple parametrizations may be needed to cover a single connected component, and each one of these will appear as a separate component in the output from `Reduce`.

In representing solution sets, it is common to find that several components can be described together by using forms such as … `&&` (u_1 `||` u_2) `&&` …. `Reduce` by default does this so as to return its results as compactly as possible. You can use `LogicalExpand` to generate an expanded form in which each component appears separately.

In generating the most compact results, `Reduce` sometimes ends up making conditions on later variables x_i depend on more of the earlier x_i than is strictly necessary. You can force `Reduce` to generate results in which a particular x_i only has minimal dependence on earlier x_i by giving the option `Backsubstitution -> True`. Usually this will lead to much larger output, although sometimes it may be easier to interpret.

By default, `Reduce` expresses the condition on `y` in terms of `x`.

```
In[6]:= Reduce[x^2 + y == 4 && x^3 - 4 y == 8, {x, y}]
```

$$\text{Out[6]= } \left(x == 2 \;||\; x == -3 - i\sqrt{3} \;||\; x == -3 + i\sqrt{3}\right) \;\&\&\; y == 4 - x^2$$

Backsubstituting allows conditions for `y` to be given without involving `x`.

```
In[7]:= Reduce[x^2 + y == 4 && x^3 - 4 y == 8, {x, y}, Backsubstitution -> True]
```

$$\text{Out[7]= } (x == 2 \;\&\&\; y == 0) \;||\; \left(x == -i\left(-3i + \sqrt{3}\right) \;\&\&\; y == -2i\left(-i + 3\sqrt{3}\right)\right) \;||\; \left(x == i\left(3i + \sqrt{3}\right) \;\&\&\; y == 2i\left(i + 3\sqrt{3}\right)\right)$$

`CylindricalDecomposition[`*expr*`,{`x_1`,`x_2`,…}]`	generate the cylindrical algebraic decomposition of the region defined by *expr*
`GenericCylindricalDecomposition[`*expr*`,{`x_1`,`x_2`,…}]`	find the full-dimensional part of the decomposition of the region defined by *expr*, together with any hypersurfaces containing the rest of the region
`SemialgebraicComponentInstances[`*expr*`,{`x_1`,`x_2`,…}]`	give at least one point in each connected component of the region defined by *expr*

Cylindrical algebraic decomposition.

For polynomial equations or inequalities over the reals, the structure of the result returned by `Reduce` is typically a *cylindrical algebraic decomposition* or *CAD*. Sometimes `Reduce` can yield a simpler form. But in all cases you can get the complete CAD by using `CylindricalDecomposition`. For systems containing inequalities only, `GenericCylindricalDecomposition` gives you "most" of the solution set and is often faster.

Here is the cylindrical algebraic decomposition of a region bounded by a hyperbola.

```
In[8]:= CylindricalDecomposition[x^2 - y^2 >= 1, {x, y}]
```

$$\text{Out[8]}= \left(x < -1 \&\& -\sqrt{-1+x^2} \le y \le \sqrt{-1+x^2}\right) \mid\mid (x == -1 \&\& y == 0) \mid\mid (x == 1 \&\& y == 0) \mid\mid \left(x > 1 \&\& -\sqrt{-1+x^2} \le y \le \sqrt{-1+x^2}\right)$$

This gives the two-dimensional part of the solution set along with a curve containing the boundary.

```
In[9]:= GenericCylindricalDecomposition[x^2 - y^2 >= 1, {x, y}]
```

$$\text{Out[9]}= \left\{\left(x < -1 \&\& -\sqrt{-1+x^2} < y < \sqrt{-1+x^2}\right) \mid\mid \left(x > 1 \&\& -\sqrt{-1+x^2} < y < \sqrt{-1+x^2}\right), 1 - x^2 + y^2 == 0\right\}$$

This finds solutions from both parts of the solution set.

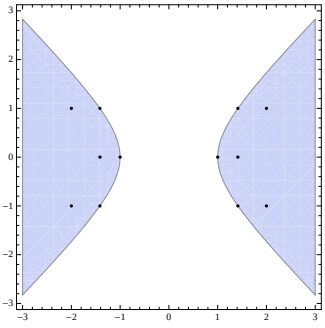
```
In[10]:= SemialgebraicComponentInstances[x^2 - y^2 >= 1, {x, y}]
```

$$\text{Out[10]}= \left\{\{x \to -2, y \to -1\}, \{x \to -2, y \to 1\}, \{x \to -1, y \to 0\}, \{x \to 1, y \to 0\}, \{x \to 2, y \to -1\}, \{x \to 2, y \to 1\}, \left\{x \to -\sqrt{2}, y \to -1\right\}, \left\{x \to -\sqrt{2}, y \to 0\right\}, \left\{x \to -\sqrt{2}, y \to 1\right\}, \left\{x \to \sqrt{2}, y \to -1\right\}, \left\{x \to \sqrt{2}, y \to 0\right\}, \left\{x \to \sqrt{2}, y \to 1\right\}\right\}$$

The results include a few points from each piece of the solution set.

```
In[11]:= Show[
  {RegionPlot[x^2 - y^2 >= 1, {x, -3, 3}, {y, -3, 3}], Graphics[Point[{x, y}] /. %]}]
```

Out[11]=



Quantifiers

In a statement like `x^4 + x^2 > 0`, *Mathematica* treats the variable `x` as having a definite, though unspecified, value. Sometimes, however, it is useful to be able to make statements about whole collections of possible values for `x`. You can do this using *quantifiers*.

`ForAll[`x`,`$expr$`]`	$expr$ holds for all values of x
`ForAll[{`x_1`,`x_2`,...},`$expr$`]`	$expr$ holds for all values of all the x_i
`ForAll[{`x_1`,`x_2`,...},`$cond$`,`$expr$`]`	$expr$ holds for all x_i satisfying $cond$
`Exists[`x`,`$expr$`]`	there exists a value of x for which $expr$ holds
`Exists[{`x_1`,`x_2`,...},`$expr$`]`	there exist values of the x_i for which $expr$ holds
`Exists[{`x_1`,...},`$cond$`,`$expr$`]`	there exist values of the x_i satisfying $cond$ for which $expr$ holds

The structure of quantifiers.

You can work with quantifiers in *Mathematica* much as you work with equations, inequalities or logical connectives. In most cases, the quantifiers will not immediately be changed by evaluation. But they can be simplified or reduced by functions like `FullSimplify` and `Reduce`.

This asserts that an `x` exists that makes the inequality true. The output here is just a formatted version of the input.

```
In[1]:= Exists[x, x^4 + x^2 > 0]
```

Out[1]= $\exists_x\, x^2 + x^4 > 0$

`FullSimplify` establishes that the assertion is true.

```
In[2]:= FullSimplify[%]
Out[2]= True
```

This gives `False`, since the inequality fails when `x` is zero.

```
In[3]:= FullSimplify[ForAll[x, x^4 + x^2 > 0]]
Out[3]= False
```

Mathematica supports a version of the standard notation for quantifiers used in predicate logic and pure mathematics. You can input ∀ as `\[ForAll]` or `:fa:`, and you can input ∃ as `\[Exists]` or `:ex:`. To make the notation precise, however, *Mathematica* makes the quantified variable a subscript. The conditions on the variable can also be given in the subscript, separated by a comma.

$\forall_x expr$	`ForAll[`x`,`$expr$`]`
$\forall_{\{x_1, x_2, \ldots\}} expr$	`ForAll[{`x_1`,`x_2`,...},`$expr$`]`
$\forall_{x, cond} expr$	`ForAll[`x`,`$cond$`,`$expr$`]`
$\exists_x expr$	`Exists[`x`,`$expr$`]`
$\exists_{\{x_1, x_2, \ldots\}} expr$	`Exists[{`x_1`,`x_2`,...},`$expr$`]`
$\exists_{x, cond} expr$	`Exists[`x`,`$cond$`,`$expr$`]`

Notation for quantifiers.

Given a statement that involves quantifiers, there are certain important cases where it is possible to resolve it into an equivalent statement in which the quantifiers have been eliminated. Somewhat like solving an equation, such quantifier elimination turns an implicit statement about what is true for all x or for some x into an explicit statement about the conditions under which this holds.

`Resolve[`$expr$`]`	attempt to eliminate quantifiers from $expr$
`Resolve[`$expr$`,`dom`]`	attempt to eliminate quantifiers with all variables assumed to be in domain dom

Quantifier elimination.

This shows that an x exists that makes the equation true.

```
In[4]:= Resolve[Exists[x, x^2 == x^3]]
Out[4]= True
```

This shows that the equations can only be satisfied if c obeys a certain condition.

```
In[5]:= Resolve[Exists[x, x^2 == c && x^3 == c + 1]]
Out[5]= -1 - 2 c - c^2 + c^3 == 0
```

`Resolve` can always eliminate quantifiers from any collection of polynomial equations and inequations over complex numbers, and from any collection of polynomial equations and inequalities over real numbers. It can also eliminate quantifiers from Boolean expressions.

This finds the conditions for a quadratic form over the reals to be positive.

```
In[6]:= Resolve[ForAll[x, a x^2 + b x + c > 0], Reals]
Out[6]= (a > 0 && -a b^2 + 4 a^2 c > 0) || (a == 0 && b == 0 && c > 0) || (a ≥ 0 && b == 0 && c > 0 && -a b^2 + 4 a^2 c > 0)
```

This shows that there is a way of assigning truth values to p and q that makes the expression true.

```
In[7]:= Resolve[Exists[{p, q}, p || q && ! q], Booleans]
Out[7]= True
```

You can also use quantifiers with `Reduce`. If you give `Reduce` a collection of equations or inequalities, then it will try to produce a detailed representation of the complete solution set. But sometimes you may want to address a more global question, such as whether the solution set covers all values of x, or whether it covers none of these values. Quantifiers provide a convenient way to specify such questions.

This gives the complete structure of the solution set.

```
In[8]:= Reduce[x^2 + x + c == 0, {c, x}, Reals]
```

$$\text{Out[8]}= \left(c < \frac{1}{4} \;\&\&\; \left(x == -\frac{1}{2} - \frac{1}{2}\sqrt{1 - 4\,c} \;||\; x == -\frac{1}{2} + \frac{1}{2}\sqrt{1 - 4\,c}\right)\right) \;||\; \left(c == \frac{1}{4} \;\&\&\; x == -\frac{1}{2}\right)$$

This instead just gives the condition for a solution to exist.

```
In[9]:= Reduce[Exists[x, x^2 + x + c == 0], {c}, Reals]
```

$$\text{Out[9]}= c \le \frac{1}{4}$$

It is possible to formulate a great many mathematical questions in terms of quantifiers.

This finds the conditions for a circle to be contained within an arbitrary conic section.

```
In[10]:= Reduce[ForAll[{x, y}, x^2 + y^2 < 1, a x^2 + b y^2 < c], {a, b, c}, Reals]
```

$$\text{Out[10]}= (a \le 0 \&\& ((b \le 0 \&\& c > 0) \,||\, (b > 0 \&\& c \ge b))) \,||\, (a > 0 \&\& ((b < a \&\& c \ge a) \,||\, (b \ge a \&\& c \ge b)))$$

This finds the conditions for a line to intersect a circle.

```
In[11]:= Reduce[Exists[{x, y}, x^2 + y^2 < 1, r x + s y == 1], {r, s}, Reals]
```

$$\text{Out[11]}= r < -1 \,||\, \left(-1 \le r \le 1 \&\& \left(s < -\sqrt{1 - r^2} \,||\, s > \sqrt{1 - r^2}\right)\right) \,||\, r > 1$$

This defines q to be a general monic quartic.

```
In[12]:= q[x_] := x^4 + b x^3 + c x^2 + d x + e
```

This finds the condition for all pairs of roots to the quartic to be equal.

```
In[13]:= Reduce[ForAll[{x, y}, q[x] == 0 && q[y] == 0, x == y], {b, c, d, e}]
```

$$\text{Out[13]}= \left(c == \frac{3\,b^2}{8} \&\& d == \frac{b^3}{16} \&\& e == \frac{b^4}{256}\right) \,||\, (b == 0 \&\& c == 0 \&\& d == 0 \&\& e == 0)$$

Although quantifier elimination over the integers is in general a computationally impossible problem, *Mathematica* can do it in specific cases.

This shows that $\sqrt{2}$ cannot be a rational number.

```
In[14]:= Resolve[Exists[{x, y}, x^2 == 2 y^2 && y > 0], Integers]
Out[14]= False
```

$\sqrt{9/4}$ is, though.

```
In[15]:= Resolve[Exists[{x, y}, 4 x^2 == 9 y^2 && y > 0], Integers]
Out[15]= True
```

Minimization and Maximization

`Minimize[`*expr*`,{`x_1`,`x_2`,...}]`	minimize *expr*
`Minimize[{`*expr*`,`*cons*`},{`x_1`,`x_2`,...}]`	minimize *expr* subject to the constraints *cons*
`Maximize[`*expr*`,{`x_1`,`x_2`,...}]`	maximize *expr*
`Maximize[{`*expr*`,`*cons*`},{`x_1`,`x_2`,...}]`	maximize *expr* subject to the constraints *cons*

Minimization and maximization.

`Minimize` and `Maximize` yield lists giving the value attained at the minimum or maximum, together with rules specifying where the minimum or maximum occurs.

This finds the minimum of a quadratic function.

```
In[1]:= Minimize[x^2 - 3 x + 6, x]
```

$$\text{Out[1]}= \left\{\frac{15}{4}, \left\{x \to \frac{3}{2}\right\}\right\}$$

Applying the rule for `x` gives the value at the minimum.

```
In[2]:= x^2 - 3 x + 6 /. Last[%]
```

$$\text{Out[2]}= \frac{15}{4}$$

This maximizes with respect to `x` and `y`.

```
In[3]:= Maximize[5 x y - x^4 - y^4, {x, y}]
```

$$\text{Out[3]}= \left\{\frac{25}{8}, \left\{x \to -\frac{\sqrt{5}}{2}, y \to -\frac{\sqrt{5}}{2}\right\}\right\}$$

`Minimize[`*expr*`, `x`]` minimizes *expr* allowing x to range over all possible values from $-\infty$ to $+\infty$. `Minimize[{`*expr*`, `*cons*`}, `x`]` minimizes *expr* subject to the constraints *cons* being satisfied. The constraints can consist of any combination of equations and inequalities.

This finds the minimum subject to the constraint $x \geq 3$.

```
In[4]:= Minimize[{x^2 - 3 x + 6, x >= 3}, x]
```

$$\text{Out[4]}= \{6, \{x \to 3\}\}$$

This finds the maximum within the unit circle.

```
In[5]:= Maximize[{5 x y - x^4 - y^4, x^2 + y^2 <= 1}, {x, y}]
```

$$\text{Out[5]}= \left\{2, \left\{x \to -\frac{1}{\sqrt{2}}, y \to -\frac{1}{\sqrt{2}}\right\}\right\}$$

This finds the maximum within an ellipse. The result is fairly complicated.

```
In[6]:= Maximize[{5 x y - x^4 - y^4, x^2 + 2 y^2 <= 1}, {x, y}]
```

$$\text{Out[6]}= \left\{-\text{Root}\left[-811\,219 + 320\,160\,\#1 + 274\,624\,\#1^2 - 170\,240\,\#1^3 + 25\,600\,\#1^4 \&,\ 1\right],\right.$$
$$\left\{x \to \text{Root}\left[25 - 102\,\#1^2 + 122\,\#1^4 - 70\,\#1^6 + 50\,\#1^8 \&,\ 2\right],\right.$$
$$\left.\left. y \to \text{Root}\left[25 - 264\,\#1^2 + 848\,\#1^4 - 1040\,\#1^6 + 800\,\#1^8 \&,\ 1\right]\right\}\right\}$$

This finds the maximum along a line.

```
In[7]:= Maximize[{5 x y - x^4 - y^4, x + y == 1}, {x, y}]
```

$$\text{Out[7]}= \left\{\frac{9}{8}, \left\{x \to \frac{1}{2}, y \to \frac{1}{2}\right\}\right\}$$

`Minimize` and `Maximize` can solve any *linear programming* problem in which both the objective function $expr$ and the constraints $cons$ involve the variables x_i only linearly.

Here is a typical linear programming problem.

```
In[8]:= Minimize[{x + 3 y, x - 3 y <= 7 && x + 2 y >= 10}, {x, y}]
```

$$\text{Out[8]}= \left\{\frac{53}{5}, \left\{x \to \frac{44}{5}, y \to \frac{3}{5}\right\}\right\}$$

They can also in principle solve any *polynomial programming* problem in which the objective function and the constraints involve arbitrary polynomial functions of the variables. There are many important geometrical and other problems that can be formulated in this way.

This solves the simple geometrical problem of maximizing the area of a rectangle with fixed perimeter.

```
In[9]:= Maximize[{x y, x + y == 1}, {x, y}]
```

$$\text{Out[9]}= \left\{\frac{1}{4}, \left\{x \to \frac{1}{2}, y \to \frac{1}{2}\right\}\right\}$$

This finds the maximal volume of a cuboid that fits inside the unit sphere.

```
In[10]:= Maximize[{8 x y z, x^2 + y^2 + z^2 <= 1}, {x, y, z}]
```

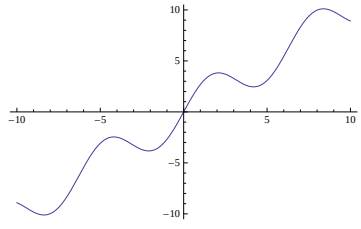
$$\text{Out[10]}= \left\{\frac{8}{3\sqrt{3}}, \left\{x \to \frac{1}{\sqrt{3}}, y \to -\frac{1}{\sqrt{3}}, z \to -\frac{1}{\sqrt{3}}\right\}\right\}$$

An important feature of `Minimize` and `Maximize` is that they always find *global* minima and maxima. Often functions will have various local minima and maxima at which derivatives vanish. But `Minimize` and `Maximize` use global methods to find absolute minima or maxima, not just local extrema.

Here is a function with many local maxima and minima.

```
In[11]:= Plot[x + 2 Sin[x], {x, -10, 10}]
```

Out[11]=



`Maximize` finds the global maximum.

```
In[12]:= Maximize[{x + 2 Sin[x], -10 <= x <= 10}, x]
```

$$\text{Out[12]=}\quad \left\{\sqrt{3} + \frac{8\pi}{3}, \left\{x \to \frac{8\pi}{3}\right\}\right\}$$

If you give functions that are unbounded, `Minimize` and `Maximize` will return $-\infty$ and $+\infty$ as the minima and maxima. And if you give constraints that can never be satisfied, they will return $+\infty$ and $-\infty$ as the minima and maxima, and `Indeterminate` as the values of variables.

One subtle issue is that `Minimize` and `Maximize` allow both *nonstrict* inequalities of the form $x <= v$, and *strict* ones of the form $x < v$. With nonstrict inequalities there is no problem with a minimum or maximum lying exactly on the boundary $x -> v$. But with strict inequalities, a minimum or maximum must in principle be at least infinitesimally inside the boundary.

With a strict inequality, *Mathematica* prints a warning, then returns the point on the boundary.

```
In[13]:= Minimize[{x^2 - 3 x + 6, x > 3}, x]

Minimize::wksol: Warning: There is no minimum in the
    region described by the contraints; returning a result on the boundary.

Out[13]= {6, {x → 3}}
```

`Minimize` and `Maximize` normally assume that all variables you give are real. But by giving a constraint such as $x \in$ `Integers` you can specify that a variable must in fact be an integer.

This does maximization only over integer values of x and y.

```
In[14]:= Maximize[{x y, x^2 + y^2 < 120 && (x | y) ∈ Integers}, {x, y}]
Out[14]= {56, {x → -8, y → -7}}
```

Minimize and Maximize can compute maxima and minima of linear functions over the integers in bounded polyhedra. This is known as integer linear programming.

This does maximization over integer values of x and y in a triangle.

```
In[15]:= Maximize[{5 + 3 y + 7 x, x >= 0 && y >= 0 && 3 x + 4 y <= 100 && (x | y) ∈ Integers}, {x, y}]
Out[15]= {236, {x → 33, y → 0}}
```

Minimize and Maximize can produce exact symbolic results for polynomial optimization problems with parameters.

This finds the minimum of a general quadratic polynomial.

```
In[16]:= Minimize[a x^2 + b x + c, x]
```

$$\left\{\begin{cases} c & (b == 0 \&\& a == 0) \,||\, (b == 0 \&\& a > 0) \\ \frac{-b^2+4ac}{4a} & (b > 0 \&\& a > 0) \,||\, (b < 0 \&\& a > 0) \\ -\infty & \text{True} \end{cases}, \left\{x \to \begin{cases} 0 & (b == 0 \&\& a == 0) \,||\, (b == 0 \&\& a > 0) \\ -\frac{b}{2a} & (b > 0 \&\& a > 0) \,||\, (b < 0 \&\& a > 0) \\ \text{Indeterminate} & \text{True} \end{cases}\right\}\right\}$$

MinValue[{f, $cons$}, {x, y, ...}]	give the minimum value of f subject to the constraints $cons$
MaxValue[{f, $cons$}, {x, y, ...}]	give the maximum value of f subject to the constraints $cons$
ArgMin[{f, $cons$}, {x, y, ...}]	give a position at which f is minimized subject to the constraints $cons$
ArgMax[{f, $cons$}, {x, y, ...}]	give a position at which f is maximized subject to the constraints $cons$

Computing values and positions of minima and maxima.

Maximize gives both the value and the position of a maximum.

```
In[17]:= Maximize[{x + 2 y, x^2 + y^2 ≤ 1}, {x, y}]
```

$$\left\{\sqrt{5}, \left\{x \to -\frac{4}{\sqrt{5}} + \sqrt{5}, y \to \frac{2}{\sqrt{5}}\right\}\right\}$$

Use `MaxValue` if you only need the maximum value.

```
In[18]:= MaxValue[{x + 2 y, x^2 + y^2 ≤ 1}, {x, y}]
```

$$\text{Out[18]}= \sqrt{5}$$

For strict polynomial inequality constraints computing only the maximum value may be much faster.

```
In[19]:= TimeConstrained[
           Maximize[{-x^2 + 2 x y - z - 1, x^2 y < z^3 && x - z^2 > y^2 + 2}, {x, y, z}], 300]
Out[19]= $Aborted
```

```
In[20]:= MaxValue[{-x^2 + 2 x y - z - 1, x^2 y < z^3 && x - z^2 > y^2 + 2}, {x, y, z}] // Timing
```

$$\text{Out[20]}= \{0.312,\ -1 - \text{Root}[-6\,674\,484\,057\,677\,824 + 27\,190\,416\,613\,703\,680\,\#1 - 9\,845\,871\,213\,297\,967\,104\,\#1^2 + 30\,310\,812\,947\,042\,320\,384\,\#1^3 - 38\,968\,344\,650\,849\,575\,680\,\#1^4 + 27\,943\,648\,095\,748\,511\,616\,\#1^5 - 13\,622\,697\,129\,083\,140\,957\,\#1^6 + 5\,905\,344\,357\,450\,294\,480\,\#1^7 - 2\,872\,859\,681\,127\,251\,424\,\#1^8 + 1\,484\,592\,492\,626\,145\,792\,\#1^9 - 567\,863\,224\,101\,551\,360\,\#1^{10} + 100\,879\,538\,475\,737\,088\,\#1^{11} + 303\,891\,741\,605\,888\,\#1^{12} - 2\,224\,545\,911\,472\,128\,\#1^{13} + 70\,301\,735\,976\,960\,\#1^{14} + 25\,686\,756\,556\,800\,\#1^{15} + 1\,786\,706\,395\,136\,\#1^{16} + 73\,014\,444\,032\,\#1^{17}\ \&,\ 1]\}$$

`ArgMax` gives a position at which the maximum value is attained.

```
In[21]:= ArgMax[{x + 2 y, x^2 + y^2 ≤ 1}, {x, y}]
```

$$\text{Out[21]}= \left\{-\frac{4}{\sqrt{5}} + \sqrt{5},\ \frac{2}{\sqrt{5}}\right\}$$

Linear Algebra

Constructing Matrices

`Table[`f`,{`i`,`m`},{`j`,`n`}]`	build an $m \times n$ matrix where f is a function of i and j that gives the value of the i, j^{th} entry
`Array[`f`,{`m`,`n`}]`	build an $m \times n$ matrix whose i, j^{th} entry is $f[i, j]$
`ConstantArray[`a`,{`m`,`n`}]`	build an $m \times n$ matrix with all entries equal to a
`DiagonalMatrix[`*list*`]`	generate a diagonal matrix with the elements of *list* on the diagonal
`IdentityMatrix[`n`]`	generate an $n \times n$ identity matrix
`Normal[SparseArray[{{`i_1`,`j_1`}->`v_1`,{`i_2`,`j_2`}->`v_2`,...},{`m`,`n`}]]`	make a matrix with nonzero values v_k at positions $\{i_k, j_k\}$

Functions for constructing matrices.

This generates a 2×2 matrix whose i, j^{th} entry is a[i, j].

```
In[1]:= Table[a[i, j], {i, 2}, {j, 2}]
Out[1]= {{a[1, 1], a[1, 2]}, {a[2, 1], a[2, 2]}}
```

Here is another way to produce the same matrix.

```
In[2]:= Array[a, {2, 2}]
Out[2]= {{a[1, 1], a[1, 2]}, {a[2, 1], a[2, 2]}}
```

This creates a 3×2 matrix of zeros.

```
In[3]:= ConstantArray[0, {3, 2}]
Out[3]= {{0, 0}, {0, 0}, {0, 0}}
```

DiagonalMatrix makes a matrix with zeros everywhere except on the leading diagonal.

```
In[4]:= DiagonalMatrix[{a, b, c}]
Out[4]= {{a, 0, 0}, {0, b, 0}, {0, 0, c}}
```

IdentityMatrix[n] produces an n×n identity matrix.

```
In[5]:= IdentityMatrix[3]
Out[5]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

This makes a 3×4 matrix with two nonzero values filled in.

```
In[6]:= Normal[SparseArray[{{2, 3} -> a, {3, 2} -> b}, {3, 4}]]
Out[6]= {{0, 0, 0, 0}, {0, 0, a, 0}, {0, b, 0, 0}}
```

MatrixForm prints the matrix in a two-dimensional form.

```
In[7]:= MatrixForm[%]
```

$$\text{Out[7]//MatrixForm=}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & b & 0 & 0 \end{pmatrix}$$

Table[0,{m},{n}]	a matrix of zeros
Table[If[i>=j,1,0],{i,m},{j,n}]	a lower-triangular matrix
RandomReal[{0,1},{m,n}]	a matrix with random numerical entries

Constructing special types of matrices.

`Table` evaluates `If[i ≥ j, a++, 0]` separately for each element, to give a matrix with sequentially increasing entries in the lower-triangular part.

```
In[8]:= a = 1; Table[If[i ≥ j, a++, 0], {i, 3}, {j, 3}]
Out[8]= {{1, 0, 0}, {2, 3, 0}, {4, 5, 6}}
```

`SparseArray[{},{n,n}]`	a zero matrix
`SparseArray[{i_,i_}->1,{n,n}]`	an $n \times n$ identity matrix
`SparseArray[{i_,j_} /; i >= j->1,{n,n}]`	a lower-triangular matrix

Constructing special types of matrices with `SparseArray`.

This sets up a general lower-triangular matrix.

```
In[9]:= SparseArray[{i_, j_} /; i >= j -> f[i, j], {3, 3}] // MatrixForm
```

$$\text{Out[9]//MatrixForm=}\begin{pmatrix} f[1,1] & 0 & 0 \\ f[2,1] & f[2,2] & 0 \\ f[3,1] & f[3,2] & f[3,3] \end{pmatrix}$$

Getting and Setting Pieces of Matrices

$m[[i,j]]$	the i, j^{th} entry
$m[[i]]$	the i^{th} row
$m[[\text{All},i]]$	the i^{th} column
`Take[m,{i0,i1},{j0,j1}]`	the submatrix with rows i_0 through i_1 and columns j_0 through j_1
$m[[i_0;;i_1,j_0;;j_1]]$	the submatrix with rows i_0 through i_1 and columns j_0 through j_1
$m[[\{i_1,\ldots,i_r\},\{j_1,\ldots,j_s\}]]$	the $r \times s$ submatrix with elements having row indices i_k and column indices j_k
`Tr[m,List]`	elements on the diagonal
`ArrayRules[m]`	positions of nonzero elements

Ways to get pieces of matrices.

Matrices in *Mathematica* are represented as lists of lists. You can use all the standard *Mathematica* list-manipulation operations on matrices.

Here is a sample 3×3 matrix.

```
In[1]:= t = Array[a, {3, 3}]
Out[1]= {{a[1, 1], a[1, 2], a[1, 3]}, {a[2, 1], a[2, 2], a[2, 3]}, {a[3, 1], a[3, 2], a[3, 3]}}
```

This picks out the second row of the matrix.

```
In[2]:= t[[2]]
Out[2]= {a[2, 1], a[2, 2], a[2, 3]}
```

Here is the second column of the matrix.

```
In[3]:= t[[All, 2]]
Out[3]= {a[1, 2], a[2, 2], a[3, 2]}
```

This picks out a submatrix.

```
In[4]:= Take[t, {1, 2}, {2, 3}]
Out[4]= {{a[1, 2], a[1, 3]}, {a[2, 2], a[2, 3]}}
```

$m=\{\{a_{11},a_{12},\ldots\},\{a_{21},a_{22},\ldots\},\ldots\}$	assign m to be a matrix
$m[[i,j]]=a$	reset element $\{i, j\}$ to be a
$m[[i]]=a$	reset all elements in row i to be a
$m[[i]]=\{a_1,a_2,\ldots\}$	reset elements in row i to be $\{a_1, a_2, \ldots\}$
$m[[i_0;;i_1]]=\{v_1,v_2,\ldots\}$	reset rows i_0 through i_1 to be vectors $\{v_1, v_2, \ldots\}$
$m[[\mathrm{All},j]]=a$	reset all elements in column j to be a
$m[[\mathrm{All},j]]=\{a_1,a_2,\ldots\}$	reset elements in column j to be $\{a_1, a_2, \ldots\}$
$m[[i_0;;i_1,j_0;;j_1]]=\{\{a_{11},a_{12},\ldots\},\{a_{21},a_{22},\ldots\},\ldots\}$	reset the submatrix with rows i_0 through i_1 and columns j_0 through j_1 to new values

Resetting parts of matrices.

Here is a 3×3 matrix.

```
In[5]:= m = {{a, b, c}, {d, e, f}, {g, h, i}}
Out[5]= {{a, b, c}, {d, e, f}, {g, h, i}}
```

This resets the 2, 2 element to be x, then shows the whole matrix.

```
In[6]:= m[[2, 2]] = x; m
Out[6]= {{a, b, c}, {d, x, f}, {g, h, i}}
```

This resets all elements in the second column to be z.

```
In[7]:= m[[All, 2]] = z; m
Out[7]= {{a, z, c}, {d, z, f}, {g, z, i}}
```

This separately resets the three elements in the second column.

```
In[8]:= m[[All, 2]] = {i, j, k}; m
Out[8]= {{a, i, c}, {d, j, f}, {g, k, i}}
```

This increments all the values in the second column.

```
In[9]:= m[[All, 2]]++; m
Out[9]= {{a, 1 + i, c}, {d, 1 + j, f}, {g, 1 + k, i}}
```

A range of indices can be specified by using ;; (Span).

This resets the first two rows to be new vectors.

```
In[10]:= m[[1 ;; 2]] = {{u, v, w}, {x, y, z}}; m
Out[10]= {{u, v, w}, {x, y, z}, {g, 1 + k, i}}
```

This resets elements in the first and third columns of each row.

```
In[11]:= m[[All, 1 ;; 3 ;; 2]] = {{1, 2}, {3, 4}, {5, 6}}; m
Out[11]= {{1, v, 2}, {3, y, 4}, {5, 1 + k, 6}}
```

This resets elements in the first and third columns of rows 2 through 3.

```
In[12]:= m[[2 ;; 3, 1 ;; 3 ;; 2]] = {{a, b}, {c, d}}; m
Out[12]= {{1, v, 2}, {a, y, b}, {c, 1 + k, d}}
```

Scalars, Vectors and Matrices

Mathematica represents matrices and vectors using lists. Anything that is not a list *Mathematica* considers as a scalar.

A vector in *Mathematica* consists of a list of scalars. A matrix consists of a list of vectors, representing each of its rows. In order to be a valid matrix, all the rows must be the same length, so that the elements of the matrix effectively form a rectangular array.

`VectorQ[`*expr*`]`	give `True` if *expr* has the form of a vector, and `False` otherwise
`MatrixQ[`*expr*`]`	give `True` if *expr* has the form of a matrix, and `False` otherwise
`Dimensions[`*expr*`]`	a list of the dimensions of a vector or matrix

Functions for testing the structure of vectors and matrices.

The list `{a, b, c}` has the form of a vector.

```
In[1]:= VectorQ[{a, b, c}]
Out[1]= True
```

Anything that is not manifestly a list is treated as a scalar, so applying `VectorQ` gives `False`.

```
In[2]:= VectorQ[x + y]
Out[2]= False
```

This is a 2×3 matrix.

```
In[3]:= Dimensions[{{a, b, c}, {ap, bp, cp}}]
Out[3]= {2, 3}
```

For a vector, `Dimensions` gives a list with a single element equal to the result from `Length`.

```
In[4]:= Dimensions[{a, b, c}]
Out[4]= {3}
```

This object does not count as a matrix because its rows are of different lengths.

```
In[5]:= MatrixQ[{{a, b, c}, {ap, bp}}]
Out[5]= False
```

Operations on Scalars, Vectors and Matrices

Most mathematical functions in *Mathematica* are set up to apply themselves separately to each element in a list. This is true in particular of all functions that carry the attribute `Listable`.

A consequence is that most mathematical functions are applied element by element to matrices and vectors.

The Log applies itself separately to each element in the vector.

```
In[1]:= Log[{a, b, c}]
Out[1]= {Log[a], Log[b], Log[c]}
```

The same is true for a matrix, or, for that matter, for any nested list.

```
In[2]:= Log[{{a, b}, {c, d}}]
Out[2]= {{Log[a], Log[b]}, {Log[c], Log[d]}}
```

The differentiation function D also applies separately to each element in a list.

```
In[3]:= D[{x, x^2, x^3}, x]
Out[3]= {1, 2 x, 3 x^2}
```

The sum of two vectors is carried out element by element.

```
In[4]:= {a, b} + {ap, bp}
Out[4]= {a + ap, b + bp}
```

If you try to add two vectors with different lengths, you get an error.

```
In[5]:= {a, b, c} + {ap, bp}

    Thread::tdlen: Objects of unequal length in {a, b, c} + {ap, bp} cannot be combined. >>

Out[5]= {ap, bp} + {a, b, c}
```

This adds the scalar 1 to each element of the vector.

```
In[6]:= 1 + {a, b}
Out[6]= {1 + a, 1 + b}
```

Any object that is not manifestly a list is treated as a scalar. Here c is treated as a scalar, and added separately to each element in the vector.

```
In[7]:= {a, b} + c
Out[7]= {a + c, b + c}
```

This multiplies each element in the vector by the scalar k.

```
In[8]:= k {a, b}
Out[8]= {a k, b k}
```

It is important to realize that *Mathematica* treats an object as a vector in a particular operation only if the object is explicitly a list at the time when the operation is done. If the object is not explicitly a list, *Mathematica* always treats it as a scalar. This means that you can get different results, depending on whether you assign a particular object to be a list before or after you do a particular operation.

The object `p` is treated as a scalar, and added separately to each element in the vector.

```
In[9]:= {a, b} + p
Out[9]= {a + p, b + p}
```

This is what happens if you now replace `p` by the list `{c, d}`.

```
In[10]:= % /. p -> {c, d}
Out[10]= {{a + c, a + d}, {b + c, b + d}}
```

You would have got a different result if you had replaced `p` by `{c, d}` before you did the first operation.

```
In[11]:= {a, b} + {c, d}
Out[11]= {a + c, b + d}
```

Multiplying Vectors and Matrices

$c\,v$, $c\,m$, etc.	multiply each element by a scalar
u.v, v.m, m.v, m_1.m_2, etc.	vector and matrix multiplication
`Cross[`u`,`v`]`	vector cross product (also input as $u \times v$)
`Outer[Times,`t`,`u`]`	outer product
`KroneckerProduct[`m_1`,`m_2`,...]`	Kronecker product

Different kinds of vector and matrix multiplication.

This multiplies each element of the vector by the scalar `k`.

```
In[1]:= k {a, b, c}
Out[1]= {a k, b k, c k}
```

The "dot" operator gives the scalar product of two vectors.

```
In[2]:= {a, b, c}.{ap, bp, cp}
Out[2]= a ap + b bp + c cp
```

You can also use dot to multiply a matrix by a vector.

```
In[3]:= {{a, b}, {c, d}}.{x, y}
Out[3]= {a x + b y, c x + d y}
```

Dot is also the notation for matrix multiplication in *Mathematica*.

```
In[4]:= {{a, b}, {c, d}}.{{1, 2}, {3, 4}}
Out[4]= {{a + 3 b, 2 a + 4 b}, {c + 3 d, 2 c + 4 d}}
```

It is important to realize that you can use "dot" for both left- and right-multiplication of vectors by matrices. *Mathematica* makes no distinction between "row" and "column" vectors. Dot carries out whatever operation is possible. (In formal terms, $a.b$ contracts the last index of the tensor a with the first index of b.)

Here are definitions for a matrix m and a vector v.

```
In[5]:= m = {{a, b}, {c, d}}; v = {x, y}
Out[5]= {x, y}
```

This left-multiplies the vector v by m. The object v is effectively treated as a column vector in this case.

```
In[6]:= m.v
Out[6]= {a x + b y, c x + d y}
```

You can also use dot to right-multiply v by m. Now v is effectively treated as a row vector.

```
In[7]:= v.m
Out[7]= {a x + c y, b x + d y}
```

You can multiply m by v on both sides, to get a scalar.

```
In[8]:= v.m.v
Out[8]= x (a x + c y) + y (b x + d y)
```

For some purposes, you may need to represent vectors and matrices symbolically, without explicitly giving their elements. You can use dot to represent multiplication of such symbolic objects.

Dot effectively acts here as a noncommutative form of multiplication.

```
In[9]:= a.b.a
Out[9]= a.b.a
```

It is, nevertheless, associative.

```
In[10]:= (a.b).(a.b)
Out[10]= a.b.a.b
```

Dot products of sums are not automatically expanded out.

```
In[11]:= (a + b).c.(d + e)
Out[11]= (a + b).c.(d + e)
```

You can apply the distributive law in this case using the function `Distribute`, as discussed in "Structural Operations".

```
In[12]:= Distribute[%]
Out[12]= a.c.d + a.c.e + b.c.d + b.c.e
```

The "dot" operator gives "inner products" of vectors, matrices, and so on. In more advanced calculations, you may also need to construct outer or Kronecker products of vectors and matrices. You can use the general function `Outer` or `KroneckerProduct` to do this.

The outer product of two vectors is a matrix.

```
In[13]:= Outer[Times, {a, b}, {c, d}]
Out[13]= {{a c, a d}, {b c, b d}}
```

The outer product of a matrix and a vector is a rank three tensor.

```
In[14]:= Outer[Times, {{1, 2}, {3, 4}}, {x, y, z}]
Out[14]= {{{x, y, z}, {2 x, 2 y, 2 z}}, {{3 x, 3 y, 3 z}, {4 x, 4 y, 4 z}}}
```

Outer products are discussed in more detail in "Tensors".

The Kronecker product of a matrix and a vector is a matrix.

```
In[15]:= KroneckerProduct[{{1, 2}, {3, 4}}, {x, y, z}]
Out[15]= {{x, y, z, 2 x, 2 y, 2 z}, {3 x, 3 y, 3 z, 4 x, 4 y, 4 z}}
```

The Kronecker product of a pair of 2×2 matrices is a 4×4 matrix.

```
In[16]:= KroneckerProduct[{{1, 2}, {3, 4}}, {{a, b}, {c, d}}]
Out[16]= {{a, b, 2 a, 2 b}, {c, d, 2 c, 2 d}, {3 a, 3 b, 4 a, 4 b}, {3 c, 3 d, 4 c, 4 d}}
```

Vector Operations

v[[i]] or Part[v,i]	give the i^{th} element in the vector v
c v	scalar multiplication of c times the vector v
u.v	dot product of two vectors
Norm[v]	give the norm of v
Normalize[v]	give a unit vector in the direction of v
Standardize[v]	shift v to have zero mean and unit sample variance
Standardize[v,f_1]	shift v by $f_1[v]$ and scale to have unit sample variance

Basic vector operations.

This is a vector in three dimensions.

```
In[1]:= v = {1, 3, 2}
Out[1]= {1, 3, 2}
```

This gives a vector u in the direction opposite to v with twice the magnitude.

```
In[2]:= u = -2 v
Out[2]= {-2, -6, -4}
```

This reassigns the first component of u to be its negative.

```
In[3]:= u[[1]] = -u[[1]]; u
Out[3]= {2, -6, -4}
```

This gives the dot product of u and v.

```
In[4]:= u.v
Out[4]= -24
```

This is the norm of v.

```
In[5]:= Norm[v]
```

$$\text{Out[5]=} \sqrt{14}$$

This is the unit vector in the same direction as v.

```
In[6]:= Normalize[v]
```

$$\text{Out[6]=} \left\{\frac{1}{\sqrt{14}}, \frac{3}{\sqrt{14}}, \sqrt{\frac{2}{7}}\right\}$$

This verifies that the norm is 1.

```
In[7]:= Norm[%]
Out[7]= 1
```

Transform v to have zero mean and unit sample variance.

```
In[8]:= Standardize[v]
Out[8]= {-1, 1, 0}
```

This shows the transformed values have mean 0 and variance 1.

```
In[9]:= {Mean[%], Variance[%]}
Out[9]= {0, 1}
```

Two vectors are orthogonal if their dot product is zero. A set of vectors is orthonormal if they are all unit vectors and are pairwise orthogonal.

`Projection[u,v]`	give the orthogonal projection of u onto v
`Orthogonalize[{v1,v2, ...}]`	generate an orthonormal set from the given list of vectors

Orthogonal vector operations.

This gives the projection of u onto v.

```
In[10]:= p = Projection[u, v]
```

$$\text{Out[10]}= \left\{-\frac{12}{7}, -\frac{36}{7}, -\frac{24}{7}\right\}$$

p is a scalar multiple of v.

```
In[11]:= p / v
```

$$\text{Out[11]}= \left\{-\frac{12}{7}, -\frac{12}{7}, -\frac{12}{7}\right\}$$

u – p is orthogonal to v.

```
In[12]:= (u - p).v
Out[12]= 0
```

Starting from the set of vectors {u, v}, this finds an orthonormal set of two vectors.

```
In[13]:= Orthogonalize[{u, v}]
```

$$\text{Out[13]}= \left\{\left\{\frac{1}{\sqrt{14}}, -\frac{3}{\sqrt{14}}, -\sqrt{\frac{2}{7}}\right\}, \left\{\sqrt{\frac{13}{14}}, \frac{3}{\sqrt{182}}, \sqrt{\frac{2}{91}}\right\}\right\}$$

When one of the vectors is linearly dependent on the vectors preceding it, the corresponding position in the result will be a zero vector.

```
In[14]:= Orthogonalize[{v, p, u}]
```

$$\text{Out[14]}= \left\{\left\{\frac{1}{\sqrt{14}}, \frac{3}{\sqrt{14}}, \sqrt{\frac{2}{7}}\right\}, \{0, 0, 0\}, \left\{\sqrt{\frac{13}{14}}, -\frac{3}{\sqrt{182}}, -\sqrt{\frac{2}{91}}\right\}\right\}$$

Matrix Inversion

`Inverse[`m`]`	find the inverse of a square matrix

Matrix inversion.

Here is a simple 2×2 matrix.

```
In[1]:= m = {{a, b}, {c, d}}
Out[1]= {{a, b}, {c, d}}
```

This gives the inverse of `m`. In producing this formula, *Mathematica* implicitly assumes that the determinant `a d – b c` is nonzero.

```
In[2]:= Inverse[m]
```

$$\text{Out[2]}= \left\{\left\{\frac{d}{-b c+a d}, -\frac{b}{-b c+a d}\right\}, \left\{-\frac{c}{-b c+a d}, \frac{a}{-b c+a d}\right\}\right\}$$

Multiplying the inverse by the original matrix should give the identity matrix.

```
In[3]:= %.m
```

$$\text{Out[3]}= \left\{\left\{-\frac{b c}{-b c+a d}+\frac{a d}{-b c+a d}, 0\right\}, \left\{0, -\frac{b c}{-b c+a d}+\frac{a d}{-b c+a d}\right\}\right\}$$

You have to use `Together` to clear the denominators, and get back a standard identity matrix.

```
In[4]:= Together[%]
Out[4]= {{1, 0}, {0, 1}}
```

Here is a matrix of rational numbers.

```
In[5]:= hb = Table[1 / (i + j), {i, 4}, {j, 4}]
```

$$\text{Out[5]}= \left\{\left\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}\right\}, \left\{\frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}\right\}, \left\{\frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}\right\}, \left\{\frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8}\right\}\right\}$$

Mathematica finds the exact inverse of the matrix.

```
In[6]:= Inverse[hb]
Out[6]= {{200, -1200, 2100, -1120}, {-1200, 8100, -15120, 8400},
 {2100, -15120, 29400, -16800}, {-1120, 8400, -16800, 9800}}
```

Multiplying by the original matrix gives the identity matrix.

```
In[7]:= %.hb
Out[7]= {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}
```

If you try to invert a singular matrix, *Mathematica* prints a warning message, and returns the input unchanged.

```
In[8]:= Inverse[{{1, 2}, {1, 2}}]

    Inverse::sing : Matrix {{1, 2}, {1, 2}} is singular. >>

Out[8]= Inverse[{{1, 2}, {1, 2}}]
```

If you give a matrix with exact symbolic or numerical entries, *Mathematica* gives the exact inverse. If, on the other hand, some of the entries in your matrix are approximate real numbers, then *Mathematica* finds an approximate numerical result.

Here is a matrix containing approximate real numbers.

```
In[9]:= m = {{1.2, 5.7}, {1.3, 5.6}}
Out[9]= {{1.2, 5.7}, {1.3, 5.6}}
```

This finds the numerical inverse.

```
In[10]:= Inverse[%]
Out[10]= {{-8.11594, 8.26087}, {1.88406, -1.73913}}
```

Multiplying by the original matrix gives you an identity matrix with small round-off errors.

```
In[11]:= %.m
Out[11]= {{1., 1.66187×10^-15}, {3.27429×10^-16, 1.}}
```

You can get rid of small off-diagonal terms using `Chop`.

```
In[12]:= Chop[%]
Out[12]= {{1., 0}, {0, 1.}}
```

When you try to invert a matrix with exact numerical entries, *Mathematica* can always tell whether or not the matrix is singular. When you invert an approximate numerical matrix, *Mathematica* can usually not tell for certain whether or not the matrix is singular: all it can tell is, for example, that the determinant is small compared to the entries of the matrix. When *Mathematica* suspects that you are trying to invert a singular numerical matrix, it prints a warning.

Mathematica prints a warning if you invert a numerical matrix that it suspects is singular.

```
In[13]:= Inverse[{{1., 2.}, {1., 2.}}]

    Inverse::sing : Matrix {{1., 2.}, {1., 2.}} is singular. >>

Out[13]= Inverse[{{1., 2.}, {1., 2.}}]
```

This matrix is singular, but the warning is different, and the result is useless.

```
In[14]:= Inverse[N[{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}]]

Inverse::luc: Result for Inverse of badly conditioned matrix
    {{1., 2., 3.}, {4., 5., 6.}, {7., 8., 9.}} may contain significant numerical errors. >>

Out[14]= {{3.15221×10^15, -6.30442×10^15, 3.15221×10^15},
  {-6.30442×10^15, 1.26088×10^16, -6.30442×10^15}, {3.15221×10^15, -6.30442×10^15, 3.15221×10^15}}
```

If you work with high-precision approximate numbers, *Mathematica* will keep track of the precision of matrix inverses that you generate.

This generates a 6×6 numerical matrix with entries of 20-digit precision.

```
In[15]:= m = N[Table[GCD[i, j] + 1, {i, 6}, {j, 6}], 20];
```

This takes the matrix, multiplies it by its inverse, and shows the first row of the result.

```
In[16]:= (m.Inverse[m])[[1]]

Out[16]= {1.0000000000000000000, 0.×10^-19, 0.×10^-19, 0.×10^-20, 0.×10^-20, 0.×10^-20}
```

This generates a 20-digit numerical approximation to a 6×6 Hilbert matrix. Hilbert matrices are notoriously hard to invert numerically.

```
In[17]:= m = N[Table[1 / (i + j - 1), {i, 6}, {j, 6}], 20];
```

The result is still correct, but the zeros now have lower accuracy.

```
In[18]:= (m.Inverse[m])[[1]]

Out[18]= {1.000000000000000, 0.×10^-15, 0.×10^-14, 0.×10^-14, 0.×10^-14, 0.×10^-14}
```

`Inverse` works only on square matrices. "Advanced Matrix Operations" discusses the function `PseudoInverse`, which can also be used with nonsquare matrices.

Basic Matrix Operations

`Transpose[`m`]`	transpose m^{T}
`ConjugateTranspose[`m`]`	conjugate transpose $m^{\dagger}$ (Hermitian conjugate)
`Inverse[`m`]`	matrix inverse
`Det[`m`]`	determinant
`Minors[`m`]`	matrix of minors

`Minors[m,k]`	k^{th} minors
`Tr[m]`	trace
`MatrixRank[m]`	rank of matrix

Some basic matrix operations.

Transposing a matrix interchanges the rows and columns in the matrix. If you transpose an $m \times n$ matrix, you get an $n \times m$ matrix as the result.

Transposing a 2×3 matrix gives a 3×2 result.

```
In[1]:= Transpose[{{a, b, c}, {ap, bp, cp}}]
Out[1]= {{a, ap}, {b, bp}, {c, cp}}
```

`Det[m]` gives the determinant of a square matrix m. `Minors[m]` is the matrix whose $(i, j)^{\text{th}}$ element gives the determinant of the submatrix obtained by deleting the $(n-i+1)^{\text{th}}$ row and the $(n-j+1)^{\text{th}}$ column of m. The $(i, j)^{\text{th}}$ cofactor of m is $(-1)^{i+j}$ times the $(n-i+1, n-j+1)^{\text{th}}$ element of the matrix of minors.

`Minors[m, k]` gives the determinants of the $k \times k$ submatrices obtained by picking each possible set of k rows and k columns from m. Note that you can apply `Minors` to rectangular, as well as square, matrices.

Here is the determinant of a simple 2×2 matrix.

```
In[2]:= Det[{{a, b}, {c, d}}]
Out[2]= -b c + a d
```

This generates a 3×3 matrix, whose i, j^{th} entry is `a[i, j]`.

```
In[3]:= m = Array[a, {3, 3}]
Out[3]= {{a[1, 1], a[1, 2], a[1, 3]}, {a[2, 1], a[2, 2], a[2, 3]}, {a[3, 1], a[3, 2], a[3, 3]}}
```

Here is the determinant of `m`.

```
In[4]:= Det[m]
Out[4]= -a[1, 3] a[2, 2] a[3, 1] + a[1, 2] a[2, 3] a[3, 1] + a[1, 3] a[2, 1] a[3, 2] -
         a[1, 1] a[2, 3] a[3, 2] - a[1, 2] a[2, 1] a[3, 3] + a[1, 1] a[2, 2] a[3, 3]
```

The *trace* or *spur* of a matrix `Tr[m]` is the sum of the terms on the leading diagonal.

This finds the trace of a simple 2×2 matrix.

```
In[5]:= Tr[{{a, b}, {c, d}}]
Out[5]= a + d
```

The *rank* of a matrix is the number of linearly independent rows or columns.

This finds the rank of a matrix.

```
In[6]:= MatrixRank[{{1, 2}, {1, 2}}]
Out[6]= 1
```

`MatrixPower[m,n]`	n^{th} matrix power
`MatrixExp[m]`	matrix exponential

Powers and exponentials of matrices.

Here is a 2×2 matrix.

```
In[7]:= m = {{0.4, 0.6}, {0.525, 0.475}}
Out[7]= {{0.4, 0.6}, {0.525, 0.475}}
```

This gives the third matrix power of `m`.

```
In[8]:= MatrixPower[m, 3]
Out[8]= {{0.465625, 0.534375}, {0.467578, 0.532422}}
```

It is equivalent to multiplying three copies of the matrix.

```
In[9]:= m.m.m
Out[9]= {{0.465625, 0.534375}, {0.467578, 0.532422}}
```

Here is the millionth matrix power.

```
In[10]:= MatrixPower[m, 10^6]
Out[10]= {{0.466667, 0.533333}, {0.466667, 0.533333}}
```

The matrix exponential of a matrix m is $\sum_{k=0}^{\infty} m^k / k!$, where m^k indicates a matrix power.

This gives the matrix exponential of `m`.

```
In[11]:= MatrixExp[m]
Out[11]= {{1.7392, 0.979085}, {0.8567, 1.86158}}
```

Here is an approximation to the exponential of `m`, based on a power series approximation.

```
In[12]:= Sum[MatrixPower[m, k] / k!, {k, 0, 5}]
Out[12]= {{1.73844, 0.978224}, {0.855946, 1.86072}}
```

Solving Linear Systems

Many calculations involve solving systems of linear equations. In many cases, you will find it convenient to write down the equations explicitly, and then solve them using `Solve`.

In some cases, however, you may prefer to convert the system of linear equations into a matrix equation, and then apply matrix manipulation operations to solve it. This approach is often useful when the system of equations arises as part of a general algorithm, and you do not know in advance how many variables will be involved.

A system of linear equations can be stated in matrix form as $m.x = b$, where x is the vector of variables.

Note that if your system of equations is sparse, so that most of the entries in the matrix m are zero, then it is best to represent the matrix as a `SparseArray` object. As discussed in "Sparse Arrays: Linear Algebra", you can convert from symbolic equations to `SparseArray` objects using `CoefficientArrays`. All the functions described here work on `SparseArray` objects as well as ordinary matrices.

`LinearSolve[`m`,`b`]`	a vector x which solves the matrix equation $m.x == b$
`NullSpace[`m`]`	a list of linearly independent vectors whose linear combinations span all solutions to the matrix equation $m.x == 0$
`MatrixRank[`m`]`	the number of linearly independent rows or columns of m
`RowReduce[`m`]`	a simplified form of m obtained by making linear combinations of rows

Solving and analyzing linear systems.

Here is a 2x2 matrix.

```
In[1]:= m = {{1, 5}, {2, 1}}
Out[1]= {{1, 5}, {2, 1}}
```

This gives two linear equations.

```
In[2]:= m.{x, y} == {a, b}
Out[2]= {x + 5 y, 2 x + y} == {a, b}
```

You can use `Solve` directly to solve these equations.

```
In[3]:= Solve[%, {x, y}]
```

$$\text{Out[3]= } \left\{\left\{x \to \frac{1}{9}(-a + 5b),\ y \to \frac{1}{9}(2a - b)\right\}\right\}$$

You can also get the vector of solutions by calling `LinearSolve`. The result is equivalent to the one you get from `Solve`.

```
In[4]:= LinearSolve[m, {a, b}]
```

$$\text{Out[4]= } \left\{\frac{1}{9}(-a + 5b),\ \frac{1}{9}(2a - b)\right\}$$

Another way to solve the equations is to invert the matrix `m`, and then multiply `{a, b}` by the inverse. This is not as efficient as using `LinearSolve`.

```
In[5]:= Inverse[m].{a, b}
```

$$\text{Out[5]= } \left\{-\frac{a}{9} + \frac{5b}{9},\ \frac{2a}{9} - \frac{b}{9}\right\}$$

`RowReduce` performs a version of Gaussian elimination and can also be used to solve the equations.

```
In[6]:= RowReduce[{{1, 5, a}, {2, 1, b}}]
```

$$\text{Out[6]= } \left\{\left\{1, 0, \frac{1}{9}(-a + 5b)\right\}, \left\{0, 1, \frac{1}{9}(2a - b)\right\}\right\}$$

If you have a square matrix m with a nonzero determinant, then you can always find a unique solution to the matrix equation $m.x = b$ for any b. If, however, the matrix m has determinant zero, then there may be either no vector, or an infinite number of vectors x which satisfy $m.x = b$ for a particular b. This occurs when the linear equations embodied in m are not independent.

When m has determinant zero, it is nevertheless always possible to find nonzero vectors x that satisfy $m.x = 0$. The set of vectors x satisfying this equation form the *null space* or *kernel* of the matrix m. Any of these vectors can be expressed as a linear combination of a particular set of basis vectors, which can be obtained using `NullSpace[m]`.

Here is a simple matrix, corresponding to two identical linear equations.

```
In[7]:= m = {{1, 2}, {1, 2}}
Out[7]= {{1, 2}, {1, 2}}
```

The matrix has determinant zero.

```
In[8]:= Det[m]
Out[8]= 0
```

`LinearSolve` cannot find a solution to the equation $m.x == b$ in this case.

```
In[9]:= LinearSolve[m, {a, b}]
LinearSolve::nosol : Linear equation encountered that has no solution. >>
Out[9]= LinearSolve[{{1, 2}, {1, 2}}, {a, b}]
```

There is a single basis vector for the null space of `m`.

```
In[10]:= NullSpace[m]
Out[10]= {{-2, 1}}
```

Multiplying the basis vector for the null space by `m` gives the zero vector.

```
In[11]:= m.%[[1]]
Out[11]= {0, 0}
```

There is only `1` linearly independent row in `m`.

```
In[12]:= MatrixRank[m]
Out[12]= 1
```

`NullSpace` and `MatrixRank` have to determine whether particular combinations of matrix elements are zero. For approximate numerical matrices, the `Tolerance` option can be used to specify how close to zero is considered good enough. For exact symbolic matrices, you may sometimes need to specify something like `ZeroTest -> (FullSimplify[#] == 0 &)` to force more to be done to test whether symbolic expressions are zero.

Here is a simple symbolic matrix with determinant zero.

```
In[13]:= m = {{a, b, c}, {2 a, 2 b, 2 c}, {3 a, 3 b, 3 c}}
Out[13]= {{a, b, c}, {2 a, 2 b, 2 c}, {3 a, 3 b, 3 c}}
```

The basis for the null space of `m` contains two vectors.

```
In[14]:= NullSpace[m]
```

$$\text{Out[14]}= \left\{\left\{-\frac{c}{a}, 0, 1\right\}, \left\{-\frac{b}{a}, 1, 0\right\}\right\}$$

Multiplying `m` by any linear combination of these vectors gives zero.

```
In[15]:= Simplify[m.(x %[[1]] + y %[[2]])]
Out[15]= {0, 0, 0}
```

An important feature of functions like `LinearSolve` and `NullSpace` is that they work with *rectangular*, as well as *square*, matrices.

When you represent a system of linear equations by a matrix equation of the form $m.x = b$, the number of columns in m gives the number of variables, and the number of rows gives the number of equations. There are a number of cases.

Underdetermined	number of equations less than the number of variables; no solutions or many solutions may exist
Overdetermined	number of equations more than the number of variables; solutions may or may not exist
Nonsingular	number of independent equations equal to the number of variables, and determinant nonzero; a unique solution exists
Consistent	at least one solution exists
Inconsistent	no solutions exist

Classes of linear systems represented by rectangular matrices.

This asks for the solution to the inconsistent set of equations $x = 1$ and $x = 0$.

```
In[16]:= LinearSolve[{{1}, {1}}, {1, 0}]

LinearSolve::nosol : Linear equation encountered that has no solution. >>

Out[16]= LinearSolve[{{1}, {1}}, {1, 0}]
```

This matrix represents two equations, for three variables.

```
In[17]:= m = {{1, 3, 4}, {2, 1, 3}}
Out[17]= {{1, 3, 4}, {2, 1, 3}}
```

LinearSolve gives one of the possible solutions to this underdetermined set of equations.

```
In[18]:= v = LinearSolve[m, {1, 1}]
Out[18]= {2/5, 1/5, 0}
```

When a matrix represents an underdetermined system of equations, the matrix has a nontrivial null space. In this case, the null space is spanned by a single vector.

```
In[19]:= NullSpace[m]
Out[19]= {{-1, -1, 1}}
```

If you take the solution you get from LinearSolve, and add any linear combination of the basis vectors for the null space, you still get a solution.

```
In[20]:= m.(v + 4 %[[1]])
Out[20]= {1, 1}
```

The number of independent equations is the *rank* of the matrix MatrixRank[m]. The number of redundant equations is Length[NullSpace[m]]. Note that the sum of these quantities is always equal to the number of columns in m.

LinearSolve[m]	generate a function for solving equations of the form $m.x = b$

Generating LinearSolveFunction objects.

In some applications, you will want to solve equations of the form $m.x = b$ many times with the same m, but different b. You can do this efficiently in *Mathematica* by using LinearSolve[m] to create a single LinearSolveFunction that you can apply to as many vectors as you want.

This creates a LinearSolveFunction.

```
In[21]:= f = LinearSolve[{{1, 4}, {2, 3}}]
Out[21]= LinearSolveFunction[{2, 2}, <>]
```

You can apply this to a vector.

```
In[22]:= f[{5, 7}]
```

$$\text{Out[22]}= \left\{\frac{13}{5}, \frac{3}{5}\right\}$$

You get the same result by giving the vector as an explicit second argument to `LinearSolve`.

```
In[23]:= LinearSolve[{{1, 4}, {2, 3}}, {5, 7}]
```

$$\text{Out[23]}= \left\{\frac{13}{5}, \frac{3}{5}\right\}$$

But you can apply `f` to any vector you want.

```
In[24]:= f[{-5, 9}]
```

$$\text{Out[24]}= \left\{\frac{51}{5}, -\frac{19}{5}\right\}$$

`LeastSquares[`m`,`b`]`	give a vector x that solves the least-squares problem $m.x == b$

Solving least-squares problems.

This linear system is inconsistent.

```
In[25]:= LinearSolve[{{1, 2}, {3, 4}, {5, 6}}, {-1, 0, 2}]

LinearSolve::nosol : Linear equation encountered that has no solution. >>

Out[25]= LinearSolve[{{1, 2}, {3, 4}, {5, 6}}, {-1, 0, 2}]
```

`LeastSquares` finds a vector x that minimizes $m.x - b$ in the least-squares sense.

```
In[26]:= LeastSquares[{{1, 2}, {3, 4}, {5, 6}}, {-1, 0, 2}]
```

$$\text{Out[26]}= \left\{\frac{8}{3}, -\frac{23}{12}\right\}$$

Eigenvalues and Eigenvectors

`Eigenvalues[`m`]`	a list of the eigenvalues of m
`Eigenvectors[`m`]`	a list of the eigenvectors of m

`Eigensystem[`*m*`]`	a list of the form {*eigenvalues*, *eigenvectors*}
`Eigenvalues[N[`*m*`]]`, etc.	numerical eigenvalues
`Eigenvalues[N[`*m*`,`*p*`]]`, etc.	numerical eigenvalues, starting with *p*-digit precision
`CharacteristicPolynomial[`*m*`,`*x*`]`	the characteristic polynomial of *m*

Eigenvalues and eigenvectors.

The eigenvalues of a matrix m are the values λ_i for which one can find nonzero vectors v_i such that $m.v_i = \lambda_i v_i$. The eigenvectors are the vectors v_i.

The *characteristic polynomial* `CharacteristicPolynomial[`*m*`, `*x*`]` for an $n \times n$ matrix is given by `Det[`*m* – *x* `IdentityMatrix[`*n*`]]`. The eigenvalues are the roots of this polynomial.

Finding the eigenvalues of an $n \times n$ matrix in general involves solving an n^{th}-degree polynomial equation. For $n \geq 5$, therefore, the results cannot in general be expressed purely in terms of explicit radicals. `Root` objects can nevertheless always be used, although except for fairly sparse or otherwise simple matrices the expressions obtained are often unmanageably complex.

Even for a matrix as simple as this, the explicit form of the eigenvalues is quite complicated.

```
In[1]:= Eigenvalues[{{a, b}, {-b, 2 a}}]
```

$$\text{Out[1]}= \left\{\frac{1}{2}\left(3\,a - \sqrt{a^2 - 4\,b^2}\right), \frac{1}{2}\left(3\,a + \sqrt{a^2 - 4\,b^2}\right)\right\}$$

If you give a matrix of approximate real numbers, *Mathematica* will find the approximate numerical eigenvalues and eigenvectors.

Here is a 2×2 numerical matrix.

```
In[2]:= m = {{2.3, 4.5}, {6.7, -1.2}}
Out[2]= {{2.3, 4.5}, {6.7, -1.2}}
```

The matrix has two eigenvalues, in this case both real.

```
In[3]:= Eigenvalues[m]
Out[3]= {6.31303, -5.21303}
```

Here are the two eigenvectors of `m`.

```
In[4]:= Eigenvectors[m]
Out[4]= {{0.746335, 0.66557}, {-0.513839, 0.857886}}
```

`Eigensystem` computes the eigenvalues and eigenvectors at the same time. The assignment sets `vals` to the list of eigenvalues, and `vecs` to the list of eigenvectors.

```
In[5]:= {vals, vecs} = Eigensystem[m]
Out[5]= {{6.31303, -5.21303}, {{0.746335, 0.66557}, {-0.513839, 0.857886}}}
```

This verifies that the first eigenvalue and eigenvector satisfy the appropriate condition.

```
In[6]:= m.vecs[[1]] == vals[[1]] vecs[[1]]
Out[6]= True
```

This finds the eigenvalues of a random 4×4 matrix. For nonsymmetric matrices, the eigenvalues can have imaginary parts.

```
In[7]:= Eigenvalues[Table[RandomReal[], {4}, {4}]]
Out[7]= {2.30022, 0.319764 + 0.547199 i, 0.319764 - 0.547199 i, 0.449291}
```

The function `Eigenvalues` always gives you a list of n eigenvalues for an $n \times n$ matrix. The eigenvalues correspond to the roots of the characteristic polynomial for the matrix, and may not necessarily be distinct. `Eigenvectors`, on the other hand, gives a list of eigenvectors which are guaranteed to be independent. If the number of such eigenvectors is less than n, then `Eigenvectors` appends zero vectors to the list it returns, so that the total length of the list is always n.

Here is a 3×3 matrix.

```
In[8]:= mz = {{0, 1, 0}, {0, 0, 1}, {0, 0, 0}}
Out[8]= {{0, 1, 0}, {0, 0, 1}, {0, 0, 0}}
```

The matrix has three eigenvalues, all equal to zero.

```
In[9]:= Eigenvalues[mz]
Out[9]= {0, 0, 0}
```

There is, however, only one independent eigenvector for the matrix. `Eigenvectors` appends two zero vectors to give a total of three vectors in this case.

```
In[10]:= Eigenvectors[mz]
Out[10]= {{1, 0, 0}, {0, 0, 0}, {0, 0, 0}}
```

This gives the characteristic polynomial of the matrix.

```
In[11]:= CharacteristicPolynomial[mz, x]
Out[11]= -x^3
```

`Eigenvalues[`m`,`k`]`	the largest k eigenvalues of m
`Eigenvectors[`m`,`k`]`	the corresponding eigenvectors of m
`Eigensystem[`m`,`k`]`	the largest k eigenvalues with corresponding eigenvectors
`Eigenvalues[`m`,-`k`]`	the smallest k eigenvalues of m
`Eigenvectors[`m`,-`k`]`	the corresponding eigenvectors of m
`Eigensystem[`m`,-`k`]`	the smallest k eigenvalues with corresponding eigenvectors

Finding largest and smallest eigenvalues.

`Eigenvalues` sorts numeric eigenvalues so that the ones with large absolute value come first. In many situations, you may be interested only in the largest or smallest eigenvalues of a matrix. You can get these efficiently using `Eigenvalues[`m`, `k`]` and `Eigenvalues[`m`, -`k`]`.

This computes the exact eigenvalues of an integer matrix.

```
In[12]:= Eigenvalues[{{1, 2}, {3, 4}}]
Out[12]= {1/2 (5 + Sqrt[33]), 1/2 (5 - Sqrt[33])}
```

The eigenvalues are sorted in decreasing order of size.

```
In[13]:= N[%]
Out[13]= {5.37228, -0.372281}
```

This gives the three eigenvalues with largest absolute value.

```
In[14]:= Eigenvalues[Table[N[Tan[i / j]], {i, 10}, {j, 10}], 3]
Out[14]= {10.044, 2.94396 + 6.03728 i, 2.94396 - 6.03728 i}
```

`Eigenvalues[{`m`,`a`}]`	the generalized eigenvalues of m with respect to a
`Eigenvectors[{`m`,`a`}]`	the generalized eigenvectors of m with respect to a
`Eigensystem[{`m`,`a`}]`	the generalized eigensystem of m with respect to a
`CharacteristicPolynomial[{`m`,`a`},`x`]`	the generalized characteristic polynomial of m with respect to a

Generalized eigenvalues, eigenvectors, and characteristic polynomial.

The generalized eigenvalues for a matrix m with respect to a matrix a are defined to be those λ_i for which $m.v_i = \lambda_i\, a.v_i$.

The generalized eigenvalues correspond to zeros of the generalized characteristic polynomial `Det[m - x a]`.

Note that while ordinary matrix eigenvalues always have definite values, some generalized eigenvalues will always be `Indeterminate` if the generalized characteristic polynomial vanishes, which happens if m and a share a null space. Note also that generalized eigenvalues can be infinite.

These two matrices share a one-dimensional null space, so one generalized eigenvalue is `Indeterminate`.

```
In[15]:= Eigenvalues[{{{1.5, 0}, {0, 0}}, {{2, 0}, {1, 0}}}]
Out[15]= {0., Indeterminate}
```

This gives a generalized characteristic polynomial.

```
In[16]:= CharacteristicPolynomial[{{{1.5, 0}, {0, 1}}, {{2, 0}, {1, 0}}}, x]
Out[16]= 1.5 - 2. x
```

Advanced Matrix Operations

`SingularValueList[m]`	the list of nonzero singular values of m
`SingularValueList[m,k]`	the k largest singular values of m
`SingularValueList[{m,a}]`	the generalized singular values of m with respect to a
`Norm[m,p]`	the p-norm of m
`Norm[m,"Frobenius"]`	the Frobenius norm of m

Finding singular values and norms of matrices.

The *singular values* of a matrix $\boldsymbol{m}$ are the square roots of the eigenvalues of $\boldsymbol{m}.\boldsymbol{m}^*$, where $*$ denotes Hermitian transpose. The number of such singular values is the smaller dimension of the matrix. `SingularValueList` sorts the singular values from largest to smallest. Very small singular values are usually numerically meaningless. With the option setting `Tolerance -> t`, `SingularValueList` drops singular values that are less than a fraction t of the largest singular value. For approximate numerical matrices, the tolerance is by default slightly greater than zero.

If you multiply the vector for each point in a unit sphere in n-dimensional space by an $m \times n$ matrix m, then you get an m-dimensional ellipsoid, whose principal axes have lengths given by the singular values of m.

The *2-norm* of a matrix `Norm[m, 2]` is the largest principal axis of the ellipsoid, equal to the largest singular value of the matrix. This is also the maximum 2-norm length of $m.v$ for any possible unit vector v.

The *p-norm* of a matrix `Norm[m, p]` is in general the maximum p-norm length of $m.v$ that can be attained. The cases most often considered are $p = 1$, $p = 2$ and $p = \infty$. Also sometimes considered is the Frobenius norm `Norm[m, "Frobenius"]`, which is the square root of the trace of $m.m^*$.

`LUDecomposition[m]`	the LU decomposition
`CholeskyDecomposition[m]`	the Cholesky decomposition

Decomposing square matrices into triangular forms.

When you create a `LinearSolveFunction` using `LinearSolve[m]`, this often works by decomposing the matrix m into triangular forms, and sometimes it is useful to be able to get such forms explicitly.

LU decomposition effectively factors any square matrix into a product of lower- and upper-triangular matrices. *Cholesky decomposition* effectively factors any Hermitian positive-definite matrix into a product of a lower-triangular matrix and its Hermitian conjugate, which can be viewed as the analog of finding a square root of a matrix.

`PseudoInverse[m]`	the pseudoinverse
`QRDecomposition[m]`	the QR decomposition
`SingularValueDecomposition[m]`	the singular value decomposition
`SingularValueDecomposition[{m,a}]`	the generalized singular value decomposition

Orthogonal decompositions of matrices.

The standard definition for the inverse of a matrix fails if the matrix is not square or is singular. The *pseudoinverse* $m^{(-1)}$ of a matrix m can however still be defined. It is set up to minimize the sum of the squares of all entries in $m.m^{(-1)} - I$, where I is the identity matrix. The pseudoinverse is sometimes known as the generalized inverse, or the Moore-Penrose inverse. It is particularly used for problems related to least-squares fitting.

QR decomposition writes any matrix m as a product $q^* . r$, where q is an orthonormal matrix, $*$ denotes Hermitian transpose, and r is a triangular matrix, in which all entries below the leading diagonal are zero.

Singular value decomposition, or *SVD*, is an underlying element in many numerical matrix algorithms. The basic idea is to write any matrix m in the form $u . s . v^*$, where s is a matrix with the singular values of m on its diagonal, u and v are orthonormal matrices, and v^* is the Hermitian transpose of v.

`JordanDecomposition[m]`	the Jordan decomposition
`SchurDecomposition[m]`	the Schur decomposition
`SchurDecomposition[{m,a}]`	the generalized Schur decomposition
`HessenbergDecomposition[m]`	the Hessenberg decomposition

Functions related to eigenvalue problems.

Most square matrices can be reduced to a diagonal matrix of eigenvalues by applying a matrix of their eigenvectors as a similarity transformation. But even when there are not enough eigenvectors to do this, one can still reduce a matrix to a *Jordan form* in which there are both eigenvalues and Jordan blocks on the diagonal. *Jordan decomposition* in general writes any square matrix in the form $s . j . s^{-1}$.

Numerically more stable is the *Schur decomposition*, which writes any square matrix m in the form $q . t . q^*$, where q is an orthonormal matrix, and t is block upper-triangular. Also related is the *Hessenberg decomposition*, which writes a square matrix m in the form $p . h . p^*$, where p is an orthonormal matrix, and h can have nonzero elements down to the diagonal below the leading diagonal.

Tensors

Tensors are mathematical objects that give generalizations of vectors and matrices. In *Mathematica*, a tensor is represented as a set of lists, nested to a certain number of levels. The nesting level is the *rank* of the tensor.

rank 0	scalar
rank 1	vector
rank 2	matrix
rank k	rank k tensor

Interpretations of nested lists.

A tensor of rank k is essentially a k-dimensional table of values. To be a true rank k tensor, it must be possible to arrange the elements in the table in a k-dimensional cuboidal array. There can be no holes or protrusions in the cuboid.

The *indices* that specify a particular element in the tensor correspond to the coordinates in the cuboid. The *dimensions* of the tensor correspond to the side lengths of the cuboid.

One simple way that a rank k tensor can arise is in giving a table of values for a function of k variables. In physics, the tensors that occur typically have indices which run over the possible directions in space or spacetime. Notice, however, that there is no built-in notion of covariant and contravariant tensor indices in *Mathematica*: you have to set these up explicitly using metric tensors.

`Table[`f`,{`i_1`,`n_1`},{`i_2`,`n_2`},…,{`i_k`,`n_k`}]`	
	create an $n_1 \times n_2 \times \ldots \times n_k$ tensor whose elements are the values of f
`Array[`a`,{`n_1`,`n_2`,…,`n_k`}]`	create an $n_1 \times n_2 \times \ldots \times n_k$ tensor with elements given by applying a to each set of indices
`ArrayQ[`t`,`n`]`	test whether t is a tensor of rank n
`Dimensions[`t`]`	give a list of the dimensions of a tensor
`ArrayDepth[`t`]`	find the rank of a tensor
`MatrixForm[`t`]`	print with the elements of t arranged in a two-dimensional array

Functions for creating and testing the structure of tensors.

Here is a 2×3×2 tensor.

```
In[1]:= t = Table[i1 + i2 i3, {i1, 2}, {i2, 3}, {i3, 2}]
Out[1]= {{{2, 3}, {3, 5}, {4, 7}}, {{3, 4}, {4, 6}, {5, 8}}}
```

This is another way to produce the same tensor.

```
In[2]:= Array[(#1 + #2 #3) &, {2, 3, 2}]
Out[2]= {{{2, 3}, {3, 5}, {4, 7}}, {{3, 4}, {4, 6}, {5, 8}}}
```

`MatrixForm` displays the elements of the tensor in a two-dimensional array. You can think of the array as being a 2×3 matrix of column vectors.

```
In[3]:= MatrixForm[t]
```

Out[3]//MatrixForm=

$$\begin{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} & \begin{pmatrix} 3 \\ 5 \end{pmatrix} & \begin{pmatrix} 4 \\ 7 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 4 \end{pmatrix} & \begin{pmatrix} 4 \\ 6 \end{pmatrix} & \begin{pmatrix} 5 \\ 8 \end{pmatrix} \end{pmatrix}$$

`Dimensions` gives the dimensions of the tensor.

```
In[4]:= Dimensions[t]
Out[4]= {2, 3, 2}
```

Here is the 111 element of the tensor.

```
In[5]:= t[[1, 1, 1]]
Out[5]= 2
```

`ArrayDepth` gives the rank of the tensor.

```
In[6]:= ArrayDepth[t]
Out[6]= 3
```

The rank of a tensor is equal to the number of indices needed to specify each element. You can pick out subtensors by using a smaller number of indices.

`Transpose[`t`]`	transpose the first two indices in a tensor
`Transpose[`t`,{`p_1`,`p_2`,…}]`	transpose the indices in a tensor so that the k^{th} becomes the $p_k{}^{\text{th}}$
`Tr[`t`,`f`]`	form the generalized trace of the tensor t
`Outer[`f`,`t_1`,`t_2`]`	form the generalized outer product of the tensors t_1 and t_2 with "multiplication operator" f
t_1`.`t_2	form the dot product of t_1 and t_2 (last index of t_1 contracted with first index of t_2)
`Inner[`f`,`t_1`,`t_2`,`g`]`	form the generalized inner product, with "multiplication operator" f and "addition operator" g

Tensor manipulation operations.

You can think of a rank k tensor as having k "slots" into which you insert indices. Applying `Transpose` is effectively a way of reordering these slots. If you think of the elements of a tensor as forming a k-dimensional cuboid, you can view `Transpose` as effectively rotating (and possibly reflecting) the cuboid.

In the most general case, `Transpose` allows you to specify an arbitrary reordering to apply to the indices of a tensor. The function `Transpose[T, {p1, p2, ..., pk}]` gives you a new tensor T' such that the value of $T'_{i_1\, i_2 \ldots i_k}$ is given by $T_{i_{p_1}\, i_{p_2} \ldots i_{p_k}}$.

If you originally had an $n_{p_1} \times n_{p_2} \times \ldots \times n_{p_k}$ tensor, then by applying `Transpose`, you will get an $n_1 \times n_2 \times \ldots \times n_k$ tensor.

Here is a matrix that you can also think of as a 2×3 tensor.

```
In[7]:= m = {{a, b, c}, {ap, bp, cp}}
Out[7]= {{a, b, c}, {ap, bp, cp}}
```

Applying `Transpose` gives you a 3×2 tensor. `Transpose` effectively interchanges the two "slots" for tensor indices.

```
In[8]:= mt = Transpose[m]
Out[8]= {{a, ap}, {b, bp}, {c, cp}}
```

The element `m[[2, 3]]` in the original tensor becomes the element `m[[3, 2]]` in the transposed tensor.

```
In[9]:= {m[[2, 3]], mt[[3, 2]]}
Out[9]= {cp, cp}
```

This produces a 2×3×1×2 tensor.

```
In[10]:= t = Array[a, {2, 3, 1, 2}]
Out[10]= {{{{a[1, 1, 1, 1], a[1, 1, 1, 2]}}, {{a[1, 2, 1, 1], a[1, 2, 1, 2]}},
    {{a[1, 3, 1, 1], a[1, 3, 1, 2]}}}, {{{a[2, 1, 1, 1], a[2, 1, 1, 2]}},
    {{a[2, 2, 1, 1], a[2, 2, 1, 2]}}, {{a[2, 3, 1, 1], a[2, 3, 1, 2]}}}}
```

This transposes the first two levels of `t`.

```
In[11]:= tt1 = Transpose[t]
Out[11]= {{{{a[1, 1, 1, 1], a[1, 1, 1, 2]}}, {{a[2, 1, 1, 1], a[2, 1, 1, 2]}}},
    {{{a[1, 2, 1, 1], a[1, 2, 1, 2]}}, {{a[2, 2, 1, 1], a[2, 2, 1, 2]}}},
    {{{a[1, 3, 1, 1], a[1, 3, 1, 2]}}, {{a[2, 3, 1, 1], a[2, 3, 1, 2]}}}}
```

The result is a 3×2×1×2 tensor.

```
In[12]:= Dimensions[tt1]
Out[12]= {3, 2, 1, 2}
```

If you have a tensor that contains lists of the same length at different levels, then you can use `Transpose` to effectively collapse different levels.

This collapses all three levels, giving a list of the elements on the "main diagonal".

```
In[13]:= Transpose[Array[a, {3, 3, 3}], {1, 1, 1}]
Out[13]= {a[1, 1, 1], a[2, 2, 2], a[3, 3, 3]}
```

This collapses only the first two levels.

```
In[14]:= Transpose[Array[a, {2, 2, 2}], {1, 1}]
Out[14]= {{a[1, 1, 1], a[1, 1, 2]}, {a[2, 2, 1], a[2, 2, 2]}}
```

You can also use `Tr` to extract diagonal elements of a tensor.

This forms the ordinary trace of a rank 3 tensor.

```
In[15]:= Tr[Array[a, {3, 3, 3}]]
Out[15]= a[1, 1, 1] + a[2, 2, 2] + a[3, 3, 3]
```

Here is a generalized trace, with elements combined into a list.

```
In[16]:= Tr[Array[a, {3, 3, 3}], List]
Out[16]= {a[1, 1, 1], a[2, 2, 2], a[3, 3, 3]}
```

This combines diagonal elements only down to level 2.

```
In[17]:= Tr[Array[a, {3, 3, 3}], List, 2]
Out[17]= {{a[1, 1, 1], a[1, 1, 2], a[1, 1, 3]},
          {a[2, 2, 1], a[2, 2, 2], a[2, 2, 3]}, {a[3, 3, 1], a[3, 3, 2], a[3, 3, 3]}}
```

Outer products, and their generalizations, are a way of building higher-rank tensors from lower-rank ones. Outer products are also sometimes known as direct, tensor or Kronecker products.

From a structural point of view, the tensor you get from `Outer[`f`,` t`,` u`]` has a copy of the structure of u inserted at the "position" of each element in t. The elements in the resulting structure are obtained by combining elements of t and u using the function f.

This gives the "outer f" of two vectors. The result is a matrix.

```
In[18]:= Outer[f, {a, b}, {ap, bp}]
Out[18]= {{f[a, ap], f[a, bp]}, {f[b, ap], f[b, bp]}}
```

If you take the "outer f" of a length 3 vector with a length 2 vector, you get a 3×2 matrix.

```
In[19]:= Outer[f, {a, b, c}, {ap, bp}]
Out[19]= {{f[a, ap], f[a, bp]}, {f[b, ap], f[b, bp]}, {f[c, ap], f[c, bp]}}
```

The result of taking the "outer f" of a 2×2 matrix and a length 3 vector is a 2×2×3 tensor.

```
In[20]:= Outer[f, {{m11, m12}, {m21, m22}}, {a, b, c}]
Out[20]= {{{f[m11, a], f[m11, b], f[m11, c]}, {f[m12, a], f[m12, b], f[m12, c]}},
          {{f[m21, a], f[m21, b], f[m21, c]}, {f[m22, a], f[m22, b], f[m22, c]}}}
```

Here are the dimensions of the tensor.

```
In[21]:= Dimensions[%]
Out[21]= {2, 2, 3}
```

If you take the generalized outer product of an $m_1 \times m_2 \times \ldots \times m_r$ tensor and an $n_1 \times n_2 \times \ldots \times n_s$ tensor, you get an $m_1 \times \ldots \times m_r \times n_1 \times \ldots \times n_s$ tensor. If the original tensors have ranks r and s, your result will be a rank $r + s$ tensor.

In terms of indices, the result of applying `Outer` to two tensors $T_{i_1 i_2 \ldots i_r}$ and $U_{j_1 j_2 \ldots j_s}$ is the tensor $V_{i_1 i_2 \ldots i_r j_1 j_2 \ldots j_s}$ with elements $f[T_{i_1 i_2 \ldots i_r}, U_{j_1 j_2 \ldots j_s}]$.

In doing standard tensor calculations, the most common function f to use in `Outer` is `Times`, corresponding to the standard outer product.

Particularly in doing combinatorial calculations, however, it is often convenient to take f to be `List`. Using `Outer`, you can then get combinations of all possible elements in one tensor, with all possible elements in the other.

In constructing `Outer[f, t, u]` you effectively insert a copy of u at every point in t. To form `Inner[f, t, u]`, you effectively combine and collapse the last dimension of t and the first dimension of u. The idea is to take an $m_1 \times m_2 \times \ldots \times m_r$ tensor and an $n_1 \times n_2 \times \ldots \times n_s$ tensor, with $m_r = n_1$, and get an $m_1 \times m_2 \times \ldots \times m_{r-1} \times n_2 \times \ldots \times n_s$ tensor as the result.

The simplest examples are with vectors. If you apply `Inner` to two vectors of equal length, you get a scalar. `Inner[f, v1, v2, g]` gives a generalization of the usual scalar product, with f playing the role of multiplication, and g playing the role of addition.

This gives a generalization of the standard scalar product of two vectors.

```
In[22]:= Inner[f, {a, b, c}, {ap, bp, cp}, g]
Out[22]= g[f[a, ap], f[b, bp], f[c, cp]]
```

This gives a generalization of a matrix product.

```
In[23]:= Inner[f, {{1, 2}, {3, 4}}, {{a, b}, {c, d}}, g]
Out[23]= {{g[f[1, a], f[2, c]], g[f[1, b], f[2, d]]}, {g[f[3, a], f[4, c]], g[f[3, b], f[4, d]]}}
```

Here is a 3×2×2 tensor.

```
In[24]:= a = Array[1 &, {3, 2, 2}]
Out[24]= {{{1, 1}, {1, 1}}, {{1, 1}, {1, 1}}, {{1, 1}, {1, 1}}}
```

Here is a 2×3×1 tensor.

```
In[25]:= b = Array[2 &, {2, 3, 1}]
Out[25]= {{{2}, {2}, {2}}, {{2}, {2}, {2}}}
```

This gives a 3×2×3×1 tensor.

```
In[26]:= a.b
Out[26]= {{{{4}, {4}, {4}}, {{4}, {4}, {4}}},
          {{{4}, {4}, {4}}, {{4}, {4}, {4}}}, {{{4}, {4}, {4}}, {{4}, {4}, {4}}}}
```

Here are the dimensions of the result.

```
In[27]:= Dimensions[%]
Out[27]= {3, 2, 3, 1}
```

You can think of `Inner` as performing a "contraction" of the last index of one tensor with the first index of another. If you want to perform contractions across other pairs of indices, you can do so by first transposing the appropriate indices into the first or last position, then applying `Inner`, and then transposing the result back.

In many applications of tensors, you need to insert signs to implement antisymmetry. The function `Signature[{i1, i2, ...}]`, which gives the signature of a permutation, is often useful for this purpose.

`Outer[`f`,`t_1`,`t_2`,...]`	form a generalized outer product by combining the lowest-level elements of t_1, t_2, ...
`Outer[`f`,`t_1`,`t_2`,...,`n`]`	treat only sublists at level n as separate elements
`Outer[`f`,`t_1`,`t_2`,...,`n_1`,`n_2`,...]`	treat only sublists at level n_i in t_i as separate elements
`Inner[`f`,`t_1`,`t_2`,`g`]`	form a generalized inner product using the lowest-level elements of t_1
`Inner[`f`,`t_1`,`t_2`,`g`,`n`]`	contract index n of the first tensor with the first index of the second tensor

Treating only certain sublists in tensors as separate elements.

Here every single symbol is treated as a separate element.

```
In[28]:= Outer[f, {{i, j}, {k, l}}, {x, y}]
Out[28]= {{{f[i, x], f[i, y]}, {f[j, x], f[j, y]}}, {{f[k, x], f[k, y]}, {f[l, x], f[l, y]}}}
```

But here only sublists at level 1 are treated as separate elements.

```
In[29]:= Outer[f, {{i, j}, {k, l}}, {x, y}, 1]
Out[29]= {{f[{i, j}, x], f[{i, j}, y]}, {f[{k, l}, x], f[{k, l}, y]}}
```

`ArrayFlatten[`t`,`r`]`	create a flat rank r tensor from a rank r tensor of rank r tensors
`ArrayFlatten[`t`]`	flatten a matrix of matrices (equivalent to `ArrayFlatten[`t`, 2]`)

Flattening block tensors.

Here is a block matrix (a matrix of matrices that can be viewed as blocks that fit edge to edge within a larger matrix).

```
In[30]:= TableForm[{{ {{1, 2}, {4, 5}}, {{3}, {6}} }, { {{7, 8}}, {{9}} }}]
Out[30]//TableForm=
1 2 3
4 5 6
7 8 9
```

Here is the matrix formed by piecing the blocks together.

```
In[31]:= TableForm[ArrayFlatten[%]]
Out[31]//TableForm=
1 2 3
4 5 6
7 8 9
```

Sparse Arrays: Linear Algebra

Many large-scale applications of linear algebra involve matrices that have many elements, but comparatively few that are nonzero. You can represent such sparse matrices efficiently in *Mathematica* using `SparseArray` objects, as discussed in "Sparse Arrays: Manipulating Lists". `SparseArray` objects work by having lists of rules that specify where nonzero values appear.

`SparseArray[`*list*`]`	a `SparseArray` version of an ordinary list
`SparseArray[{{`i_1`,`j_1`}->`v_1`,{`i_2`,`j_2`}->`v_2`,...},{`m`,`n`}]`	an $m \times n$ sparse array with element $\{i_k, j_k\}$ having value v_k
`SparseArray[{{`i_1`,`j_1`},{`i_2`,`j_2`},...}->{`v_1`,`v_2`,...},{`m`,`n`}]`	the same sparse array
`Normal[`*array*`]`	the ordinary list corresponding to a `SparseArray`

Specifying sparse arrays.

As discussed in "Sparse Arrays: Manipulating Lists", you can use patterns to specify collections of elements in sparse arrays. You can also have sparse arrays that correspond to tensors of any rank.

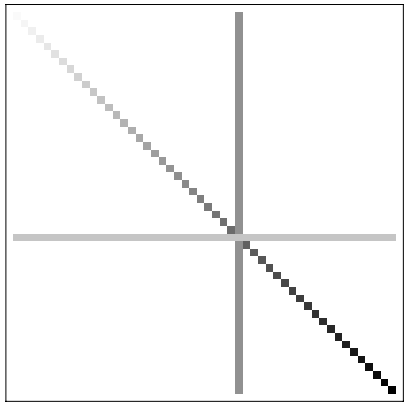
This makes a 50×50 sparse numerical matrix, with 148 nonzero elements.

```
In[1]:= m = SparseArray[{{30, _} -> 11.5, {_, 30} -> 21.5, {i_, i_} -> i}, {50, 50}]
Out[1]= SparseArray[<148>, {50, 50}]
```

This shows a visual representation of the matrix elements.

```
In[2]:= ArrayPlot[m]
```

Out[2]=



Here are the four largest eigenvalues of the matrix.

```
In[3]:= Eigenvalues[m, 4]
Out[3]= {129.846, -92.6878, 49.7867, 48.7478}
```

Dot gives a SparseArray result.

```
In[4]:= m.m
Out[4]= SparseArray[<2500>, {50, 50}]
```

You can extract parts just like in an ordinary array.

```
In[5]:= %[[20, 20]]
Out[5]= 647.25
```

You can apply most standard structural operations directly to SparseArray objects, just as you would to ordinary lists. When the results are sparse, they typically return SparseArray objects.

Dimensions[*m*]	the dimensions of an array
ArrayRules[*m*]	the rules for nonzero elements in an array
m[[*i*, *j*]]	element *i*, *j*
m[[*i*]]	the i^{th} row
m[[All, *j*]]	the i^{th} column
m[[*i*, *j*]] = *v*	reset element *i*, *j*

A few structural operations that can be done directly on SparseArray objects.

This gives the first column of m. It has only 2 nonzero elements.

```
In[6]:= m[[All, 1]]
Out[6]= SparseArray[<2>, {50}]
```

This adds 3 to each element in the first column of m.

```
In[7]:= m[[All, 1]] = 3 + m[[All, 1]]
Out[7]= SparseArray[<2>, {50}, 3]
```

Now all the elements in the first column are nonzero.

```
In[8]:= m[[All, 1]]
Out[8]= SparseArray[<50>, {50}]
```

This gives the rules for the nonzero elements on the second row.

```
In[9]:= ArrayRules[m[[2]]]
Out[9]= {{1} → 3, {2} → 2, {30} → 21.5, {_} → 0}
```

`SparseArray[`*rules*`]`	generate a sparse array from rules
`CoefficientArrays[{`*eqns*$_1$`,`*eqns*$_2$`,…},{`x_1`,`x_2`,…}]`	get arrays of coefficients from equations
`Import["`*file*`.mtx"]`	import a sparse array from a file

Typical ways to get sparse arrays.

This generates a tridiagonal random matrix.

```
In[10]:= SparseArray[{i_, j_} /; Abs[i - j] <= 1 :> RandomReal[], {100, 100}]
Out[10]= SparseArray[<298>, {100, 100}]
```

Even the tenth power of the matrix is still fairly sparse.

```
In[11]:= MatrixPower[%, 10]
Out[11]= SparseArray[<1990>, {100, 100}]
```

This extracts the coefficients as sparse arrays.

```
In[12]:= s = CoefficientArrays[{c + x - z == 0, x + 2 y + z == 0}, {x, y, z}]
Out[12]= {SparseArray[<1>, {2}], SparseArray[<5>, {2, 3}]}
```

Here are the corresponding ordinary arrays.

```
In[13]:= Normal[%]
Out[13]= {{c, 0}, {{1, 0, -1}, {1, 2, 1}}}
```

This reproduces the original forms.

```
In[14]:= s[[1]] + s[[2]].{x, y, z}
Out[14]= {c + x - z, x + 2 y + z}
```

`CoefficientArrays` can handle general polynomial equations.

```
In[15]:= s = CoefficientArrays[{c + x^2 - z == 0, x^2 + 2 y + z^2 == 0}, {x, y, z}]
Out[15]= {SparseArray[<1>, {2}], SparseArray[<2>, {2, 3}], SparseArray[<3>, {2, 3, 3}]}
```

The coefficients of the quadratic part are given in a rank 3 tensor.

```
In[16]:= Normal[%]
Out[16]= {{c, 0}, {{0, 0, -1}, {0, 2, 0}},
          {{{1, 0, 0}, {0, 0, 0}, {0, 0, 0}}, {{1, 0, 0}, {0, 0, 0}, {0, 0, 1}}}}
```

This reproduces the original forms.

```
In[17]:= s[[1]] + s[[2]].{x, y, z} + s[[3]].{x, y, z}.{x, y, z}
Out[17]= {c + x^2 - z, x^2 + 2 y + z^2}
```

For machine-precision numerical sparse matrices, *Mathematica* supports standard file formats such as Matrix Market (`.mtx`) and Harwell-Boeing. You can import and export matrices in these formats using `Import` and `Export`.

Series, Limits and Residues

Sums and Products

This constructs the sum $\sum_{i=1}^{7} \frac{x^i}{i}$.

```
In[1]:= Sum[x^i / i, {i, 1, 7}]
Out[1]= x + x^2/2 + x^3/3 + x^4/4 + x^5/5 + x^6/6 + x^7/7
```

You can leave out the lower limit if it is equal to 1.

```
In[2]:= Sum[x^i / i, {i, 7}]
Out[2]= x + x^2/2 + x^3/3 + x^4/4 + x^5/5 + x^6/6 + x^7/7
```

This makes i increase in steps of 2, so that only odd-numbered values are included.

```
In[3]:= Sum[x^i / i, {i, 1, 5, 2}]
Out[3]= x + x^3/3 + x^5/5
```

Products work just like sums.

```
In[4]:= Product[x + i, {i, 1, 4}]
Out[4]= (1 + x) (2 + x) (3 + x) (4 + x)
```

`Sum[`f`,{`i`,`i_{min}`,`i_{max}`}]`	the sum $\sum_{i=i_{min}}^{i_{max}} f$
`Sum[`f`,{`i`,`i_{min}`,`i_{max}`,`di`}]`	the sum with i increasing in steps of di
`Sum[`f`,{`i`,`i_{min}`,`i_{max}`},{`j`,`j_{min}`,`j_{max}`}]`	the nested sum $\sum_{i=i_{min}}^{i_{max}} \sum_{j=j_{min}}^{j_{max}} f$
`Product[`f`,{`i`,`i_{min}`,`i_{max}`}]`	the product $\prod_{i=i_{min}}^{i_{max}} f$

Sums and products.

This sum is computed symbolically as a function of `n`.

```
In[5]:= Sum[i^2, {i, 1, n}]
Out[5]= 1/6 n (1 + n) (1 + 2 n)
```

Mathematica can also give an exact result for this infinite sum.

```
In[6]:= Sum[1 / i^4, {i, 1, Infinity}]
Out[6]= π^4/90
```

As with integrals, simple sums can lead to complicated results.

```
In[7]:= Sum[x^(i (i + 1)), {i, 1, Infinity}]
Out[7]= (-2 x^(1/4) + EllipticTheta[2, 0, x])/(2 x^(1/4))
```

This sum cannot be evaluated exactly using standard mathematical functions.

```
In[8]:= Sum[1 / (i! + (2 i)!), {i, 1, Infinity}]
Out[8]= Sum[1/(i! + (2 i)!), {i, 1, ∞}]
```

You can nevertheless find a numerical approximation to the result.

```
In[9]:= N[%]
Out[9]= 0.373197
```

Mathematica also has a notation for multiple sums and products. `Sum[f, {i, imin, imax}, {j, jmin, jmax}]` represents a sum over i and j, which would be written in standard mathematical notation as $\sum_{i=i_{min}}^{i_{max}} \sum_{j=j_{min}}^{j_{max}} f$. Notice that in *Mathematica* notation, as in standard mathematical notation, the range of the *outermost* variable is given *first*.

This is the multiple sum $\sum_{i=1}^{3} \sum_{j=1}^{i} x^i y^j$. Notice that the outermost sum over `i` is given first, just as in the mathematical notation.

```
In[10]:= Sum[x^i y^j, {i, 1, 3}, {j, 1, i}]
```

$$\text{Out[10]= } x\,y + x^2\,y + x^3\,y + x^2\,y^2 + x^3\,y^2 + x^3\,y^3$$

The way the ranges of variables are specified in `Sum` and `Product` is an example of the rather general *iterator notation* that *Mathematica* uses. You will see this notation again when we discuss generating tables and lists using `Table` ("Making Tables of Values"), and when we describe `Do` loops ("Repetitive Operations").

$\{i_{max}\}$	iterate i_{max} times, without incrementing any variables
$\{i, i_{max}\}$	i goes from 1 to i_{max} in steps of 1
$\{i, i_{min}, i_{max}\}$	i goes from i_{min} to i_{max} in steps of 1
$\{i, i_{min}, i_{max}, di\}$	i goes from i_{min} to i_{max} in steps of di
$\{i, i_{min}, i_{max}\}, \{j, j_{min}, j_{max}\}, \ldots$	i goes from i_{min} to i_{max}, and for each such value, j goes from j_{min} to j_{max}, etc.

Mathematica iterator notation.

Power Series

The mathematical operations we have discussed so far are *exact*. Given precise input, their results are exact formulas.

In many situations, however, you do not need an exact result. It may be quite sufficient, for example, to find an *approximate* formula that is valid, say, when the quantity `x` is small.

This gives a power series approximation to $(1 + x)^n$ for x close to 0, up to terms of order x^3.

```
In[1]:= Series[(1 + x)^n, {x, 0, 3}]
```

$$\text{Out[1]= } 1 + n\,x + \frac{1}{2}(-1 + n)\,n\,x^2 + \frac{1}{6}(-2 + n)(-1 + n)\,n\,x^3 + O[x]^4$$

Mathematica knows the power series expansions for many mathematical functions.

```
In[2]:= Series[Exp[-a t] (1 + Sin[2 t]), {t, 0, 4}]
```

$$\text{Out[2]}= 1 + (2 - a)\, t + \left(-2\, a + \frac{a^2}{2}\right) t^2 + \left(-\frac{4}{3} + a^2 - \frac{a^3}{6}\right) t^3 + \frac{1}{24}\left(32\, a - 8\, a^3 + a^4\right) t^4 + O[t]^5$$

If you give it a function that it does not know, `Series` writes out the power series in terms of derivatives.

```
In[3]:= Series[1 + f[t], {t, 0, 3}]
```

$$\text{Out[3]}= (1 + f[0]) + f'[0]\, t + \frac{1}{2} f''[0]\, t^2 + \frac{1}{6} f^{(3)}[0]\, t^3 + O[t]^4$$

Power series are approximate formulas that play much the same role with respect to algebraic expressions as approximate numbers play with respect to numerical expressions. *Mathematica* allows you to perform operations on power series, in all cases maintaining the appropriate order or "degree of precision" for the resulting power series.

Here is a simple power series, accurate to order x^5.

```
In[4]:= Series[Exp[x], {x, 0, 5}]
```

$$\text{Out[4]}= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O[x]^6$$

When you do operations on a power series, the result is computed only to the appropriate order in `x`.

```
In[5]:= %^2 (1 + %)
```

$$\text{Out[5]}= 2 + 5\, x + \frac{13\, x^2}{2} + \frac{35\, x^3}{6} + \frac{97\, x^4}{24} + \frac{55\, x^5}{24} + O[x]^6$$

This turns the power series back into an ordinary expression.

```
In[6]:= Normal[%]
```

$$\text{Out[6]}= 2 + 5\, x + \frac{13\, x^2}{2} + \frac{35\, x^3}{6} + \frac{97\, x^4}{24} + \frac{55\, x^5}{24}$$

Now the square is computed *exactly*.

```
In[7]:= %^2
```

$$\text{Out[7]}= \left(2 + 5\, x + \frac{13\, x^2}{2} + \frac{35\, x^3}{6} + \frac{97\, x^4}{24} + \frac{55\, x^5}{24}\right)^2$$

Applying `Expand` gives a result with 11 terms.

```
In[8]:= Expand[%]
```

$$\text{Out[8]}= 4 + 20\,x + 51\,x^2 + \frac{265\,x^3}{3} + \frac{467\,x^4}{4} + \frac{1505\,x^5}{12} + \frac{7883\,x^6}{72} + \frac{1385\,x^7}{18} + \frac{24\,809\,x^8}{576} + \frac{5335\,x^9}{288} + \frac{3025\,x^{10}}{576}$$

`Series[`*expr*`,{`*x*`,`x_0`,`*n*`}]`	find the power series expansion of *expr* about the point $x = x_0$ to at most n^{th} order
`Normal[`*series*`]`	truncate a power series to give an ordinary expression

Power series operations.

Making Power Series Expansions

`Series[`*expr*`,{`*x*`,`x_0`,`*n*`}]`	find the power series expansion of *expr* about the point $x = x_0$ to order at most $(x - x_0)^n$
`Series[`*expr*`,{`*x*`,`x_0`,`n_x`},{`*y*`,`y_0`,`n_y`}]`	find series expansions with respect to y then x

Functions for creating power series.

Here is the power series expansion for $\exp(x)$ about the point $x = 0$ to order x^4.

```
In[1]:= Series[Exp[x], {x, 0, 4}]
```

$$\text{Out[1]}= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + O[x]^5$$

Here is the series expansion of $\exp(x)$ about the point $x = 1$.

```
In[2]:= Series[Exp[x], {x, 1, 4}]
```

$$\text{Out[2]}= e + e\,(x-1) + \frac{1}{2}e\,(x-1)^2 + \frac{1}{6}e\,(x-1)^3 + \frac{1}{24}e\,(x-1)^4 + O[x-1]^5$$

If *Mathematica* does not know the series expansion of a particular function, it writes the result symbolically in terms of derivatives.

```
In[3]:= Series[f[x], {x, 0, 3}]
```

$$\text{Out[3]}= f[0] + f'[0]\,x + \frac{1}{2}f''[0]\,x^2 + \frac{1}{6}f^{(3)}[0]\,x^3 + O[x]^4$$

In mathematical terms, `Series` can be viewed as a way of constructing Taylor series for functions.

The standard formula for the Taylor series expansion about the point $x = x_0$ of a function $g(x)$ with k^{th} derivative $g^{(k)}(x)$ is $g(x) = \sum_{k=0}^{\infty} g^{(k)}(x_0) \frac{(x-x_0)^k}{k!}$. Whenever this formula applies, it gives the same results as `Series`. (For common functions, `Series` nevertheless internally uses somewhat more efficient algorithms.)

`Series` can also generate some power series that involve fractional and negative powers, not directly covered by the standard Taylor series formula.

Here is a power series that contains negative powers of x.

```
In[4]:= Series[Exp[x] / x^2, {x, 0, 4}]
```

$$\text{Out[4]}= \frac{1}{x^2} + \frac{1}{x} + \frac{1}{2} + \frac{x}{6} + \frac{x^2}{24} + \frac{x^3}{120} + \frac{x^4}{720} + O[x]^5$$

Here is a power series involving fractional powers of x.

```
In[5]:= Series[Exp[Sqrt[x]], {x, 0, 2}]
```

$$\text{Out[5]}= 1 + \sqrt{x} + \frac{x}{2} + \frac{x^{3/2}}{6} + \frac{x^2}{24} + O[x]^{5/2}$$

`Series` can also handle series that involve logarithmic terms.

```
In[6]:= Series[Exp[2 x] Log[x], {x, 0, 2}]
```

```
Out[6]= Log[x] + 2 Log[x] x + 2 Log[x] x^2 + O[x]^3
```

There are, of course, mathematical functions for which no standard power series exist. *Mathematica* recognizes many such cases.

`Series` sees that $\exp\left(\frac{1}{x}\right)$ has an essential singularity at $x = 0$, and does not produce a power series.

```
In[7]:= Series[Exp[1 / x], {x, 0, 2}]
```

$$\text{Out[7]}= e^{\frac{1}{x}}$$

`Series` can nevertheless give you the power series for $\exp\left(\frac{1}{x}\right)$ about the point $x = \infty$.

```
In[8]:= Series[Exp[1 / x], {x, Infinity, 3}]
```

$$\text{Out[8]}= 1 + \frac{1}{x} + \frac{1}{2}\left(\frac{1}{x}\right)^2 + \frac{1}{6}\left(\frac{1}{x}\right)^3 + O\left[\frac{1}{x}\right]^4$$

Especially when negative powers occur, there is some subtlety in exactly how many terms of a particular power series the function `Series` will generate.

One way to understand what happens is to think of the analogy between power series taken to a certain order, and real numbers taken to a certain precision. Power series are "approximate formulas" in much the same sense as finite-precision real numbers are approximate numbers.

The procedure that `Series` follows in constructing a power series is largely analogous to the procedure that `N` follows in constructing a real-number approximation. Both functions effectively start by replacing the smallest pieces of your expression by finite-order, or finite-precision, approximations, and then evaluating the resulting expression. If there are, for example, cancellations, this procedure may give a final result whose order or precision is less than the order or precision that you originally asked for. Like `N`, however, `Series` has some ability to retry its computations so as to get results to the order you ask for. In cases where it does not succeed, you can usually still get results to a particular order by asking for a higher order than you need.

`Series` compensates for cancellations in this computation, and succeeds in giving you a result to order x^3.

```
In[9]:= Series[Sin[x] / x^2, {x, 0, 3}]
```

$$\text{Out[9]}= \frac{1}{x} - \frac{x}{6} + \frac{x^3}{120} + O[x]^4$$

When you make a power series expansion in a variable x, *Mathematica* assumes that all objects that do not explicitly contain x are in fact independent of x. `Series` thus does partial derivatives (effectively using `D`) to build up Taylor series.

Both `a` and `n` are assumed to be independent of `x`.

```
In[10]:= Series[(a + x)^n, {x, 0, 2}]
```

$$\text{Out[10]}= a^n + a^{-1+n}\, n\, x + \frac{1}{2} a^{-2+n}\, (-1+n)\, n\, x^2 + O[x]^3$$

`a[x]` is now given as an explicit function of `x`.

```
In[11]:= Series[(a[x] + x)^n, {x, 0, 2}]
```

$$\text{Out[11]= } a[0]^n + n\,a[0]^{-1+n}\,(1 + a'[0])\,x + \left(\frac{1}{2}\,(-1+n)\,n\,a[0]^{-2+n}\,(1+a'[0])^2 + \frac{1}{2}\,n\,a[0]^{-1+n}\,a''[0]\right)x^2 + O[x]^3$$

You can use `Series` to generate power series in a sequence of different variables. `Series` works like `Integrate`, `Sum` and so on, and expands first with respect to the last variable you specify.

`Series` performs a series expansion successively with respect to each variable. The result in this case is a series in `x`, whose coefficients are series in `y`.

```
In[12]:= Series[Exp[x y], {x, 0, 3}, {y, 0, 3}]
```

$$\text{Out[12]= } 1 + \left(y + O[y]^4\right)x + \left(\frac{y^2}{2} + O[y]^4\right)x^2 + \left(\frac{y^3}{6} + O[y]^4\right)x^3 + O[x]^4$$

The Representation of Power Series

Power series are represented in *Mathematica* as `SeriesData` objects.

The power series is printed out as a sum of terms, ending with `O[x]` raised to a power.

```
In[1]:= Series[Cos[x], {x, 0, 4}]
```

$$\text{Out[1]= } 1 - \frac{x^2}{2} + \frac{x^4}{24} + O[x]^5$$

Internally, however, the series is stored as a `SeriesData` object.

```
In[2]:= InputForm[%]
```

```
Out[2]//InputForm= SeriesData[x, 0, {1, 0, -1/2, 0, 1/24}, 0, 5, 1]
```

By using `SeriesData` objects, rather than ordinary expressions, to represent power series, *Mathematica* can keep track of the order and expansion point, and do operations on the power series appropriately. You should not normally need to know the internal structure of `SeriesData` objects.

You can recognize a power series that is printed out in standard output form by the presence of an `O[x]` term. This term mimics the standard mathematical notation $O(x)$, and represents omitted terms of order x. For various reasons of consistency, *Mathematica* uses the notation `O[x]^n` for omitted terms of order x^n, corresponding to the mathematical notation $O(x)^n$, rather than the slightly more familiar, though equivalent, form $O(x^n)$.

Any time that an object like $O[x]$ appears in a sum of terms, *Mathematica* will in fact convert the whole sum into a power series.

The presence of `O[x]` makes *Mathematica* convert the whole sum to a power series.

```
In[3]:= a x + Exp[x] + O[x]^3
```

$$\text{Out[3]}= 1 + (1 + a)\, x + \frac{x^2}{2} + O[x]^3$$

Series objects can involve fractional powers.

```
In[4]:= Series[Sqrt[1 - x^3], {x, 1, 5}]
```

$$\text{Out[4]}= i\sqrt{3}\ \sqrt{x-1} + \frac{1}{2} i \sqrt{3}\ (x-1)^{3/2} + \frac{i\,(x-1)^{5/2}}{8\sqrt{3}} + -\frac{i\,(x-1)^{7/2}}{16\sqrt{3}} + \frac{11\,i\,(x-1)^{9/2}}{384\sqrt{3}} + O[x-1]^{11/2}$$

Here is the series' internal representation.

```
In[5]:= % // InputForm
```

```
Out[5]//InputForm= SeriesData[x, 1, {I*Sqrt[3], 0, (I/2)*Sqrt[3], 0, (I/8)/Sqrt[3], 0, (-I/16)/Sqrt[3], 0, ((
                     2]
```

Series can involve logarithmic terms.

```
In[6]:= Series[x^x, {x, 0, 4}]
```

$$\text{Out[6]}= 1 + \text{Log}[x]\, x + \frac{1}{2}\text{Log}[x]^2 x^2 + \frac{1}{6}\text{Log}[x]^3 x^3 + \frac{1}{24}\text{Log}[x]^4 x^4 + O[x]^5$$

The logarithmic factors appear explicitly inside the `SeriesData` coefficient list.

```
In[7]:= % // InputForm
```

```
Out[7]//InputForm= SeriesData[x, 0, {1, Log[x], Log[x]^2/2, Log[x]^3/6, Log[x]^4/24}, 0, 5, 1]
```

Operations on Power Series

Mathematica allows you to perform many operations on power series. In all cases, *Mathematica* gives results only to as many terms as can be justified from the accuracy of your input.

Here is a power series accurate to fourth order in x.

```
In[1]:= Series[Exp[x], {x, 0, 4}]
```

$$\text{Out[1]}= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + O[x]^5$$

When you square the power series, you get another power series, also accurate to fourth order.

```
In[2]:= %^2
```

$$\text{Out[2]}= 1 + 2\,x + 2\,x^2 + \frac{4\,x^3}{3} + \frac{2\,x^4}{3} + O[x]^5$$

Taking the logarithm gives you the result $2\,x$, but only to order x^4.

```
In[3]:= Log[%]
```

$$\text{Out[3]}= 2\,x + O[x]^5$$

Mathematica keeps track of the orders of power series in much the same way as it keeps track of the precision of approximate real numbers. Just as with numerical calculations, there are operations on power series which can increase, or decrease, the precision (or order) of your results.

Here is a power series accurate to order x^{10}.

```
In[4]:= Series[Cos[x], {x, 0, 10}]
```

$$\text{Out[4]}= 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40\,320} - \frac{x^{10}}{3\,628\,800} + O[x]^{11}$$

This gives a power series that is accurate only to order x^6.

```
In[5]:= 1 / (1 - %)
```

$$\text{Out[5]}= \frac{2}{x^2} + \frac{1}{6} + \frac{x^2}{120} + \frac{x^4}{3024} + \frac{x^6}{86\,400} + O[x]^7$$

Mathematica also allows you to do calculus with power series.

Here is a power series for $\tan(x)$.

```
In[6]:= Series[Tan[x], {x, 0, 10}]
```

$$\text{Out[6]}= x + \frac{x^3}{3} + \frac{2\,x^5}{15} + \frac{17\,x^7}{315} + \frac{62\,x^9}{2835} + O[x]^{11}$$

Here is its derivative with respect to x.

```
In[7]:= D[%, x]
```

$$\text{Out[7]}= 1 + x^2 + \frac{2\,x^4}{3} + \frac{17\,x^6}{45} + \frac{62\,x^8}{315} + O[x]^{10}$$

Integrating with respect to `x` gives back the original power series.

```
In[8]:= Integrate[%, x]
```

$$\text{Out[8]}= x + \frac{x^3}{3} + \frac{2\,x^5}{15} + \frac{17\,x^7}{315} + \frac{62\,x^9}{2835} + O[x]^{11}$$

When you perform an operation that involves both a normal expression and a power series, *Mathematica* "absorbs" the normal expression into the power series whenever possible.

The `1` is automatically absorbed into the power series.

```
In[9]:= 1 + Series[Exp[x], {x, 0, 4}]
```

$$\text{Out[9]}= 2 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + O[x]^5$$

The `x^2` is also absorbed into the power series.

```
In[10]:= % + x^2
```

$$\text{Out[10]}= 2 + x + \frac{3\,x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + O[x]^5$$

If you add `Sin[x]`, *Mathematica* generates the appropriate power series for `Sin[x]`, and combines it with the power series you have.

```
In[11]:= % + Sin[x]
```

$$\text{Out[11]}= 2 + 2\,x + \frac{3\,x^2}{2} + \frac{x^4}{24} + O[x]^5$$

Mathematica also absorbs expressions that multiply power series. The symbol `a` is assumed to be independent of `x`.

```
In[12]:= (a + x) %^2
```

$$\text{Out[12]}= 4\,a + (4 + 8\,a)\,x + (8 + 10\,a)\,x^2 + (10 + 6\,a)\,x^3 + \left(6 + \frac{29\,a}{12}\right) x^4 + O[x]^5$$

Mathematica knows how to apply a wide variety of functions to power series. However, if you apply an arbitrary function to a power series, it is impossible for *Mathematica* to give you anything but a symbolic result.

Mathematica does not know how to apply the function `f` to a power series, so it just leaves the symbolic result.

```
In[13]:= f[Series[Exp[x], {x, 0, 3}]]
```

$$\text{Out[13]}= f\left[1 + x + \frac{x^2}{2} + \frac{x^3}{6} + O[x]^4\right]$$

Composition and Inversion of Power Series

When you manipulate power series, it is sometimes convenient to think of the series as representing *functions*, which you can, for example, compose or invert.

`ComposeSeries[`$series_1$`,`$series_2$`,...]`	compose power series
`InverseSeries[`$series$`,`x`]`	invert a power series

Composition and inversion of power series.

Here is the power series for $\exp(x)$ to order x^5.

```
In[1]:= Series[Exp[x], {x, 0, 5}]
```

$$\text{Out[1]}= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O[x]^6$$

This replaces the variable x in the power series for $\exp(x)$ by a power series for $\sin(x)$.

```
In[2]:= ComposeSeries[%, Series[Sin[x], {x, 0, 5}]]
```

$$\text{Out[2]}= 1 + x + \frac{x^2}{2} - \frac{x^4}{8} - \frac{x^5}{15} + O[x]^6$$

The result is the power series for $\exp(\sin(x))$.

```
In[3]:= Series[Exp[Sin[x]], {x, 0, 5}]
```

$$\text{Out[3]}= 1 + x + \frac{x^2}{2} - \frac{x^4}{8} - \frac{x^5}{15} + O[x]^6$$

If you have a power series for a function $f(y)$, then it is often possible to get a power series approximation to the solution for y in the equation $f(y) = x$. This power series effectively gives the inverse function $f^{-1}(x)$ such that $f\left(f^{-1}(x)\right) = x$. The operation of finding the power series for an inverse function is sometimes known as *reversion* of power series.

Here is the series for $\sin(y)$.

```
In[4]:= Series[Sin[y], {y, 0, 5}]
```

$$\text{Out[4]}= y - \frac{y^3}{6} + \frac{y^5}{120} + O[y]^6$$

Inverting the series gives the series for $\sin^{-1}(x)$.

```
In[5]:= InverseSeries[%, x]
```

$$\text{Out[5]}= x + \frac{x^3}{6} + \frac{3\,x^5}{40} + O[x]^6$$

This agrees with the direct series for $\sin^{-1}(x)$.

```
In[6]:= Series[ArcSin[x], {x, 0, 5}]
```

$$\text{Out[6]}= x + \frac{x^3}{6} + \frac{3\,x^5}{40} + O[x]^6$$

Composing the series with its inverse gives the identity function.

```
In[7]:= ComposeSeries[%, %%%]
```

$$\text{Out[7]}= y + O[y]^6$$

Converting Power Series to Normal Expressions

`Normal[`*expr*`]`	convert a power series to a normal expression

Converting power series to normal expressions.

Power series in *Mathematica* are represented in a special internal form, which keeps track of such attributes as their expansion order.

For some purposes, you may want to convert power series to normal expressions. From a mathematical point of view, this corresponds to truncating the power series, and assuming that all higher-order terms are zero.

This generates a power series, with four terms.

```
In[1]:= t = Series[ArcTan[x], {x, 0, 8}]
```

$$\text{Out[1]}= x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + O[x]^9$$

Squaring the power series gives you another power series, with the appropriate number of terms.

```
In[2]:= t^2
```

$$\text{Out[2]}= x^2 - \frac{2\,x^4}{3} + \frac{23\,x^6}{45} - \frac{44\,x^8}{105} + O[x]^{10}$$

`Normal` truncates the power series, giving a normal expression.

```
In[3]:= Normal[%]
```

$$\text{Out[3]}= x^2 - \frac{2x^4}{3} + \frac{23x^6}{45} - \frac{44x^8}{105}$$

You can now apply standard algebraic operations.

```
In[4]:= Factor[%]
```

$$\text{Out[4]}= -\frac{1}{315}x^2\left(-315 + 210x^2 - 161x^4 + 132x^6\right)$$

`SeriesCoefficient[`*series*`,`*n*`]`	give the coefficient of the n^{th}-order term in a power series

Extracting coefficients of terms in power series.

This gives the coefficient of x^7 in the original power series.

```
In[5]:= SeriesCoefficient[t, 7]
```

$$\text{Out[5]}= -\frac{1}{7}$$

This gives the coefficient for the term x^n in the Taylor expansion of the function e^{x^2} about zero.

```
In[6]:= SeriesCoefficient[E^x^2, {x, 0, n}]
```

$$\text{Out[6]}= \frac{\text{KroneckerDelta[Mod[n, 2]]}}{\frac{n}{2}!}$$

Solving Equations Involving Power Series

`LogicalExpand[`*series*$_1$`==`*series*$_2$`]`	give the equations obtained by equating corresponding coefficients in the power series
`Solve[`*series*$_1$`==`*series*$_2$`,{`a_1`,`a_2`,...}]`	solve for coefficients in power series

Solving equations involving power series.

Here is a power series.

```
In[1]:= y = 1 + Sum[a[i] x^i, {i, 3}] + O[x]^4
```

$$\text{Out[1]}= 1 + a[1]\,x + a[2]\,x^2 + a[3]\,x^3 + O[x]^4$$

This gives an equation involving the power series.

```
In[2]:= D[y, x]^2 - y == x
```

$$\text{Out[2]}= \left(-1+a[1]^2\right)+\left(-a[1]+4\,a[1]\,a[2]\right)x+\left(-a[2]+4\,a[2]^2+6\,a[1]\,a[3]\right)x^2+O[x]^3==x$$

`LogicalExpand` generates a sequence of equations for each power of `x`.

```
In[3]:= LogicalExpand[%]
```

$$\text{Out[3]}= -1+a[1]^2==0\ \&\&\ -1-a[1]+4\,a[1]\,a[2]==0\ \&\&\ -a[2]+4\,a[2]^2+6\,a[1]\,a[3]==0$$

This solves the equations for the coefficients `a[`i`]`. You can also feed equations involving power series directly to `Solve`.

```
In[4]:= Solve[%]
```

$$\text{Out[4]}= \left\{\left\{a[3]\to -\frac{1}{12},\ a[1]\to 1,\ a[2]\to \frac{1}{2}\right\},\ \{a[3]\to 0,\ a[1]\to -1,\ a[2]\to 0\}\right\}$$

Some equations involving power series can also be solved using the `InverseSeries` function discussed in "Composition and Inversion of Power Series".

Summation of Series

`Sum[`*expr*`,{`n`,`n_{min}`,`n_{max}`}]`	find the sum of *expr* as n goes from n_{min} to n_{max}

Evaluating sums.

Mathematica recognizes this as the power series expansion of e^x.

```
In[1]:= Sum[x^n / n!, {n, 0, Infinity}]
```

$$\text{Out[1]}= e^x$$

This sum comes out in terms of a Bessel function.

```
In[2]:= Sum[x^n / (n!^2), {n, 0, Infinity}]
```

$$\text{Out[2]}= \text{BesselI}\left[0,\ 2\sqrt{x}\right]$$

Here is another sum that can be done in terms of common special functions.

```
In[3]:= Sum[n! x^n / (2 n)!, {n, 1, Infinity}]
```

$$\text{Out[3]}= \frac{1}{2}e^{x/4}\sqrt{\pi}\ \sqrt{x}\ \text{Erf}\left[\frac{\sqrt{x}}{2}\right]$$

Generalized hypergeometric functions are not uncommon in sums.

```
In[4]:= Sum[x^n / (n!^4), {n, 0, Infinity}]
Out[4]= HypergeometricPFQ[{}, {1, 1, 1}, x]
```

There are many analogies between sums and integrals. And just as it is possible to have indefinite integrals, so indefinite sums can be set up by using symbolic variables as upper limits.

This is effectively an indefinite sum.

```
In[5]:= Sum[k, {k, 0, n}]
```

$$\text{Out[5]}= \frac{1}{2} n (1 + n)$$

This sum comes out in terms of incomplete gamma functions.

```
In[6]:= Sum[x^k / k!, {k, 0, n}]
```

$$\text{Out[6]}= \frac{e^x (1 + n)\ \text{Gamma}[1 + n, x]}{\text{Gamma}[2 + n]}$$

This sum involves polygamma functions.

```
In[7]:= Sum[1 / (k + 1)^4, {k, 0, n}]
```

$$\text{Out[7]}= \frac{\pi^4}{90} - \frac{1}{6}\ \text{PolyGamma}[3, 2 + n]$$

Taking the difference between results for successive values of n gives back the original summand.

```
In[8]:= FullSimplify[% - (% /. n -> n - 1)]
```

$$\text{Out[8]}= \frac{1}{(1 + n)^4}$$

Mathematica can do essentially all sums that are found in books of tables. Just as with indefinite integrals, indefinite sums of expressions involving simple functions tend to give answers that involve more complicated functions. Definite sums, like definite integrals, often, however, come out in terms of simpler functions.

This indefinite sum gives a quite complicated result.

```
In[9]:= Sum[Binomial[2 k, k] / 3^(2 k), {k, 0, n}]
```

$$\text{Out[9]}= \frac{3}{\sqrt{5}} - \frac{\left(\frac{9}{4}\right)^{-1-n} \text{Gamma}\left[\frac{3}{2} + n\right] \text{Hypergeometric2F1}\left[1, \frac{3}{2} + n, 2 + n, \frac{4}{9}\right]}{\sqrt{\pi}\ \text{Gamma}[2 + n]}$$

The definite form is much simpler.

```
In[10]:= Sum[Binomial[2 k, k] / 3^(2 k), {k, 0, Infinity}]
```

$$\text{Out[10]}= \frac{3}{\sqrt{5}}$$

Here is a slightly more complicated definite sum.

```
In[11]:= Sum[PolyGamma[k] / k^2, {k, 1, Infinity}]
```

$$\text{Out[11]}= \frac{1}{6}\left(-\text{EulerGamma}\,\pi^2 + 6\,\text{Zeta}[3]\right)$$

Solving Recurrence Equations

If you represent the n^{th} term in a sequence as $a[n]$, you can use a *recurrence equation* to specify how it is related to other terms in the sequence.

`RSolve` takes recurrence equations and solves them to get explicit formulas for $a[n]$.

This solves a simple recurrence equation.

```
In[1]:= RSolve[{a[n] == 2 a[n - 1], a[1] == 1}, a[n], n]
```

$$\text{Out[1]}= \left\{\left\{\text{a}[\text{n}] \to 2^{-1+\text{n}}\right\}\right\}$$

This takes the solution and makes an explicit table of the first ten `a[n]`.

```
In[2]:= Table[a[n] /. First[%], {n, 10}]
Out[2]= {1, 2, 4, 8, 16, 32, 64, 128, 256, 512}
```

`RSolve[`eqn`,`$a[n]$`,`n`]`	solve a recurrence equation

Solving a recurrence equation.

This solves a recurrence equation for a geometric series.

```
In[3]:= RSolve[{a[n] == r a[n - 1] + 1, a[1] == 1}, a[n], n]
```

$$\text{Out[3]}= \left\{\left\{\text{a}[\text{n}] \to \frac{-1+\text{r}^{\text{n}}}{-1+\text{r}}\right\}\right\}$$

This gives the same result.

```
In[4]:= RSolve[{a[n + 1] == r a[n] + 1, a[1] == 1}, a[n], n]
```

$$\text{Out[4]}= \left\{\left\{a[n] \to \frac{-1 + r^n}{-1 + r}\right\}\right\}$$

This gives an algebraic solution to a recurrence equation.

```
In[5]:= RSolve[{a[n] == 4 a[n - 1] + a[n - 2]}, a[n], n]
```

$$\text{Out[5]}= \left\{\left\{a[n] \to \left(2 - \sqrt{5}\right)^n C[1] + \left(2 + \sqrt{5}\right)^n C[2]\right\}\right\}$$

This solves the Fibonacci recurrence equation.

```
In[6]:= RSolve[{a[n] == a[n - 1] + a[n - 2], a[1] == a[2] == 1}, a[n], n]
Out[6]= {{a[n] → Fibonacci[n]}}
```

`RSolve` can be thought of as a discrete analog of `DSolve`. Many of the same functions generated in solving differential equations also appear in finding symbolic solutions to recurrence equations.

This generates a gamma function, which generalizes the factorial.

```
In[7]:= RSolve[{a[n] == n a[n - 1], a[1] == 1}, a[n], n]
Out[7]= {{a[n] → Gamma[1 + n]}}
```

This second-order recurrence equation comes out in terms of Bessel functions.

```
In[8]:= RSolve[{a[n + 1] == n a[n] + a[n - 1], a[1] == 0, a[2] == 1}, a[n], n]
```

$$\text{Out[8]}= \left\{\left\{a[n] \to \frac{\text{BesselI}[n, -2]\,\text{BesselK}[1, 2] + \text{BesselI}[1, 2]\,\text{BesselK}[n, 2]}{\text{BesselI}[2, 2]\,\text{BesselK}[1, 2] + \text{BesselI}[1, 2]\,\text{BesselK}[2, 2]}\right\}\right\}$$

`RSolve` does not require you to specify explicit values for terms such as `a[1]`. Like `DSolve`, it automatically introduces undetermined constants `C[`i`]` to give a general solution.

This gives a general solution with one undetermined constant.

```
In[9]:= RSolve[a[n] == n a[n - 1], a[n], n]
Out[9]= {{a[n] → C[1] Gamma[1 + n]}}
```

`RSolve` can solve equations that do not depend only linearly on $a[n]$. For nonlinear equations, however, there are sometimes several distinct solutions that must be given. Just as for differential equations, it is a difficult matter to find symbolic solutions to recurrence equations, and standard mathematical functions only cover a limited set of cases.

Here is the general solution to a nonlinear recurrence equation.

```
In[10]:= RSolve[{a[n] == a[n + 1] a[n - 1]}, a[n], n]
```

$$\text{Out[10]= } \left\{\left\{a[n] \to e^{C[1] \cos\left[\frac{n\pi}{3}\right] + C[2] \sin\left[\frac{n\pi}{3}\right]}\right\}\right\}$$

This gives two distinct solutions.

```
In[11]:= RSolve[a[n] == (a[n + 1] a[n - 1])^2, a[n], n]
```

$$\text{Out[11]= } \left\{\left\{a[n] \to e^{C[2] \cos\left[n \text{ ArcTan}\left[\sqrt{15}\right]\right] + C[1] \sin\left[n \text{ ArcTan}\left[\sqrt{15}\right]\right]}\right\}, \left\{a[n] \to e^{\frac{2i\pi}{3} + C[2] \cos\left[n \text{ ArcTan}\left[\sqrt{15}\right]\right] + C[1] \sin\left[n \text{ ArcTan}\left[\sqrt{15}\right]\right]}\right\}\right\}$$

`RSolve` can solve not only ordinary *difference equations* in which the arguments of a differ by integers, but also q*-difference equations* in which the arguments of a are related by multiplicative factors.

This solves the q-difference analog of the factorial equation.

```
In[12]:= RSolve[a[q n] == n a[n], a[n], n]
```

$$\text{Out[12]= } \left\{\left\{a[n] \to n^{\frac{1}{2}\left(-1 + \frac{\text{Log}[n]}{\text{Log}[q]}\right)} C[1]\right\}\right\}$$

Here is a second-order q-difference equation.

```
In[13]:= RSolve[a[n] == a[q n] + a[n / q], a[n], n]
```

$$\text{Out[13]= } \left\{\left\{a[n] \to C[1] \cos\left[\frac{\pi \text{Log}[n]}{3 \text{Log}[q]}\right] + C[2] \sin\left[\frac{\pi \text{Log}[n]}{3 \text{Log}[q]}\right]\right\}\right\}$$

`RSolve[{`eqn_1`,`eqn_2`,...},{`$a_1[n]$`,`$a_2[n]$`,...},n]`

solve a coupled system of recurrence equations

Solving systems of recurrence equations.

This solves a system of two coupled recurrence equations.

```
In[14]:= RSolve[{a[n] == b[n - 1] + n, b[n] == a[n - 1] - n, a[1] == b[1] == 1}, {a[n], b[n]}, n]
```

$$\text{Out[14]= } \left\{\left\{a[n] \to \frac{1}{4}\left(4 + 3(-1)^n + (-1)^{2n} + 2(-1)^{2n} n\right), b[n] \to \frac{1}{4}\left(4 - 3(-1)^n - (-1)^{2n} - 2(-1)^{2n} n\right)\right\}\right\}$$

RSolve[*eqns*, *a*[n_1, n_2, ...], {n_1, n_2, ...}]	solve partial recurrence equations

Solving partial recurrence equations.

Just as one can set up partial differential equations that involve functions of several variables, so one can also set up partial recurrence equations that involve multidimensional sequences. Just as in the differential equations case, general solutions to partial recurrence equations can involve undetermined functions.

This gives the general solution to a simple partial recurrence equation.

```
In[15]:= RSolve[a[i + 1, j + 1] == i j a[i, j], a[i, j], {i, j}]
```

$$\text{Out[15]}= \left\{\left\{\text{a[i, j]} \to \frac{\text{Gamma[i] Gamma[j] C[1][i - j]}}{\text{Gamma[1 - i + j]}}\right\}\right\}$$

Finding Limits

In doing many kinds of calculations, you need to evaluate expressions when variables take on particular values. In many cases, you can do this simply by applying transformation rules for the variables using the `/.` operator.

You can get the value of $\cos\left(x^2\right)$ at 0 just by explicitly replacing x with 0, and then evaluating the result.

```
In[1]:= Cos[x^2] /. x -> 0
Out[1]= 1
```

In some cases, however, you have to be more careful.

Consider, for example, finding the value of the expression $\frac{\sin(x)}{x}$ when $x = 0$. If you simply replace x by 0 in this expression, you get the indeterminate result $\frac{0}{0}$. To find the correct value of $\frac{\sin(x)}{x}$ when $x = 0$, you need to take the *limit*.

Limit[*expr*, *x*->x_0]	find the limit of *expr* when x approaches x_0

Finding limits.

This gives the correct value for the limit of $\frac{\sin(x)}{x}$ as $x \to 0$.

```
In[2]:= Limit[Sin[x] / x, x -> 0]
Out[2]= 1
```

No finite limit exists in this case.

```
In[3]:= Limit[Sin[x] / x^2, x -> 0]
Out[3]= ∞
```

`Limit` can find this limit, even though you cannot get an ordinary power series for $x \log(x)$ at $x = 0$.

```
In[4]:= Limit[x Log[x], x -> 0]
Out[4]= 0
```

The same is true here.

```
In[5]:= Limit[(1 + 2 x) ^ (1 / x), x -> 0]
Out[5]= e^2
```

The value of `Sign[x]` at $x = 0$ is 0.

```
In[6]:= Sign[0]
Out[6]= 0
```

Its *limit*, however, is 1. The limit is by default taken from above.

```
In[7]:= Limit[Sign[x], x -> 0]
Out[7]= 1
```

Not all functions have definite limits at particular points. For example, the function $\sin(1/x)$ oscillates infinitely often near $x = 0$, so it has no definite limit there. Nevertheless, at least so long as x remains real, the values of the function near $x = 0$ always lie between -1 and 1. `Limit` represents values with bounded variation using `Interval` objects. In general, `Interval[{`x_{min}`,` x_{max}`}]` represents an uncertain value which lies somewhere in the interval x_{min} to x_{max}.

`Limit` returns an `Interval` object, representing the range of possible values of $\sin(1/x)$ near its essential singularity at $x = 0$.

```
In[8]:= Limit[Sin[1 / x], x -> 0]
Out[8]= Interval[{-1, 1}]
```

Mathematica can do arithmetic with `Interval` objects.

```
In[9]:= (1 + %) ^ 3
Out[9]= Interval[{0, 8}]
```

Mathematica represents this limit symbolically in terms of an `Interval` object.

```
In[10]:= Limit[Exp[Sin[x]], x -> Infinity]
Out[10]= Interval[{1/E, E}]
```

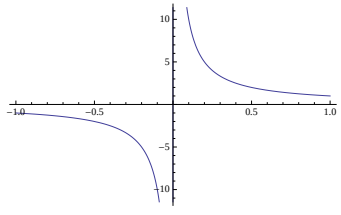
Some functions may have different limits at particular points, depending on the direction from which you approach those points. You can use the `Direction` option for `Limit` to specify the direction you want.

`Limit[`*expr*`,`*x*`->`x_0`,Direction->1]`	find the limit as x approaches x_0 from below
`Limit[`*expr*`,`*x*`->`x_0`,Direction->-1]`	find the limit as x approaches x_0 from above

Directional limits.

The function $1/x$ has a different limiting value at $x = 0$, depending on whether you approach from above or below.

```
In[11]:= Plot[1 / x, {x, -1, 1}]
```



Out[11]=

Approaching from below gives a limiting value of $-\infty$.

```
In[12]:= Limit[1 / x, x -> 0, Direction -> 1]
Out[12]= -∞
```

Approaching from above gives a limiting value of ∞.

```
In[13]:= Limit[1 / x, x -> 0, Direction -> -1]
Out[13]= ∞
```

`Limit` makes no assumptions about functions like `f[x]` about which it does not have definite knowledge. As a result, `Limit` remains unevaluated in most cases involving symbolic functions.

`Limit` has no definite knowledge about `f`, so it leaves this limit unevaluated.

```
In[14]:= Limit[x f[x], x -> 0]
Out[14]= Limit[x f[x], x → 0]
```

Residues

`Limit[`*expr*`, x -> `x_0`]` tells you what the value of *expr* is when x tends to x_0. When this value is infinite, it is often useful instead to know the *residue* of *expr* when x equals x_0. The residue is given by the coefficient of $(x - x_0)^{-1}$ in the power series expansion of *expr* about the point x_0.

`Residue[`*expr*`, {`x`, `x_0`}]`	the residue of *expr* when x equals x_0

Computing residues.

The residue here is equal to 1.

```
In[1]:= Residue[1 / x, {x, 0}]
Out[1]= 1
```

The residue here is zero.

```
In[2]:= Residue[1 / x^2, {x, 0}]
Out[2]= 0
```

Residues can be computed at the point at infinity.

```
In[3]:= Residue[1 / x, {x, ComplexInfinity}]
Out[3]= -1
```

Padé Approximation

The Padé approximation is a rational function that can be thought of as a generalization of a Taylor polynomial. A rational function is the ratio of polynomials. Because these functions only use the elementary arithmetic operations, they are very easy to evaluate numerically. The polynomial in the denominator allows you to approximate functions that have rational singularities.

`PadeApproximant[f,{x,x0,{n,m}}]`	give the Padé approximation to f centered at x_0 of order (n, m)
`PadeApproximant[f,{x,x0,n}]`	give the diagonal Padé approximation to f centered at x_0 of order n

Padé approximations.

More precisely, a Padé approximation of order (n, m) to an analytic function $f(x)$ at a regular point or pole x_0 is the rational function $\frac{p(x)}{q(x)}$ where $p(x)$ is a polynomial of degree n, $q(x)$ is a polynomial of degree m, and the formal power series of $f(x)q(x) - p(x)$ about the point x_0 begins with the term $(x - x_0)^{n+m+1}$. If m is equal to n, the approximation is called a diagonal Padé approximation of order n.

Here is the Padé approximation of order $(2, 4)$ to $\cos(x)$ at $x = 0$.

```
In[1]:= PadeApproximant[Cos[x], {x, 0, {2, 4}}]
```

$$\text{Out[1]}= \frac{1 - \frac{61\,x^2}{150}}{1 + \frac{7\,x^2}{75} + \frac{x^4}{200}}$$

This gives another Padé approximation of the same order.

```
In[2]:= pd = PadeApproximant[E^x, {x, 1, {2, 4}}]
```

$$\text{Out[2]}= \frac{e + \frac{1}{3}e\,(-1+x) + \frac{1}{30}e\,(-1+x)^2}{1 - \frac{2}{3}(-1+x) + \frac{1}{5}(-1+x)^2 - \frac{1}{30}(-1+x)^3 + \frac{1}{360}(-1+x)^4}$$

The initial terms of this series vanish. This is the property that characterizes the Padé approximation.

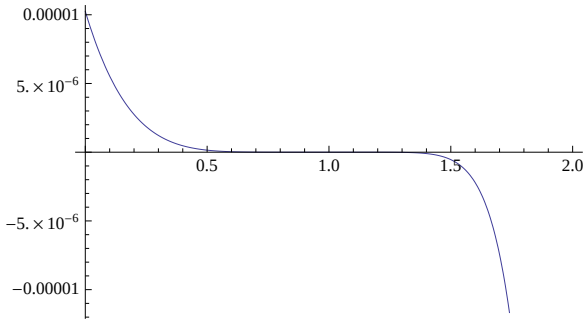
```
In[3]:= Series[E^x Denominator[pd] - Numerator[pd], {x, 1, 8}]
```

$$\text{Out[3]}= \frac{e\,(x-1)^7}{75\,600} + \frac{e\,(x-1)^8}{120\,960} + O[x-1]^9$$

This plots the difference between the approximation and the true function. Notice that the approximation is very good near the center of expansion, but the error increases rapidly as you move away.

```
In[4]:= Plot[pd - e^x, {x, 0, 2}]
```

Out[4]=



In *Mathematica* `PadeApproximant` is generalized to allow expansion about branch points.

This gives the diagonal Padé approximation of order 1 to a generalized rational function at $x = 0$.

```
In[5]:= PadeApproximant[Sqrt[x]/(1 + Sqrt[x])^3, {x, 0, 1}]
```

Out[5]=

$$\frac{\sqrt{x} - \frac{x}{3}}{1 + \frac{8\sqrt{x}}{3} + 2x}$$

This gives the diagonal Padé approximation of order 5 to the logarithm of a rational function at the branch point $x = 0$.

```
In[6]:= PadeApproximant[Log[x/(1 + x)], {x, 0, 5}]
```

Out[6]=

$$\frac{-x - 2x^2 - \frac{47x^3}{36} - \frac{11x^4}{36} - \frac{137x^5}{7560}}{1 + \frac{5x}{2} + \frac{20x^2}{9} + \frac{5x^3}{6} + \frac{5x^4}{42} + \frac{x^5}{252}} + \text{Log}[x]$$

The series expansion of the function agrees with the diagonal Padé approximation up to order 10.

```
In[7]:= Series[% - Log[x/(1 + x)], {x, 0, 11}]
```

Out[7]=

$$\frac{x^{11}}{698\,544} + O[x]^{12}$$

Calculus

Differentiation

D[*f*, *x*]	partial derivative $\frac{\partial}{\partial x} f$
D[*f*, *x*, *y*, ...]	multiple derivative $\frac{\partial}{\partial x} \frac{\partial}{\partial y} \ldots f$
D[*f* {*x*, *n*}]	n^{th} derivative $\frac{\partial^n}{\partial x^n} f$
D[*f*, *x*, NonConstants->{v_1, v_2, ...}]	$\frac{\partial}{\partial x} f$ with the v_i taken to depend on x

Partial differentiation operations.

This gives $\frac{\partial}{\partial x} x^n$.

```
In[1]:= D[x^n, x]
Out[1]= n x^(-1+n)
```

This gives the third derivative.

```
In[2]:= D[x^n, {x, 3}]
Out[2]= (-2 + n) (-1 + n) n x^(-3+n)
```

You can differentiate with respect to any expression that does not involve explicit mathematical operations.

```
In[3]:= D[x[1]^2 + x[2]^2, x[1]]
Out[3]= 2 x[1]
```

D does *partial differentiation*. It assumes here that y is independent of x.

```
In[4]:= D[x^2 + y^2, x]
Out[4]= 2 x
```

If `y` does in fact depend on `x`, you can use the explicit functional form `y[x]`. "The Representation of Derivatives" describes how objects like `y'[x]` work.

```
In[5]:= D[x^2 + y[x]^2, x]
Out[5]= 2 x + 2 y[x] y'[x]
```

Instead of giving an explicit function `y[x]`, you can tell `D` that `y` *implicitly* depends on `x`. `D[y, x, NonConstants -> {y}]` then represents $\frac{\partial y}{\partial x}$, with `y` implicitly depending on `x`.

```
In[6]:= D[x^2 + y^2, x, NonConstants -> {y}]
Out[6]= 2 x + 2 y D[y, x, NonConstants → {y}]
```

D[f, {{x_1, x_2, ...}}]	the gradient of a scalar function f ($\partial f/\partial x_1, \partial f/\partial x_2, \ldots$)
D[f, {{x_1, x_2, ...}, 2}]	the Hessian matrix for f
D[f, {{x_1, x_2, ...}, n}]	the n^{th}-order Taylor series coefficient
D[{f_1, f_2, ...}, {{x_1, x_2, ...}}]	the Jacobian for a vector function f

Vector derivatives.

This gives the gradient of the function $x^2 + y^2$.

```
In[7]:= D[x^2 + y^2, {{x, y}}]
Out[7]= {2 x, 2 y}
```

This gives the Hessian.

```
In[8]:= D[x^2 + y^2, {{x, y}, 2}]
Out[8]= {{2, 0}, {0, 2}}
```

This gives the Jacobian for a vector function.

```
In[9]:= D[{x^2 + y^2, x y}, {{x, y}}]
Out[9]= {{2 x, 2 y}, {y, x}}
```

Total Derivatives

`Dt[f]`	total differential df
`Dt[f,x]`	total derivative $\frac{df}{dx}$
`Dt[f,x,y,...]`	multiple total derivative $\frac{d}{dx}\frac{d}{dy}\ldots f$
`Dt[f,x,Constants->{c1,c2,...}]`	total derivative with c_i constant (i.e., $dc_i = 0$)
`y/:Dt[y,x]=0`	set $\frac{dy}{dx} = 0$
`SetAttributes[c,Constant]`	define c to be a constant in all cases

Total differentiation operations.

When you find the derivative of some expression f with respect to x, you are effectively finding out how fast f changes as you vary x. Often f will depend not only on x, but also on other variables, say y and z. The results that you get then depend on how you assume that y and z vary as you change x.

There are two common cases. Either y and z are assumed to stay fixed when x changes, or they are allowed to vary with x. In a standard *partial derivative* $\frac{\partial f}{\partial x}$, all variables other than x are assumed fixed. On the other hand, in the *total derivative* $\frac{df}{dx}$, all variables are allowed to change with x.

In *Mathematica*, `D[f, x]` gives a partial derivative, with all other variables assumed independent of x. `Dt[f, x]` gives a total derivative, in which all variables are assumed to depend on x. In both cases, you can add an argument to give more information on dependencies.

This gives the *partial derivative* $\frac{\partial}{\partial x}(x^2+y^2)$. `y` is assumed to be independent of `x`.

```
In[1]:= D[x^2 + y^2, x]
Out[1]= 2 x
```

This gives the *total derivative* $\frac{d}{dx}(x^2+y^2)$. Now `y` is assumed to depend on `x`.

```
In[2]:= Dt[x^2 + y^2, x]
Out[2]= 2 x + 2 y Dt[y, x]
```

You can make a replacement for $\frac{dy}{dx}$.

```
In[3]:= % /. Dt[y, x] -> yp
Out[3]= 2 x + 2 y yp
```

You can also make an explicit definition for $\frac{dy}{dx}$. You need to use `y /:` to make sure that the definition is associated with `y`.

```
In[4]:= y /: Dt[y, x] = 0
Out[4]= 0
```

With this definition made, `Dt` treats `y` as independent of `x`.

```
In[5]:= Dt[x^2 + y^2 + z^2, x]
Out[5]= 2 x + 2 z Dt[z, x]
```

This removes your definition for the derivative of `y`.

```
In[6]:= Clear[y]
```

This takes the total derivative, with `z` held fixed.

```
In[7]:= Dt[x^2 + y^2 + z^2, x, Constants -> {z}]
Out[7]= 2 x + 2 y Dt[y, x, Constants → {z}]
```

This specifies that `c` is a constant under differentiation.

```
In[8]:= SetAttributes[c, Constant]
```

The variable `c` is taken as a constant.

```
In[9]:= Dt[a^2 + c x^2, x]
Out[9]= 2 c x + 2 a Dt[a, x]
```

The *function* `c` is also assumed to be a constant.

```
In[10]:= Dt[a^2 + c[x] x^2, x]
Out[10]= 2 x c[x] + 2 a Dt[a, x]
```

This gives the total differential $d\left(x^2 + c\, y^2\right)$.

```
In[11]:= Dt[x^2 + c y^2]
Out[11]= 2 x Dt[x] + 2 c y Dt[y]
```

You can make replacements and assignments for total differentials.

```
In[12]:= % /. Dt[y] -> dy
Out[12]= 2 c dy y + 2 x Dt[x]
```

Derivatives of Unknown Functions

Differentiating a known function gives an explicit result.

```
In[1]:= D[Log[x]^2, x]
```

$$\text{Out[1]=} \frac{2\,\text{Log}[x]}{x}$$

Differentiating an unknown function `f` gives a result in terms of `f '`.

```
In[2]:= D[f[x]^2, x]
Out[2]= 2 f[x] f'[x]
```

Mathematica applies the chain rule for differentiation, and leaves the result in terms of `f '`.

```
In[3]:= D[x f[x^2], x]
Out[3]= f[x^2] + 2 x^2 f'[x^2]
```

Differentiating again gives a result in terms of `f`, `f '` and `f ''`.

```
In[4]:= D[%, x]
Out[4]= 6 x f'[x^2] + 4 x^3 f''[x^2]
```

When a function has more than one argument, superscripts are used to indicate how many times each argument is being differentiated.

```
In[5]:= D[g[x^2, y^2], x]
Out[5]= 2 x g^(1,0)[x^2, y^2]
```

This represents $\frac{\partial}{\partial x}\frac{\partial}{\partial x}\frac{\partial}{\partial y} g(x, y)$. *Mathematica* assumes that the order in which derivatives are taken with respect to different variables is irrelevant.

```
In[6]:= D[g[x, y], x, x, y]
Out[6]= g^(2,1)[x, y]
```

You can find the value of the derivative when $x = 0$ by replacing x with 0.

```
In[7]:= % /. x -> 0
Out[7]= g^(2,1)[0, y]
```

$f'[x]$	first derivative of a function of one variable
$f^{(n)}[x]$	n^{th} derivative of a function of one variable
$f^{(n_1,n_2,\dots)}[x]$	derivative of a function of several variables, n_i times with respect to variable i

Output forms for derivatives of unknown functions.

The Representation of Derivatives

Derivatives in *Mathematica* work essentially the same as in standard mathematics. The usual mathematical notation, however, often hides many details. To understand how derivatives are represented in *Mathematica*, we must look at these details.

The standard mathematical notation $f'(0)$ is really a shorthand for $\frac{d}{dt} f(t)|_{t=0}$, where t is a "dummy variable". Similarly, $f'(x^2)$ is a shorthand for $\frac{d}{dt} f(t)|_{t=x^2}$. As suggested by the notation f', the object $\frac{d}{dt} f(t)$ can in fact be viewed as a "pure function", to be evaluated with a particular choice of its parameter t. You can think of the operation of differentiation as acting on a function f, to give a new function, usually called f'.

With functions of more than one argument, the simple notation based on primes breaks down. You cannot tell for example whether $g'(0, 1)$ stands for $\frac{d}{dt} g(t, 1)|_{t=0}$ or $\frac{d}{dt} g(0, t)|_{t=1}$, and for almost any g, these will have totally different values. Once again, however, t is just a dummy variable, whose sole purpose is to show with respect to which "slot" g is to be differentiated.

In *Mathematica*, as in some branches of mathematics, it is convenient to think about a kind of differentiation that acts on *functions*, rather than expressions. We need an operation that takes the function f, and gives us the *derivative function* f'. Operations such as this that act on *functions*, rather than variables, are known in mathematics as *operators*.

The object `f'` in *Mathematica* is the result of applying the differentiation operator to the function `f`. The full form of `f'` is in fact `Derivative[1][f]`. `Derivative[1]` is the *Mathematica* differentiation operator.

The arguments in the operator `Derivative[`n_1`, `n_2`, ...]` specify how many times to differentiate with respect to each "slot" of the function on which it acts. By using operators to represent differentiation, *Mathematica* avoids any need to introduce explicit "dummy variables".

This is the full form of the derivative of the function `f`.

```
In[1]:= f' // FullForm
Out[1]//FullForm= Derivative[1][f]
```

Here an argument `x` is supplied.

```
In[2]:= f'[x] // FullForm
Out[2]//FullForm= Derivative[1][f][x]
```

This is the second derivative.

```
In[3]:= f''[x] // FullForm
Out[3]//FullForm= Derivative[2][f][x]
```

This gives a derivative of the function `g` with respect to its second "slot".

```
In[4]:= D[g[x, y], y]
Out[4]= g^(0,1)[x, y]
```

Here is the full form.

```
In[5]:= % // FullForm
Out[5]//FullForm= Derivative[0, 1][g][x, y]
```

Here is the second derivative with respect to the variable `y`, which appears in the second slot of `g`.

```
In[6]:= D[g[x, y], {y, 2}] // FullForm
Out[6]//FullForm= Derivative[0, 2][g][x, y]
```

This is a mixed derivative.

```
In[7]:= D[g[x, y], x, y, y] // FullForm
Out[7]//FullForm= Derivative[1, 2][g][x, y]
```

Since `Derivative` only specifies how many times to differentiate with respect to each slot, the order of the derivatives is irrelevant.

```
In[8]:= D[g[x, y], y, y, x] // FullForm
Out[8]//FullForm= Derivative[1, 2][g][x, y]
```

Here is a more complicated case, in which both arguments of `g` depend on the differentiation variable.

```
In[9]:= D[g[x, x], x]
Out[9]= g^(0,1)[x, x] + g^(1,0)[x, x]
```

This is the full form of the result.

```
In[10]:= % // FullForm
Out[10]//FullForm= Plus[Derivative[0, 1][g][x, x], Derivative[1, 0][g][x, x]]
```

The object `f'` behaves essentially like any other function in *Mathematica*. You can evaluate the function with any argument, and you can use standard *Mathematica* `/.` operations to change the argument. (This would not be possible if explicit dummy variables had been introduced in the course of the differentiation.)

This is the *Mathematica* representation of the derivative of a function `f`, evaluated at the origin.

```
In[11]:= f'[0] // FullForm
Out[11]//FullForm= Derivative[1][f][0]
```

The result of this derivative involves `f'` evaluated with the argument `x^2`.

```
In[12]:= D[f[x^2], x]
Out[12]= 2 x f'[x^2]
```

You can evaluate the result at the point $x = 2$ by using the standard *Mathematica* replacement operation.

```
In[13]:= % /. x -> 2
Out[13]= 4 f'[4]
```

There is some slight subtlety when you need to deduce the value of `f'` based on definitions for objects like `f[x_]`.

Here is a definition for a function `h`.

```
In[14]:= h[x_] := x^4
```

When you take the derivative of `h[x]`, *Mathematica* first evaluates `h[x]`, then differentiates the result.

```
In[15]:= D[h[x], x]
Out[15]= 4 x^3
```

You can get the same result by applying the function `h'` to the argument `x`.

```
In[16]:= h'[x]
Out[16]= 4 x^3
```

Here is the function `h'` on its own.

```
In[17]:= h'
Out[17]= 4 #1^3 &
```

The function `f'` is completely determined by the form of the function `f`. Definitions for objects like `f[x_]` do not immediately apply however to expressions like `f'[x]`. The problem is that `f'[x]` has the full form `Derivative[1][f][x]`, which nowhere contains anything that explicitly matches the pattern `f[x_]`. In addition, for many purposes it is convenient to have a representation of the function `f'` itself, without necessarily applying it to any arguments.

What *Mathematica* does is to try and find the explicit form of a *pure function* which represents the object `f'`. When *Mathematica* gets an expression like `Derivative[1][f]`, it effectively converts it to the explicit form `D[f[#], #] &` and then tries to evaluate the derivative. In the explicit form, *Mathematica* can immediately use values that have been defined for objects like `f[x_]`. If *Mathematica* succeeds in doing the derivative, it returns the explicit pure-function result. If it does not succeed, it leaves the derivative in the original `f'` form.

This gives the derivative of `Tan` in pure-function form.

```
In[18]:= Tan'
Out[18]= Sec[#1]^2 &
```

Here is the result of applying the pure function to the specific argument y.

```
In[19]:= %[y]
Out[19]= Sec[y]^2
```

Defining Derivatives

You can define the derivative in *Mathematica* of a function f of one argument simply by an assignment like f'[x_] = fp[x].

This defines the derivative of $f(x)$ to be $fp(x)$. In this case, you could have used = instead of :=.

```
In[1]:= f'[x_] := fp[x]
```

The rule for f'[x_] is used to evaluate this derivative.

```
In[2]:= D[f[x^2], x]
Out[2]= 2 x fp[x^2]
```

Differentiating again gives derivatives of fp.

```
In[3]:= D[%, x]
Out[3]= 2 fp[x^2] + 4 x^2 fp'[x^2]
```

This defines a value for the derivative of g at the origin.

```
In[4]:= g'[0] = g0
Out[4]= g0
```

The value for g'[0] is used.

```
In[5]:= D[g[x]^2, x] /. x -> 0
Out[5]= 2 g0 g[0]
```

This defines the second derivative of g, with any argument.

```
In[6]:= g''[x_] = gpp[x]
Out[6]= gpp[x]
```

The value defined for the second derivative is used.

```
In[7]:= D[g[x]^2, {x, 2}]
Out[7]= 2 g[x] gpp[x] + 2 g'[x]^2
```

To define derivatives of functions with several arguments, you have to use the general representation of derivatives in *Mathematica*.

f'[x_]:=*rhs*	define the first derivative of f
Derivative[n][f][x_]:=*rhs*	define the n^{th} derivative of f
Derivative[m,n,...][g][x_,_,...]:=*rhs*	define derivatives of g with respect to various arguments

Defining derivatives.

This defines the second derivative of g with respect to its second argument.

```
In[8]:= Derivative[0, 2][g][x_, y_] := g2p[x, y]
```

This uses the definition just given.

```
In[9]:= D[g[a^2, x^2], x, x]
Out[9]= 4 x^2 g2p[a^2, x^2] + 2 g^(0,1)[a^2, x^2]
```

Integration

Here is the integral $\int x^n \, dx$ in *Mathematica*.

```
In[1]:= Integrate[x^n, x]
Out[1]=  x^(1+n)
         -------
         1 + n
```

Here is a slightly more complicated example.

```
In[2]:= Integrate[1 / (x^4 - a^4), x]
Out[2]= -ArcTan[x/a]/(2 a^3) + Log[a - x]/(4 a^3) - Log[a + x]/(4 a^3)
```

Mathematica knows how to do almost any integral that can be done in terms of standard mathematical functions. But you should realize that even though an integrand may contain only fairly simple functions, its integral may involve much more complicated functions—or may not be expressible at all in terms of standard mathematical functions.

Here is a fairly straightforward integral.

```
In[3]:= Integrate[Log[1 - x^2], x]
```

$$\text{Out[3]= } -2\,x - \text{Log}[-1 + x] + \text{Log}[1 + x] + x\,\text{Log}\left[1 - x^2\right]$$

This integral can be done only in terms of a dilogarithm function.

```
In[4]:= Integrate[Log[1 - x^2] / x, x]
```

$$\text{Out[4]= } -\frac{1}{2}\,\text{PolyLog}\left[2,\, x^2\right]$$

This integral involves `Erf`.

```
In[5]:= Integrate[Exp[1 - x^2], x]
```

$$\text{Out[5]= } \frac{1}{2}\,e\,\sqrt{\pi}\;\text{Erf}[x]$$

And this one involves a Fresnel function.

```
In[6]:= Integrate[Sin[x^2], x]
```

$$\text{Out[6]= } \sqrt{\frac{\pi}{2}}\;\text{FresnelS}\left[\sqrt{\frac{2}{\pi}}\;x\right]$$

Even this integral requires a hypergeometric function.

```
In[7]:= Integrate[(1 - x^2)^n, x]
```

$$\text{Out[7]= } x\,\text{Hypergeometric2F1}\left[\frac{1}{2},\, -n,\, \frac{3}{2},\, x^2\right]$$

This integral simply cannot be done in terms of standard mathematical functions. As a result, *Mathematica* just leaves it undone.

```
In[8]:= Integrate[x^x, x]
```

$$\text{Out[8]= } \int x^x\,dx$$

`Integrate[`f`,`x`]`	the indefinite integral $\int f\, dx$
`Integrate[`f`,`x`,`y`]`	the multiple integral $\int dx\, dy\, f$
`Integrate[`f`,{`x`,`x_{min}`,`x_{max}`}]`	the definite integral $\int_{x_{min}}^{x_{max}} f\, dx$
`Integrate[`f`,{`x`,`x_{min}`,`x_{max}`},{`y`,`y_{min}`,`y_{max}`}]`	the multiple integral $\int_{x_{min}}^{x_{max}} dx \int_{y_{min}}^{y_{max}} dy\, f$

Integration.

Here is the definite integral $\int_a^b \sin^2(x)\, dx$.

```
In[9]:= Integrate[Sin[x]^2, {x, a, b}]
Out[9]= 1/2 (-a + b + Cos[a] Sin[a] - Cos[b] Sin[b])
```

Here is another definite integral.

```
In[10]:= Integrate[Exp[-x^2], {x, 0, Infinity}]
Out[10]= Sqrt[π]/2
```

Mathematica cannot give you a formula for this definite integral.

```
In[11]:= Integrate[x^x, {x, 0, 1}]
Out[11]= ∫_0^1 x^x dx
```

You can still get a numerical result, though.

```
In[12]:= N[%]
Out[12]= 0.783431
```

This evaluates the multiple integral $\int_0^1 dx \int_0^x dy\, (x^2 + y^2)$. The range of the outermost integration variable appears first.

```
In[13]:= Integrate[x^2 + y^2, {x, 0, 1}, {y, 0, x}]
Out[13]= 1/3
```

This integrates x^{10} over a circular region.

```
In[14]:= Integrate[x^10 Boole[x^2 + y^2 <= 1], {x, -1, 1}, {y, -1, 1}]

Out[14]= 21 π
         ----
         512
```

Indefinite Integrals

The *Mathematica* function `Integrate[`f`, `x`]` gives you the *indefinite integral* $\int f\, dx$. You can think of the operation of indefinite integration as being an inverse of differentiation. If you take the result from `Integrate[`f`, `x`]`, and then differentiate it, you always get a result that is mathematically equal to the original expression f.

In general, however, there is a whole family of results which have the property that their derivative is f. `Integrate[`f`, `x`]` gives you *an* expression whose derivative is f. You can get other expressions by adding an arbitrary constant of integration, or indeed by adding any function that is constant except at discrete points.

If you fill in explicit limits for your integral, any such constants of integration must cancel out. But even though the indefinite integral can have arbitrary constants added, it is still often very convenient to manipulate it without filling in the limits.

Mathematica applies standard rules to find indefinite integrals.

```
In[1]:= Integrate[x^2, x]

Out[1]= x^3
        ---
        3
```

You can add an arbitrary constant to the indefinite integral, and still get the same derivative. `Integrate` simply gives you *an* expression with the required derivative.

```
In[2]:= D[% + c, x]

Out[2]= x^2
```

This gives the indefinite integral $\int \frac{dx}{x^2-1}$.

```
In[3]:= Integrate[1 / (x^2 - 1), x]

Out[3]= 1          1
        - Log[-1 + x] - - Log[1 + x]
        2          2
```

Differentiating should give the original function back again.

```
In[4]:= D[%, x]

Out[4]= 1/(2 (-1 + x)) - 1/(2 (1 + x))
```

You need to manipulate it to get it back into the original form.

```
In[5]:= Simplify[%]

Out[5]= 1/(-1 + x^2)
```

The `Integrate` function assumes that any object that does not explicitly contain the integration variable is independent of it, and can be treated as a constant. As a result, `Integrate` is like an inverse of the *partial differentiation* function `D`.

The variable `a` is assumed to be independent of `x`.

```
In[6]:= Integrate[a x^2, x]

Out[6]= a x^3/3
```

The integration variable can be any expression that does not involve explicit mathematical operations.

```
In[7]:= Integrate[x b[x]^2, b[x]]

Out[7]= 1/3 x b[x]^3
```

Another assumption that `Integrate` implicitly makes is that all the symbolic quantities in your integrand have "generic" values. Thus, for example, *Mathematica* will tell you that $\int x^n dx$ is $\frac{x^{n+1}}{n+1}$ even though this is not true in the special case $n = -1$.

Mathematica gives the standard result for this integral, implicitly assuming that `n` is not equal to `-1`.

```
In[8]:= Integrate[x^n, x]

Out[8]= x^(1+n)/(1 + n)
```

If you specifically give an exponent of `-1`, *Mathematica* produces a different result.

```
In[9]:= Integrate[x^-1, x]
Out[9]= Log[x]
```

You should realize that the result for any particular integral can often be written in many different forms. *Mathematica* tries to give you the most convenient form, following principles such as avoiding explicit complex numbers unless your input already contains them.

This integral is given in terms of `ArcTan`.

```
In[10]:= Integrate[1 / (1 + a x^2), x]
```

$$\text{Out[10]}= \frac{\text{ArcTan}\left[\sqrt{a}\ x\right]}{\sqrt{a}}$$

This integral is given in terms of `ArcTanh`.

```
In[11]:= Integrate[1 / (1 - b x^2), x]
```

$$\text{Out[11]}= \frac{\text{ArcTanh}\left[\sqrt{b}\ x\right]}{\sqrt{b}}$$

This is mathematically equal to the first integral, but is given in a somewhat different form.

```
In[12]:= % /. b -> -a
```

$$\text{Out[12]}= \frac{\text{ArcTanh}\left[\sqrt{-a}\ x\right]}{\sqrt{-a}}$$

The derivative is still correct.

```
In[13]:= D[%, x]
```

$$\text{Out[13]}= \frac{1}{1 + a\,x^2}$$

Even though they look quite different, both `ArcTan[x]` and `-ArcTan[1 / x]` are indefinite integrals of $1/(1+x^2)$.

```
In[14]:= Simplify[D[{ArcTan[x], -ArcTan[1 / x]}, x]]
```

$$\text{Out[14]}= \left\{\frac{1}{1 + x^2}, \frac{1}{1 + x^2}\right\}$$

`Integrate` chooses to use the simpler of the two forms.

```
In[15]:= Integrate[1 / (1 + x^2), x]
Out[15]= ArcTan[x]
```

Integrals That Can and Cannot Be Done

Evaluating integrals is much more difficult than evaluating derivatives. For derivatives, there is a systematic procedure based on the chain rule that effectively allows any derivative to be worked out. But for integrals, there is no such systematic procedure.

One of the main problems is that it is difficult to know what kinds of functions will be needed to evaluate a particular integral. When you work out a derivative, you always end up with functions that are of the same kind or simpler than the ones you started with. But when you work out integrals, you often end up needing to use functions that are much more complicated than the ones you started with.

This integral can be evaluated using the same kind of functions that appeared in the input.

```
In[1]:= Integrate[Log[x]^2, x]
```

$$\text{Out[1]= } 2x - 2x\,\text{Log}[x] + x\,\text{Log}[x]^2$$

But for this integral the special function `LogIntegral` is needed.

```
In[2]:= Integrate[Log[Log[x]], x]
```

$$\text{Out[2]= } x\,\text{Log}[\text{Log}[x]] - \text{LogIntegral}[x]$$

It is not difficult to find integrals that require all sorts of functions.

```
In[3]:= Integrate[Sin[x^2], x]
```

$$\text{Out[3]= } \sqrt{\frac{\pi}{2}}\ \text{FresnelS}\left[\sqrt{\frac{2}{\pi}}\ x\right]$$

This integral involves an incomplete gamma function. Note that the power is carefully set up to allow any complex value of `x`.

```
In[4]:= Integrate[Exp[-x^a], x]
```

$$\text{Out[4]= } -\frac{x\ (x^a)^{-1/a}\ \text{Gamma}\left[\frac{1}{a}, x^a\right]}{a}$$

Mathematica includes a very wide range of mathematical functions, and by using these functions a great many integrals can be done. But it is still possible to find even fairly simple-looking integrals that just cannot be done in terms of any standard mathematical functions.

Here is a fairly simple-looking integral that cannot be done in terms of any standard mathematical functions.

```
In[5]:= Integrate[Sin[x] / Log[x], x]
```

$$\text{Out[5]= } \int \frac{\text{Sin}[x]}{\text{Log}[x]}\, dx$$

The main point of being able to do an integral in terms of standard mathematical functions is that it lets one use the known properties of these functions to evaluate or manipulate the result one gets.

In the most convenient cases, integrals can be done purely in terms of elementary functions such as exponentials, logarithms and trigonometric functions. In fact, if you give an integrand that involves only such elementary functions, then one of the important capabilities of `Integrate` is that if the corresponding integral can be expressed in terms of elementary functions, then `Integrate` will essentially always succeed in finding it.

Integrals of rational functions are straightforward to evaluate, and always come out in terms of rational functions, logarithms and inverse trigonometric functions.

```
In[6]:= Integrate[x / ((x - 1) (x + 2)), x]
```

$$\text{Out[6]= } \frac{1}{3}\text{Log}[-1 + x] + \frac{2}{3}\text{Log}[2 + x]$$

The integral here is still of the same form, but now involves an implicit sum over the roots of a polynomial.

```
In[7]:= Integrate[1 / (1 + 2 x + x^3), x]
```

$$\text{Out[7]= } \text{RootSum}\left[1 + 2\,\#1 + \#1^3\ \&,\ \frac{\text{Log}[x - \#1]}{2 + 3\,\#1^2}\ \&\right]$$

This finds numerical approximations to all the root objects.

```
In[8]:= N[%]
```

```
Out[8]= (-0.19108 - 0.088541 i) Log[(-0.226699 - 1.46771 i) + x] -
 (0.19108 - 0.088541 i) Log[(-0.226699 + 1.46771 i) + x] + 0.38216 Log[0.453398 + x]
```

Integrals of trigonometric functions usually come out in terms of other trigonometric functions.

```
In[9]:= Integrate[Sin[x]^3 Cos[x]^2, x]
```

$$\text{Out[9]= } -\frac{\text{Cos}[x]}{8} - \frac{1}{48}\text{Cos}[3x] + \frac{1}{80}\text{Cos}[5x]$$

This is a fairly simple integral involving algebraic functions.

```
In[10]:= Integrate[Sqrt[x] Sqrt[1 + x], x]
```

$$\text{Out[10]}= \frac{1}{4}\left(\sqrt{x}\,\sqrt{1+x}\,(1+2x) - \text{ArcSinh}\left[\sqrt{x}\right]\right)$$

Here is an integral involving nested square roots.

```
In[11]:= Integrate[Sqrt[x + Sqrt[x]], x]
```

$$\text{Out[11]}= \frac{1}{12}\sqrt{\sqrt{x}+x}\,\left(-3+2\sqrt{x}+8x\right) + \frac{1}{8}\text{Log}\left[1+2\sqrt{x}+2\sqrt{\sqrt{x}+x}\right]$$

By nesting elementary functions you sometimes get integrals that can be done in terms of elementary functions.

```
In[12]:= Integrate[Cos[Log[x]], x]
```

$$\text{Out[12]}= \frac{1}{2}x\,\text{Cos}[\text{Log}[x]] + \frac{1}{2}x\,\text{Sin}[\text{Log}[x]]$$

But more often other kinds of functions are needed.

```
In[13]:= Integrate[Log[Cos[x]], x]
```

$$\text{Out[13]}= \frac{i\,x^2}{2} - x\,\text{Log}\left[1+e^{2ix}\right] + x\,\text{Log}[\text{Cos}[x]] + \frac{1}{2}i\,\text{PolyLog}\left[2,-e^{2ix}\right]$$

Integrals like this typically come out in terms of elliptic functions.

```
In[14]:= Integrate[Sqrt[Cos[x]], x]
```

$$\text{Out[14]}= 2\,\text{EllipticE}\left[\frac{x}{2}, 2\right]$$

But occasionally one can get results in terms of elementary functions alone.

```
In[15]:= Integrate[Sqrt[Tan[x]], x]
```

$$\text{Out[15]}= \frac{1}{2\sqrt{2}}\left(-2\,\text{ArcTan}\left[1-\sqrt{2}\,\sqrt{\text{Tan}[x]}\right] + 2\,\text{ArcTan}\left[1+\sqrt{2}\,\sqrt{\text{Tan}[x]}\right] + \text{Log}\left[-1+\sqrt{2}\,\sqrt{\text{Tan}[x]}-\text{Tan}[x]\right] - \text{Log}\left[1+\sqrt{2}\,\sqrt{\text{Tan}[x]}+\text{Tan}[x]\right]\right)$$

Integrals like this can systematically be done using `Piecewise`.

```
In[16]:= Integrate[2^Max[x, 1 - x], x]
```

$$\text{Out[16]}= \begin{cases} -\frac{2^{1-x}}{\text{Log}[2]} & x \le \frac{1}{2} \\ -\frac{2\sqrt{2}}{\text{Log}[2]} + \frac{2^x}{\text{Log}[2]} & \text{True} \end{cases}$$

Beyond working with elementary functions, `Integrate` includes a large number of algorithms for dealing with special functions. Sometimes it uses a direct generalization of the procedure for elementary functions. But more often its strategy is first to try to write the integrand in a form that can be integrated in terms of certain sophisticated special functions, and then having done this to try to find reductions of these sophisticated functions to more familiar functions.

To integrate this Bessel function requires a generalized hypergeometric function.

```
In[17]:= Integrate[BesselJ[0, x], x]
```

$$\text{Out[17]= } x\,\text{HypergeometricPFQ}\left[\left\{\frac{1}{2}\right\}, \left\{1, \frac{3}{2}\right\}, -\frac{x^2}{4}\right]$$

Sometimes the integrals can be reduced to more familiar forms.

```
In[18]:= Integrate[x^3 BesselJ[0, x], x]
```

$$\text{Out[18]= } -x^2\,(-2\,\text{BesselJ}[2, x] + x\,\text{BesselJ}[3, x])$$

A large book of integral tables will list perhaps a few thousand indefinite integrals. *Mathematica* can do essentially all of these integrals. And because it contains general algorithms rather than just specific cases, *Mathematica* can actually do a vastly wider range of integrals.

You could expect to find this integral in any large book of integral tables.

```
In[19]:= Integrate[Log[1 - x] / x, x]
```

```
Out[19]= -PolyLog[2, x]
```

To do this integral, however, requires a more general algorithm, rather than just a direct table lookup.

```
In[20]:= Integrate[Log[1 + 3 x + x^2] / x, x]
```

$$\begin{aligned}\text{Out[20]= } &\text{Log}[x]\left(\text{Log}\left[\frac{1}{2}\left(3-\sqrt{5}\right)+x\right]-\text{Log}\left[1+\frac{2x}{3-\sqrt{5}}\right]\right)+\text{Log}[x]\left(\text{Log}\left[\frac{1}{2}\left(3+\sqrt{5}\right)+x\right]-\text{Log}\left[1+\frac{2x}{3+\sqrt{5}}\right]\right)+\\ &\text{Log}[x]\left(-\text{Log}\left[\frac{1}{2}\left(3-\sqrt{5}\right)+x\right]-\text{Log}\left[\frac{1}{2}\left(3+\sqrt{5}\right)+x\right]+\text{Log}\left[1+3x+x^2\right]\right)-\\ &\text{PolyLog}\left[2, -\frac{2x}{3-\sqrt{5}}\right]-\text{PolyLog}\left[2, -\frac{2x}{3+\sqrt{5}}\right]\end{aligned}$$

Particularly if you introduce new mathematical functions of your own, you may want to teach *Mathematica* new kinds of integrals. You can do this by making appropriate definitions for `Integrate`.

In the case of differentiation, the chain rule allows one to reduce all derivatives to a standard form, represented in *Mathematica* using `Derivative`. But for integration, no such similar standard form exists, and as a result you often have to make definitions for several different versions of the same integral. Changes of variables and other transformations can rarely be done automatically by `Integrate`.

This integral cannot be done in terms of any of the standard mathematical functions built into *Mathematica*.

```
In[21]:= Integrate[Sin[Sin[x]], x]
Out[21]= ∫Sin[Sin[x]] dx
```

Before you add your own rules for integration, you have to remove write protection.

```
In[22]:= Unprotect[Integrate]
Out[22]= {Integrate}
```

You can set up your own rule to define the integral to be, say, a "Jones" function.

```
In[23]:= Integrate[Sin[Sin[a_. + b_. x_]], x_] := Jones[a, x] / b
```

Now *Mathematica* can do integrals that give Jones functions.

```
In[24]:= Integrate[Sin[Sin[3 x]], x]
Out[24]= -1/3 Jones[0, x]
```

As it turns out, the integral $\int \sin(\sin(x))\, dx$ can in principle be represented as an infinite sum of ${}_2F_1$ hypergeometric functions, or as a suitably generalized Kampé de Fériet hypergeometric function of two variables.

Definite Integrals

`Integrate[`f`,`x`]`	the indefinite integral $\int f\, dx$
`Integrate[`f`,{`x`,`x_{min}`,`x_{max}`}]`	the definite integral $\int_{x_{min}}^{x_{max}} f\, dx$
`Integrate[`f`,{`x`,`x_{min}`,`x_{max}`},{`y`,`y_{min}`,`y_{max}`}]`	the multiple integral $\int_{x_{min}}^{x_{max}} dx \int_{y_{min}}^{y_{max}} dy\, f$

Integration functions.

Here is the integral $\int_a^b x^2\, dx$.

```
In[1]:= Integrate[x^2, {x, a, b}]
```

$$\text{Out[1]}= -\frac{a^3}{3} + \frac{b^3}{3}$$

This gives the multiple integral $\int_0^a dx \int_0^b dy\left(x^2 + y^2\right)$.

```
In[2]:= Integrate[x^2 + y^2, {x, 0, a}, {y, 0, b}]
```

$$\text{Out[2]}= \frac{1}{3} a\, b\, \left(a^2 + b^2\right)$$

The `y` integral is done first. Its limits can depend on the value of `x`. This ordering is the same as is used in functions like `Sum` and `Table`.

```
In[3]:= Integrate[x^2 + y^2, {x, 0, a}, {y, 0, x}]
```

$$\text{Out[3]}= \frac{a^4}{3}$$

In simple cases, definite integrals can be done by finding indefinite forms and then computing appropriate limits. But there is a vast range of integrals for which the indefinite form cannot be expressed in terms of standard mathematical functions, but the definite form still can be.

This indefinite integral cannot be done in terms of standard mathematical functions.

```
In[4]:= Integrate[Cos[Sin[x]], x]
```

```
Out[4]= ∫Cos[Sin[x]] dx
```

This definite integral, however, can be done in terms of a Bessel function.

```
In[5]:= Integrate[Cos[Sin[x]], {x, 0, 2 Pi}]
```

```
Out[5]= 2 π BesselJ[0, 1]
```

Here is an integral where the indefinite form can be found, but it is much more efficient to work out the definite form directly.

```
In[6]:= Integrate[Log[x] Exp[-x^2], {x, 0, Infinity}]
```

$$\text{Out[6]}= -\frac{1}{4}\sqrt{\pi}\ (\text{EulerGamma} + \text{Log}[4])$$

Just because an integrand may contain special functions, it does not mean that the definite integral will necessarily be complicated.

```
In[7]:= Integrate[BesselK[0, x]^2, {x, 0, Infinity}]
```

$$\text{Out[7]}= \frac{\pi^2}{4}$$

Special functions nevertheless occur in this result.

```
In[8]:= Integrate[BesselK[0, x] BesselJ[0, x], {x, 0, Infinity}]
```

$$\text{Out[8]}= \frac{\text{Gamma}\left[\frac{1}{4}\right]^2}{4\sqrt{2\pi}}$$

The integrand here is simple, but the definite integral is not.

```
In[9]:= Integrate[Sin[x^2] Exp[-x], {x, 0, Infinity}]
```

$$\text{Out[9]}= \frac{1}{4}\left(-2\,\text{HypergeometricPFQ}\left[\{1\}, \left\{\frac{3}{4}, \frac{5}{4}\right\}, -\frac{1}{64}\right] + \sqrt{2\pi}\left(\text{Cos}\left[\frac{1}{4}\right] + \text{Sin}\left[\frac{1}{4}\right]\right)\right)$$

Even when you can find the indefinite form of an integral, you will often not get the correct answer for the definite integral if you just subtract the values of the limits at each end point. The problem is that within the domain of integration there may be singularities whose effects are ignored if you follow this procedure.

Here is the indefinite integral of $1/x^2$.

```
In[10]:= Integrate[1 / x^2, x]
```

$$\text{Out[10]}= -\frac{1}{x}$$

This subtracts the limits at each end point.

```
In[11]:= Limit[%, x -> 2] - Limit[%, x -> -2]
```

```
Out[11]= -1
```

The true definite integral is divergent because of the double pole at $x = 0$.

```
In[12]:= Integrate[1 / x^2, {x, -2, 2}]
```

```
Integrate::idiv: Integral of 1/x^2 does not converge on {-2, 2}. >>
```

$$\text{Out[12]}= \int_{-2}^{2} \frac{1}{x^2}\, dx$$

Here is a more subtle example, involving branch cuts rather than poles.

```
In[13]:= Integrate[1 / (1 + a Sin[x]), x]
```

$$\text{Out[13]}= \frac{2\,\text{ArcTan}\left[\frac{a+\text{Tan}\left[\frac{x}{2}\right]}{\sqrt{1-a^2}}\right]}{\sqrt{1-a^2}}$$

Taking limits in the indefinite integral gives 0.

```
In[14]:= Limit[%, x -> 2 Pi] - Limit[%, x -> 0]
Out[14]= 0
```

The definite integral, however, gives the correct result which depends on a. The assumption assures convergence.

```
In[15]:= Integrate[1 / (1 + a Sin[x]), {x, 0, 2 Pi}, Assumptions -> -1 < a < 1]
```

$$\text{Out[15]}= \frac{2\,\pi}{\sqrt{1-a^2}}$$

`Integrate[`f`,{`x`,`x_{min}`,`x_{max}`},PrincipalValue->True]`

the Cauchy principal value of a definite integral

Principal value integrals.

Here is the indefinite integral of $1/x$.

```
In[16]:= Integrate[1 / x, x]
Out[16]= Log[x]
```

Substituting in the limits -1 and $+2$ yields a strange result involving $i\,\pi$.

```
In[17]:= Limit[%, x -> 2] - Limit[%, x -> -1]
Out[17]= -i π + Log[2]
```

The ordinary Riemann definite integral is divergent.

```
In[18]:= Integrate[1 / x, {x, -1, 2}]
```

Integrate::idiv: Integral of $\frac{1}{x}$ does not converge on {−1, 2}. ≫

$$\text{Out[18]}= \int_{-1}^{2} \frac{1}{x}\,dx$$

The Cauchy principal value, however, is finite.

```
In[19]:= Integrate[1 / x, {x, -1, 2}, PrincipalValue -> True]
Out[19]= Log[2]
```

When parameters appear in an indefinite integral, it is essentially always possible to get results that are correct for almost all values of these parameters. But for definite integrals this is no longer the case. The most common problem is that a definite integral may converge only when the parameters that appear in it satisfy certain specific conditions.

This indefinite integral is correct for all $n \neq -1$.

```
In[20]:= Integrate[x^n, x]
```

$$\text{Out[20]}= \frac{x^{1+n}}{1+n}$$

For the definite integral, however, n must satisfy a condition in order for the integral to be convergent.

```
In[21]:= Integrate[x^n, {x, 0, 1}]
```

$$\text{Out[21]}= \text{If}\left[\text{Re}[n] > -1,\ \frac{1}{1+n},\ \text{Integrate}\left[x^n,\ \{x,\ 0,\ 1\},\ \text{Assumptions} \to \text{Re}[n] \leq -1\right]\right]$$

If n is replaced by 2, the condition is satisfied.

```
In[22]:= % /. n -> 2
```

$$\text{Out[22]}= \frac{1}{3}$$

option name	*default value*	
`GenerateConditions`	`Automatic`	whether to generate explicit conditions
`Assumptions`	`$Assumptions`	what relations about parameters to assume

Options for `Integrate`.

With the assumption $n > 2$, the result is always $1/(1+n)$.

```
In[23]:= Integrate[x^n, {x, 0, 1}, Assumptions -> (n > 2)]
```

$$\text{Out[23]}= \frac{1}{1+n}$$

Even when a definite integral is convergent, the presence of singularities on the integration path can lead to discontinuous changes when the parameters vary. Sometimes a single formula containing functions like `Sign` can be used to summarize the result. In other cases, however, an explicit `If` is more convenient.

The `If` here gives the condition for the integral to be convergent.

```
In[24]:= Integrate[Sin[a x] / x, {x, 0, Infinity}]
```

$$\text{Out[24]= If}\left[a \in \text{Reals}, \frac{1}{2}\pi\,\text{Sign}[a], \text{Integrate}\left[\frac{\text{Sin}[a\,x]}{x}, \{x, 0, \infty\}, \text{Assumptions} \to a \neq \text{Re}[a]\right]\right]$$

Here is the result assuming that a is real.

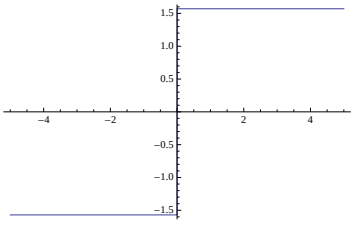
```
In[25]:= Integrate[Sin[a x] / x, {x, 0, Infinity}, Assumptions -> Im[a] == 0]
```

$$\text{Out[25]= } \frac{1}{2}\pi\,\text{Sign}[a]$$

The result is discontinuous as a function of a. The discontinuity can be traced to the essential singularity of $\sin(x)$ at $x = \infty$.

```
In[26]:= Plot[%, {a, -5, 5}]
```

Out[26]=



There is no convenient way to represent this answer in terms of `Sign`, so *Mathematica* generates an explicit `If`.

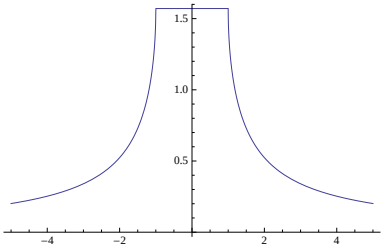
```
In[27]:= Integrate[Sin[x] BesselJ[0, a x] / x, {x, 0, Infinity}, Assumptions -> Im[a] == 0]
```

$$\text{Out[27]= If}\left[a < -1 \,||\, a > 1, \frac{a\,\text{ArcSin}\left[\frac{1}{a}\right]}{\text{Abs}[a]}, \frac{\pi}{2}\right]$$

Here is a plot of the resulting function of a.

```
In[28]:= Plot[Evaluate[%], {a, -5, 5}]
```

Out[28]=



Integrals over Regions

This does an integral over the interior of the unit circle.

```
In[1]:= Integrate[If[x^2 + y^2 < 1, 1, 0], {x, -1, 1}, {y, -1, 1}]
Out[1]= π
```

Here is an equivalent form.

```
In[2]:= Integrate[Boole[x^2 + y^2 < 1], {x, -1, 1}, {y, -1, 1}]
Out[2]= π
```

Even though an integral may be straightforward over a simple rectangular region, it can be significantly more complicated even over a circular region.

This gives a Bessel function.

```
In[3]:= Integrate[Exp[x] Boole[x^2 + y^2 < 1], {x, -1, 1}, {y, -1, 1}]
Out[3]= 2 π BesselI[1, 1]
```

`Integrate[`f `Boole[`$cond$`],{`x`,`x_{min}`,`x_{max}`},{`y`,`y_{min}`,`y_{max}`}]`
integrate f over the region where $cond$ is `True`

Integrals over regions.

Particularly if there are parameters inside the conditions that define regions, the results for integrals over regions may break into several cases.

This gives a piecewise function of a.

```
In[4]:= Integrate[Boole[a x < y], {x, 0, 1}, {y, 0, 1}]
Out[4]= ⎧ 1      a ≤ 0
        ⎪ 2-a
        ⎨ ---    0 < a ≤ 1
        ⎪  2
        ⎩ 1
        ⎪ ---    True
          2 a
```

With two parameters even this breaks into quite a few cases.

```
In[5]:= Integrate[Boole[a x < b], {x, 0, 1}]
Out[5]= ⎧ 1      (a > 0 && a - b ≤ 0) || (a ≤ 0 && b > 0)
        ⎪ a-b
        ⎨ ---    a ≤ 0 && a - b < 0 && b ≤ 0
        ⎪  a
        ⎩ b
        ⎪ ---    a > 0 && b > 0 && a - b > 0
          a
```

This involves intersecting a circle with a square.

```
In[6]:= Integrate[Boole[x^2 + y^2 < a], {x, 0, 1}, {y, 0, 1}]
```

$$\text{Out[6]}= \begin{cases} 1 & a \geq 2 \\ \frac{a\pi}{4} & 0 < a \leq 1 \\ \frac{1}{2}\left(2\sqrt{-1+a} + a\,\text{ArcCot}\left[\sqrt{-1+a}\right] - a\,\text{ArcTan}\left[\sqrt{-1+a}\right]\right) & 1 < a < 2 \end{cases}$$

The region can have an infinite number of components.

```
In[7]:= Integrate[Boole[Sin[x] > 1 / 2] Exp[-x], {x, 0, Infinity}]
```

$$\text{Out[7]}= \frac{e^{7\pi/6}}{1 + e^{2\pi/3} + e^{4\pi/3}}$$

Manipulating Integrals in Symbolic Form

When *Mathematica* cannot give you an explicit result for an integral, it leaves the integral in a symbolic form. It is often useful to manipulate this symbolic form.

Mathematica cannot give an explicit result for this integral, so it leaves the integral in symbolic form.

```
In[1]:= Integrate[x^2 f[x], x]
```

$$\text{Out[1]}= \int x^2 f[x]\, dx$$

Differentiating the symbolic form gives the integrand back again.

```
In[2]:= D[%, x]
```

```
Out[2]= x^2 f[x]
```

Here is a definite integral which cannot be done explicitly.

```
In[3]:= Integrate[f[x], {x, a[x], b[x]}]
```

$$\text{Out[3]}= \int_{a[x]}^{b[x]} f[x]\, dx$$

This gives the derivative of the definite integral.

```
In[4]:= D[%, x]
```

```
Out[4]= -f[a[x]] a'[x] + f[b[x]] b'[x]
```

Here is a definite integral with end points that do not explicitly depend on x.

```
In[5]:= defint = Integrate[f[x], {x, a, b}]
Out[5]= ∫_a^b f[x] dx
```

The partial derivative of this with respect to u is zero.

```
In[6]:= D[defint, u]
Out[6]= 0
```

There is a non-trivial total derivative, however.

```
In[7]:= Dt[defint, u]
Out[7]= -Dt[a, u] f[a] + Dt[b, u] f[b]
```

Differential Equations

You can use the *Mathematica* function DSolve to find symbolic solutions to ordinary and partial differential equations.

Solving a differential equation consists essentially in finding the form of an unknown function. In *Mathematica*, unknown functions are represented by expressions like y[x]. The derivatives of such functions are represented by y'[x], y''[x] and so on.

The *Mathematica* function DSolve returns as its result a list of rules for functions. There is a question of how these functions are represented. If you ask DSolve to solve for y[x], then DSolve will indeed return a rule for y[x]. In some cases, this rule may be all you need. But this rule, on its own, does not give values for y'[x] or even y[0]. In many cases, therefore, it is better to ask DSolve to solve not for y[x], but instead for y itself. In this case, what DSolve will return is a rule which gives y as a pure function, in the sense discussed in "Pure Functions".

If you ask DSolve to solve for y[x], it will give a rule specifically for y[x].

```
In[1]:= DSolve[y'[x] + y[x] == 1, y[x], x]
Out[1]= {{y[x] → 1 + e^-x C[1]}}
```

The rule applies only to y[x] itself, and not, for example, to objects like y[0] or y'[x].

```
In[2]:= y[x] + 2 y'[x] + y[0] /. %
Out[2]= {1 + e^-x C[1] + y[0] + 2 y'[x]}
```

If you ask `DSolve` to solve for `y`, it gives a rule for the object `y` on its own as a pure function.

```
In[3]:= DSolve[y'[x] + y[x] == 1, y, x]
Out[3]= {{y → Function[{x}, 1 + e^-x C[1]]}}
```

Now the rule applies to all occurrences of `y`.

```
In[4]:= y[x] + 2 y'[x] + y[0] /. %
Out[4]= {2 + C[1] - e^-x C[1]}
```

Substituting the solution into the original equation yields `True`.

```
In[5]:= y'[x] + y[x] == 1 /. %%
Out[5]= {True}
```

`DSolve[`*eqn*`,`*y*`[`*x*`],`*x*`]`	solve a differential equation for *y*[*x*]
`DSolve[`*eqn*`,`*y*`,`*x*`]`	solve a differential equation for the function *y*

Getting solutions to differential equations in different forms.

In standard mathematical notation, one typically represents solutions to differential equations by explicitly introducing "dummy variables" to represent the arguments of the functions that appear. If all you need is a symbolic form for the solution, then introducing such dummy variables may be convenient. However, if you actually intend to use the solution in a variety of other computations, then you will usually find it better to get the solution in pure-function form, without dummy variables. Notice that this form, while easy to represent in *Mathematica*, has no direct analog in standard mathematical notation.

`DSolve[{`*eqn*$_1$`,`*eqn*$_2$`,...},{`*y*$_1$`,`*y*$_2$`,...},`*x*`]`	solve a list of differential equations

Solving simultaneous differential equations.

This solves two simultaneous differential equations.

```
In[6]:= DSolve[{y[x] == -z'[x], z[x] == -y'[x]}, {y, z}, x]
Out[6]= {{z → Function[{x}, 1/2 e^-x (1 + e^2 x) C[1] - 1/2 e^-x (-1 + e^2 x) C[2]],
    y → Function[{x}, -1/2 e^-x (-1 + e^2 x) C[1] + 1/2 e^-x (1 + e^2 x) C[2]]}}
```

Mathematica returns two distinct solutions for `y` in this case.

```
In[7]:= DSolve[y[x] y'[x] == 1, y, x]
Out[7]= {{y → Function[{x}, -√2 √(x + C[1])]}, {y → Function[{x}, √2 √(x + C[1])]}}
```

You can add constraints and boundary conditions for differential equations by explicitly giving additional equations such as `y[0] == 0`.

This asks for a solution which satisfies the condition `y[0] == 1`.

```
In[8]:= DSolve[{y'[x] == a y[x], y[0] == 1}, y[x], x]
Out[8]= {{y[x] → e^(a x)}}
```

If you ask *Mathematica* to solve a set of differential equations and you do not give any constraints or boundary conditions, then *Mathematica* will try to find a *general solution* to your equations. This general solution will involve various undetermined constants. One new constant is introduced for each order of derivative in each equation you give.

The default is that these constants are named `C[n]`, where the index n starts at `1` for each invocation of `DSolve`. You can override this choice, by explicitly giving a setting for the option `GeneratedParameters`. Any function you give is applied to each successive index value n to get the constants to use for each invocation of `DSolve`.

The general solution to this fourth-order equation involves four undetermined constants.

```
In[9]:= DSolve[y''''[x] == y[x], y[x], x]
Out[9]= {{y[x] → e^x C[1] + e^-x C[3] + C[2] Cos[x] + C[4] Sin[x]}}
```

Each independent initial or boundary condition you give reduces the number of undetermined constants by one.

```
In[10]:= DSolve[{y''''[x] == y[x], y[0] == y'[0] == 0}, y[x], x]
Out[10]= {{y[x] → e^-x (C[3] + e^(2 x) C[3] - e^(2 x) C[4] - 2 e^x C[3] Cos[x] + e^x C[4] Cos[x] + e^x C[4] Sin[x])}}
```

You should realize that finding exact formulas for the solutions to differential equations is a difficult matter. In fact, there are only fairly few kinds of equations for which such formulas can be found, at least in terms of standard mathematical functions.

The most widely investigated differential equations are linear ones, in which the functions you are solving for, as well as their derivatives, appear only linearly.

This is a homogeneous first-order linear differential equation, and its solution is quite simple.

```
In[11]:= DSolve[y'[x] - x y[x] == 0, y[x], x]
```

$$\text{Out[11]}= \left\{\left\{y[x] \to e^{\frac{x^2}{2}} C[1]\right\}\right\}$$

Making the equation inhomogeneous leads to a significantly more complicated solution.

```
In[12]:= DSolve[y'[x] - x y[x] == 1, y[x], x]
```

$$\text{Out[12]}= \left\{\left\{y[x] \to e^{\frac{x^2}{2}} C[1] + e^{\frac{x^2}{2}} \sqrt{\frac{\pi}{2}}\ \text{Erf}\left[\frac{x}{\sqrt{2}}\right]\right\}\right\}$$

If you have only a single linear differential equation, and it involves only a first derivative of the function you are solving for, then it turns out that the solution can always be found just by doing integrals.

But as soon as you have more than one differential equation, or more than a first-order derivative, this is no longer true. However, some simple second-order linear differential equations can nevertheless be solved using various special functions from "Special Functions". Indeed, historically many of these special functions were first introduced specifically in order to represent the solutions to such equations.

This is Airy's equation, which is solved in terms of Airy functions.

```
In[13]:= DSolve[y''[x] - x y[x] == 0, y[x], x]
Out[13]= {{y[x] → AiryAi[x] C[1] + AiryBi[x] C[2]}}
```

This equation comes out in terms of Bessel functions.

```
In[14]:= DSolve[y''[x] - Exp[x] y[x] == 0, y[x], x]
```

$$\text{Out[14]}= \left\{\left\{y[x] \to \text{BesselI}\left[0, 2\sqrt{e^x}\right] C[1] + 2\,\text{BesselK}\left[0, 2\sqrt{e^x}\right] C[2]\right\}\right\}$$

This requires Mathieu functions.

```
In[15]:= DSolve[y''[x] + Cos[x] y[x] == 0, y, x]
```

$$\text{Out[15]}= \left\{\left\{y \to \text{Function}\left[\{x\}, C[1]\,\text{MathieuC}\left[0, -2, \frac{x}{2}\right] + C[2]\,\text{MathieuS}\left[0, -2, \frac{x}{2}\right]\right]\right\}\right\}$$

And this Legendre functions.

```
In[16]:= DSolve[y''[x] - Cot[x]^2 y[x] == 0, y[x], x]
```

$$\text{Out[16]}= \left\{\left\{y[x] \to C[1]\ \left(-1+\text{Cos}[x]^2\right)^{1/4}\ \text{LegendreP}\left[\frac{1}{2},\ \frac{\sqrt{5}}{2},\ \text{Cos}[x]\right] + C[2]\ \left(-1+\text{Cos}[x]^2\right)^{1/4}\ \text{LegendreQ}\left[\frac{1}{2},\ \frac{\sqrt{5}}{2},\ \text{Cos}[x]\right]\right\}\right\}$$

Occasionally second-order linear equations can be solved using only elementary functions.

```
In[17]:= DSolve[x^2 y''[x] + y[x] == 0, y[x], x]
```

$$\text{Out[17]}= \left\{\left\{y[x] \to \sqrt{x}\ C[1]\ \text{Cos}\left[\frac{1}{2}\sqrt{3}\ \text{Log}[x]\right] + \sqrt{x}\ C[2]\ \text{Sin}\left[\frac{1}{2}\sqrt{3}\ \text{Log}[x]\right]\right\}\right\}$$

Beyond second order, the kinds of functions needed to solve even fairly simple linear differential equations become extremely complicated. At third order, the generalized Meijer G function `MeijerG` can sometimes be used, but at fourth order and beyond absolutely no standard mathematical functions are typically adequate, except in very special cases.

Here is a third-order linear differential equation which can be solved in terms of generalized hypergeometric functions.

```
In[18]:= DSolve[y'''[x] + x y[x] == 0, y[x], x]
```

$$\text{Out[18]}= \left\{\left\{y[x] \to C[1]\ \text{HypergeometricPFQ}\left[\{\}, \left\{\frac{1}{2}, \frac{3}{4}\right\}, -\frac{x^4}{64}\right] + \frac{1}{2\sqrt{2}}\ x\ C[2]\ \text{HypergeometricPFQ}\left[\{\}, \left\{\frac{3}{4}, \frac{5}{4}\right\}, -\frac{x^4}{64}\right] + \frac{1}{8}\ x^2\ C[3]\ \text{HypergeometricPFQ}\left[\{\}, \left\{\frac{5}{4}, \frac{3}{2}\right\}, -\frac{x^4}{64}\right]\right\}\right\}$$

This requires more general Meijer G functions.

```
In[19]:= DSolve[y'''[x] + Exp[x] y[x] == 0, y[x], x]
```

$$\text{Out[19]}= \left\{\left\{y[x] \to C[1]\ \text{HypergeometricPFQ}\left[\{\}, \{1, 1\}, -e^x\right] + C[2]\ \text{MeijerG}\left[\{\{\}, \{\}\}, \{\{0, 0\}, \{0\}\}, -e^x\right] + C[3]\ \text{MeijerG}\left[\{\{\}, \{\}\}, \{\{0, 0, 0\}, \{\}\}, e^x\right]\right\}\right\}$$

For nonlinear differential equations, only rather special cases can usually ever be solved in terms of standard mathematical functions. Nevertheless, `DSolve` includes fairly general procedures which allow it to handle almost all nonlinear differential equations whose solutions are found in standard reference books.

First-order nonlinear differential equations in which x does not appear on its own are fairly easy to solve.

```
In[20]:= DSolve[y'[x] - y[x]^2 == 0, y[x], x]
```

$$\text{Out[20]= } \left\{\left\{y[x] \to \frac{1}{-x - C[1]}\right\}\right\}$$

This Riccati equation already gives a significantly more complicated solution.

```
In[21]:= DSolve[y'[x] - y[x]^2 == x, y[x], x] // FullSimplify
```

$$\text{Out[21]= } \left\{\left\{y[x] \to \left(\sqrt{x}\left(-\text{BesselJ}\left[-\frac{2}{3}, \frac{2x^{3/2}}{3}\right] + \text{BesselJ}\left[\frac{2}{3}, \frac{2x^{3/2}}{3}\right]C[1]\right)\right) \Big/ \left(\text{BesselJ}\left[\frac{1}{3}, \frac{2x^{3/2}}{3}\right] + \text{BesselJ}\left[-\frac{1}{3}, \frac{2x^{3/2}}{3}\right]C[1]\right)\right\}\right\}$$

This Bernoulli equation, however, has a fairly simple solution.

```
In[22]:= DSolve[y'[x] - x y[x]^2 - y[x] == 0, y[x], x]
```

$$\text{Out[22]= } \left\{\left\{y[x] \to -\frac{e^x}{-e^x + e^x x - C[1]}\right\}\right\}$$

An n^{th} order Bernoulli equation typically has $n-1$ distinct solutions.

```
In[23]:= DSolve[y'[x] - x y[x]^3 + y[x] == 0, y[x], x]
```

$$\text{Out[23]= } \left\{\left\{y[x] \to -\frac{\sqrt{2}}{\sqrt{1 + 2x + 2e^{2x}C[1]}}\right\}, \left\{y[x] \to \frac{\sqrt{2}}{\sqrt{1 + 2x + 2e^{2x}C[1]}}\right\}\right\}$$

This Abel equation can be solved, but only implicitly.

```
In[24]:= DSolve[y'[x] + x y[x]^3 + y[x]^2 == 0, y[x], x]
```

```
Solve::tdep :
  The equations appear to involve the variables to be solved for in an essentially non-algebraic way.
```

$$\text{Out[24]= } \text{Solve}\left[\frac{1}{2}\left(\frac{2\,\text{ArcTanh}\left[\frac{-1-2x\,y[x]}{\sqrt{5}}\right]}{\sqrt{5}} + \text{Log}\left[\frac{-1 - x\,y[x]\,(-1 - x\,y[x])}{x^2\,y[x]^2}\right]\right) == C[1] - \text{Log}[x], y[x]\right]$$

In practical applications, it is quite often convenient to set up differential equations that involve piecewise functions. You can use `DSolve` to find symbolic solutions to such equations.

This equation involves a piecewise forcing function.

```
In[25]:= DSolve[y'[x] - y[x] == UnitStep[x], y[x], x]
```

$$\text{Out[25]=}\ \left\{\left\{y[x] \to e^{x} C[1] + e^{x} \left(\begin{cases} 0 & x \leq 0 \\ 1 - e^{-x} & \text{True} \end{cases}\right)\right\}\right\}$$

Here the solution is explicitly broken into three cases.

```
In[26]:= DSolve[y'[x] + Clip[x] y[x] == 0, y[x], x]
```

$$\text{Out[26]=}\ \left\{\left\{y[x] \to e^{\begin{cases} x & x \leq -1 \\ -\frac{1}{2} - \frac{x^2}{2} & -1 < x \leq 1 \\ -x & \text{True} \end{cases}} C[1]\right\}\right\}$$

Beyond ordinary differential equations, one can consider *differential-algebraic equations* that involve a mixture of differential and algebraic equations.

This solves a differential-algebraic equation.

```
In[27]:= DSolve[{y'[x] + 3 z'[x] == 4 y[x] + 1 / x, y[x] + z[x] == 1}, {y[x], z[x]}, x]
```

$$\text{Out[27]=}\ \left\{\left\{y[x] \to \frac{3}{2} + \frac{1}{18}\left(-e^{-2x} C[1] - 9\, e^{-2x}\left(3\, e^{2x} + \text{ExpIntegralEi}[2x]\right)\right),\right.\right.$$
$$\left.\left. z[x] \to -\frac{1}{2} + \frac{1}{18}\left(e^{-2x} C[1] + 9\, e^{-2x}\left(3\, e^{2x} + \text{ExpIntegralEi}[2x]\right)\right)\right\}\right\}$$

DSolve[*eqn*, $y[x_1, x_2, \ldots]$, $\{x_1, x_2, \ldots\}$]	solve a partial differential equation for $y[x_1, x_2, \ldots]$
DSolve[*eqn*, *y*, $\{x_1, x_2, \ldots\}$]	solve a partial differential equation for the function y

Solving partial differential equations.

DSolve is set up to handle not only *ordinary differential equations* in which just a single independent variable appears, but also *partial differential equations* in which two or more independent variables appear.

This finds the general solution to a simple partial differential equation with two independent variables.

```
In[28]:= DSolve[D[y[x1, x2], x1] + D[y[x1, x2], x2] == 1 / (x1 x2), y[x1, x2], {x1, x2}]
```

$$\text{Out[28]=}\ \left\{\left\{y[x1, x2] \to \frac{1}{x1 - x2}\left(-\text{Log}[x1] + \text{Log}[x2] + x1\, C[1][-x1 + x2] - x2\, C[1][-x1 + x2]\right)\right\}\right\}$$

Here is the result represented as a pure function.

```
In[29]:= DSolve[D[y[x1, x2], x1] + D[y[x1, x2], x2] == 1 / (x1 x2), y, {x1, x2}]
```

$$\text{Out[29]= } \left\{\left\{y \to \text{Function}\left[\{x1, x2\}, \frac{1}{x1 - x2}\left(-\text{Log}[x1] + \text{Log}[x2] + x1\, C[1][-x1 + x2] - x2\, C[1][-x1 + x2]\right)\right]\right\}\right\}$$

The basic mathematics of partial differential equations is considerably more complicated than that of ordinary differential equations. One feature is that whereas the general solution to an ordinary differential equation involves only arbitrary *constants*, the general solution to a partial differential equation, if it can be found at all, must involve arbitrary *functions*. Indeed, with m independent variables, arbitrary functions of $m-1$ arguments appear. `DSolve` by default names these functions `C[`n`]`.

Here is a simple PDE involving three independent variables.

```
In[30]:= (D[#, x1] + D[#, x2] + D[#, x3]) &[y[x1, x2, x3]] == 0
```

$$\text{Out[30]= } y^{(0,0,1)}[x1, x2, x3] + y^{(0,1,0)}[x1, x2, x3] + y^{(1,0,0)}[x1, x2, x3] == 0$$

The solution involves an arbitrary function of two variables.

```
In[31]:= DSolve[%, y[x1, x2, x3], {x1, x2, x3}]
```

$$\text{Out[31]= } \{\{y[x1, x2, x3] \to C[1][-x1 + x2, -x1 + x3]\}\}$$

Here is the one-dimensional wave equation.

```
In[32]:= (c^2 D[#, x, x] - D[#, t, t]) &[y[x, t]] == 0
```

$$\text{Out[32]= } -y^{(0,2)}[x, t] + c^2\, y^{(2,0)}[x, t] == 0$$

The solution to this second-order equation involves two arbitrary functions.

```
In[33]:= DSolve[%, y[x, t], {x, t}]
```

$$\text{Out[33]= } \left\{\left\{y[x, t] \to C[1]\left[t - \frac{\sqrt{c^2}\, x}{c^2}\right] + C[2]\left[t + \frac{\sqrt{c^2}\, x}{c^2}\right]\right\}\right\}$$

For an ordinary differential equation, it is guaranteed that a general solution must exist, with the property that adding initial or boundary conditions simply corresponds to forcing specific choices for arbitrary constants in the solution. But for partial differential equations this is no longer true. Indeed, it is only for linear partial differential and a few other special types that such general solutions exist.

Other partial differential equations can be solved only when specific initial or boundary values are given, and in the vast majority of cases no solutions can be found as exact formulas in terms of standard mathematical functions.

Since y and its derivatives appear only linearly here, a general solution exists.

```
In[34]:= DSolve[x1 D[y[x1, x2], x1] + x2 D[y[x1, x2], x2] == Exp[x1 x2], y[x1, x2], {x1, x2}]
```

$$\text{Out[34]= } \left\{\left\{\text{y[x1, x2]} \to \frac{1}{2}\left(\text{ExpIntegralEi[x1 x2]} + 2\,\text{C[1]}\left[\frac{\text{x2}}{\text{x1}}\right]\right)\right\}\right\}$$

This weakly nonlinear PDE turns out to have a general solution.

```
In[35]:= DSolve[D[y[x1, x2], x1] + D[y[x1, x2], x2] == Exp[y[x1, x2]], y[x1, x2], {x1, x2}]
Out[35]= {{y[x1, x2] → -Log[-x1 - C[1][-x1 + x2]]}}
```

Here is a nonlinear PDE which has no general solution.

```
In[36]:= DSolve[D[y[x1, x2], x1] D[y[x1, x2], x2] == a, y[x1, x2], {x1, x2}]
```

DSolve::nlpde :
Solution requested to nonlinear partial differential equation. Trying to build a complete integral.

$$\text{Out[36]= } \left\{\left\{\text{y[x1, x2]} \to \text{C[1]} + \frac{\text{a x1}}{\text{C[2]}} + \text{x2 C[2]}\right\}\right\}$$

Integral Transforms and Related Operations

Laplace Transforms

LaplaceTransform[*expr*, *t*, *s*]	the Laplace transform of *expr*
InverseLaplaceTransform[*expr*, *s*, *t*]	the inverse Laplace transform of *expr*

One-dimensional Laplace transforms.

The Laplace transform of a function $f(t)$ is given by $\int_0^\infty f(t)\, e^{-s\,t}\, dt$. The inverse Laplace transform of $F(s)$ is given for suitable γ by $\frac{1}{2\,\pi\,i}\int_{\gamma-i\,\infty}^{\gamma+i\,\infty} F(s)\, e^{s\,t}\, ds$.

Here is a simple Laplace transform.

```
In[1]:= LaplaceTransform[t^4 Sin[t], t, s]
```

$$\text{Out[1]}= \frac{24\left(1-10\,s^2+5\,s^4\right)}{\left(1+s^2\right)^5}$$

Here is the inverse.

```
In[2]:= InverseLaplaceTransform[%, s, t]
```

```
Out[2]= t^4 Sin[t]
```

Even simple transforms often involve special functions.

```
In[3]:= LaplaceTransform[1 / (1 + t^2), t, s]
```

$$\text{Out[3]}= \text{CosIntegral}[s]\,\text{Sin}[s]+\frac{1}{2}\text{Cos}[s]\,(\pi-2\,\text{SinIntegral}[s])$$

Here the result involves a Meijer G function.

```
In[4]:= LaplaceTransform[1 / (1 + t^3), t, s]
```

$$\text{Out[4]}= \frac{\text{MeijerG}\left[\left\{\left\{\frac{2}{3}\right\},\{\}\right\},\left\{\left\{0,\frac{1}{3},\frac{2}{3},\frac{2}{3}\right\},\{\}\right\},\frac{s^3}{27}\right]}{2\sqrt{3}\,\pi}$$

`InverseLaplaceTransform` returns the original function.

```
In[6]:= InverseLaplaceTransform[%, s, t]
```

$$\text{Out[6]}= \frac{1}{1+t^3}$$

The Laplace transform of this Bessel function just involves elementary functions.

```
In[5]:= LaplaceTransform[BesselJ[n, t], t, s]
```

$$\text{Out[5]}= \frac{\left(s+\sqrt{1+s^2}\right)^{-n}}{\sqrt{1+s^2}}$$

Laplace transforms have the property that they turn integration and differentiation into essentially algebraic operations. They are therefore commonly used in studying systems governed by differential equations.

Integration becomes multiplication by $1/s$ when one does a Laplace transform.

```
In[6]:= LaplaceTransform[Integrate[f[u], {u, 0, t}], t, s]
```

$$\text{Out[6]}= \frac{\text{LaplaceTransform}[f[t], t, s]}{s}$$

LaplaceTransform[*expr*, {t_1, t_2, ...}, {s_1, s_2, ...}]	the multidimensional Laplace transform of *expr*
InverseLaplaceTransform[*expr*, {s_1, s_2, ...}, {t_1, t_2, ...}]	the multidimensional inverse Laplace transform of *expr*

Multidimensional Laplace transforms.

Fourier Transforms

FourierTransform[*expr*, t, ω]	the Fourier transform of *expr*
InverseFourierTransform[*expr*, ω, t]	the inverse Fourier transform of *expr*

One-dimensional Fourier transforms.

Integral transforms can produce results that involve "generalized functions" such as HeavisideTheta.

```
In[1]:= FourierTransform[1 / (1 + t^4), t, ω]
```

$$\text{Out[1]}= \left(\frac{1}{4}+\frac{i}{4}\right) e^{-\frac{(1+i)\,\omega}{\sqrt{2}}} \sqrt{\pi}\left(e^{\sqrt{2}\,\omega}\left(-i+e^{i\sqrt{2}\,\omega}\right)\text{HeavisideTheta}[-\omega]+\left(1-i\,e^{i\sqrt{2}\,\omega}\right)\text{HeavisideTheta}[\omega]\right)$$

This finds the inverse.

```
In[2]:= InverseFourierTransform[%, ω, t]
```

$$\text{Out[2]}= \frac{1}{1+t^4}$$

In *Mathematica* the Fourier transform of a function $f(t)$ is by default defined to be $\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(t)\, e^{i\omega t}\, dt$. The inverse Fourier transform of $F(\omega)$ is similarly defined as $\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} F(\omega)\, e^{-i\omega t}\, d\omega$.

In different scientific and technical fields different conventions are often used for defining Fourier transforms. The option `FourierParameters` in *Mathematica* allows you to choose any of these conventions you want.

common convention	*setting*	*Fourier transform*	*inverse Fourier transform*
Mathematica default	`{0, 1}`	$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(t)\, e^{i\omega t}\, dt$	$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} F(\omega)\, e^{-i\omega t}\, d\omega$
pure mathematics	`{1, -1}`	$\int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt$	$\frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\, e^{i\omega t}\, d\omega$
classical physics	`{-1, 1}`	$\frac{1}{2\pi}\int_{-\infty}^{\infty} f(t)\, e^{i\omega t}\, dt$	$\int_{-\infty}^{\infty} F(\omega)\, e^{-i\omega t}\, d\omega$
modern physics	`{0, 1}`	$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(t)\, e^{i\omega t}\, dt$	$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} F(\omega)\, e^{-i\omega t}\, d\omega$
systems engineering	`{1, -1}`	$\int_{-\infty}^{\infty} f(t)\, e^{-i\omega t}\, dt$	$\frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\, e^{i\omega t}\, d\omega$
signal processing	`{0, -2 Pi}`	$\int_{-\infty}^{\infty} f(t)\, e^{-2\pi i\omega t}\, dt$	$\int_{-\infty}^{\infty} F(\omega)\, e^{2\pi i\omega t}\, d\omega$
general case	`{a, b}`	$\sqrt{(\lvert b\rvert)/(2\pi)^{1-a}}\int_{-\infty}^{\infty} f(t)\, e^{ib\omega t}\, dt$	$\sqrt{\frac{\lvert b\rvert}{(2\pi)^{1+a}}}\int_{-\infty}^{\infty} F(\omega)\, e^{-ib\omega t}\, d\omega$

Typical settings for `FourierParameters` with various conventions.

Here is a Fourier transform with the default choice of parameters.

```
In[3]:= FourierTransform[Exp[-t^2], t, ω]
```

Out[3]= $\frac{e^{-\frac{\omega^2}{4}}}{\sqrt{2}}$

Here is the same Fourier transform with the choice of parameters typically used in signal processing.

```
In[4]:= FourierTransform[Exp[-t^2], t, ω, FourierParameters -> {0, -2 Pi}]
```

Out[4]= $e^{-\pi^2\omega^2}\sqrt{\pi}$

`FourierSinTransform[`*expr*`,`*t*`,`ω`]`
Fourier sine transform

`FourierCosTransform[`*expr*`,`*t*`,`ω`]`
Fourier cosine transform

`InverseFourierSinTransform[`*expr*`,`ω`,`*t*`]`
inverse Fourier sine transform

`InverseFourierCosTransform[`*expr*`,`ω`,`*t*`]`
inverse Fourier cosine transform

Fourier sine and cosine transforms.

In some applications of Fourier transforms, it is convenient to avoid ever introducing complex exponentials. Fourier sine and cosine transforms correspond to integrating respectively with $\sin(\omega t)$ and $\cos(\omega t)$ instead of $\exp(i\omega t)$, and using limits 0 and ∞ rather than $-\infty$ and ∞.

Here are the Fourier sine and cosine transforms of e^{-t}.

```
In[5]:= {FourierSinTransform[Exp[-t], t, ω], FourierCosTransform[Exp[-t], t, ω]}
```

$$\text{Out[5]}= \left\{\frac{\sqrt{\frac{2}{\pi}}\ \omega}{1+\omega^2},\ \frac{\sqrt{\frac{2}{\pi}}}{1+\omega^2}\right\}$$

`FourierTransform[`*expr*`,{`t_1`,`t_2`,...},{`ω_1`,`ω_2`,...}]`
the multidimensional Fourier transform of *expr*

`InverseFourierTransform[`*expr*`,{`ω_1`,`ω_2`,...},{`t_1`,`t_2`,...}]`
the multidimensional inverse Fourier transform of *expr*

`FourierSinTransform[`*expr*`,{`t_1`,`t_2`,...},{`ω_1`,`ω_2`,...}]`,
`FourierCosTransform[`*expr*`,{`t_1`,`t_2`,...},{`ω_1`,`ω_2`,...}]`
the multidimensional sine and cosine Fourier transforms of *expr*

`InverseFourierSinTransform[`*expr*`,{`ω_1`,`ω_2`,...},{`t_1`,`t_2`,...}]`,
`InverseFourierCosTransform[`*expr*`,{`ω_1`,`ω_2`,...},{`t_1`,`t_2`,...}]`
the multidimensional inverse Fourier sine and cosine transforms of *expr*

Multidimensional Fourier transforms.

This evaluates a two-dimensional Fourier transform.

```
In[6]:= FourierTransform[(u v) ^ 2 Exp[-u ^ 2 - v ^ 2], {u, v}, {a, b}]
```

$$\text{Out[6]}= \frac{1}{32}\left(-2+a^2\right)\left(-2+b^2\right)e^{-\frac{a^2}{4}-\frac{b^2}{4}}$$

This inverts the transform.

```
In[7]:= InverseFourierTransform[%, {a, b}, {u, v}]
```

$$\text{Out[7]}= e^{-u^2-v^2}\,u^2\,v^2$$

Z Transforms

`ZTransform[`*expr*`,`*n*`,`*z*`]`	Z transform of *expr*
`InverseZTransform[`*expr*`,`*z*`,`*n*`]`	inverse Z transform of *expr*

Z transforms.

The Z transform of a function $f(n)$ is given by $\sum_{n=0}^{\infty} f(n)\, z^{-n}$. The inverse Z transform of $F(z)$ is given by the contour integral $\frac{1}{2\pi i} \oint F(z)\, z^{n-1}\, dz$. Z transforms are effectively discrete analogs of Laplace transforms. They are widely used for solving difference equations, especially in digital signal processing and control theory. They can be thought of as producing generating functions, of the kind commonly used in combinatorics and number theory.

This computes the Z transform of 2^{-n}.

```
In[1]:= ZTransform[2^-n, n, z]
```

$$\text{Out[1]}= \frac{2\,z}{-1+2\,z}$$

Here is the inverse Z transform.

```
In[2]:= InverseZTransform[%, z, n]
```

```
Out[2]= 2^-n
```

The generating function for $1/n!$ is an exponential function.

```
In[3]:= ZTransform[1 / n!, n, z]
```

$$\text{Out[3]}= e^{\frac{1}{z}}$$

Generalized Functions and Related Objects

In many practical situations it is convenient to consider limits in which a fixed amount of something is concentrated into an infinitesimal region. Ordinary mathematical functions of the kind normally encountered in calculus cannot readily represent such limits. However, it is possible to introduce *generalized functions* or *distributions* which can represent these limits in integrals and other types of calculations.

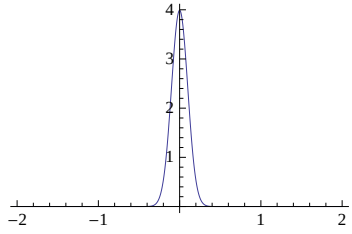
`DiracDelta[x]`	Dirac delta function $\delta(x)$
`HeavisideTheta[x]`	Heaviside theta function $\theta(x)$, equal to 0 for $x<0$ and 1 for $x>0$

Dirac delta and Heaviside theta functions.

Here is a function concentrated around $x=0$.

```
In[1]:= Plot[Sqrt[50 / Pi] Exp[-50 x^2], {x, -2, 2}, PlotRange -> All]
```

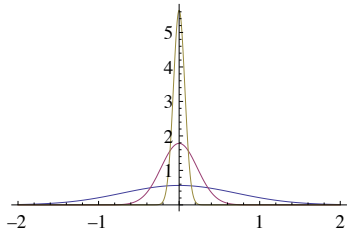
Out[1]=



As n gets larger, the functions become progressively more concentrated.

```
In[2]:= Plot[Evaluate[Sqrt[n / Pi] Exp[-n x^2] /. n -> {1, 10, 100}],
         {x, -2, 2}, PlotRange -> All]
```

Out[2]=



For any $n>0$, their integrals are nevertheless always equal to 1.

```
In[3]:= Integrate[Sqrt[n / Pi] Exp[-n x^2], {x, -Infinity, Infinity}, Assumptions -> n > 0]
Out[3]= 1
```

The limit of the functions for infinite n is effectively a Dirac delta function, whose integral is again 1.

```
In[4]:= Integrate[DiracDelta[x], {x, -Infinity, Infinity}]
Out[4]= 1
```

`DiracDelta` evaluates to 0 at all real points except $x=0$.

```
In[5]:= Table[DiracDelta[x], {x, -3, 3}]
Out[5]= {0, 0, 0, DiracDelta[0], 0, 0, 0}
```

Inserting a delta function in an integral effectively causes the integrand to be sampled at discrete points where the argument of the delta function vanishes.

This samples the function `f` with argument 2.

```
In[6]:= Integrate[DiracDelta[x - 2] f[x], {x, -4, 4}]
Out[6]= f[2]
```

Here is a slightly more complicated example.

```
In[7]:= Integrate[DiracDelta[x^2 - x - 1], {x, 0, 2}]
```

$$\text{Out[7]}= \frac{1}{\sqrt{5}}$$

This effectively counts the number of zeros of $\cos(x)$ in the region of integration.

```
In[8]:= Integrate[DiracDelta[Cos[x]], {x, -30, 30}]
Out[8]= 20
```

The *Heaviside function* `HeavisideTheta[x]` is the indefinite integral of the delta function. It is variously denoted $H(x)$, $\theta(x)$, $\mu(x)$, and $U(x)$. As a generalized function, the Heaviside function is defined only inside an integral. This distinguishes it from the unit step function `UnitStep[x]`, which is a piecewise function.

The indefinite integral of the delta function is the Heaviside theta function.

```
In[9]:= Integrate[DiracDelta[x], x]
Out[9]= HeavisideTheta[x]
```

The value of this integral depends on whether a lies in the interval $(-2, 2)$.

```
In[10]:= Integrate[f[x] DiracDelta[x - a], {x, -2, 2}, Assumptions -> a ∈ Reals]
Out[10]= f[a] HeavisideTheta[2 - a] HeavisideTheta[2 + a]
```

`DiracDelta` and `HeavisideTheta` often arise in doing integral transforms.

The Fourier transform of a constant function is a delta function.

```
In[11]:= FourierTransform[1, t, ω]
```

$$\text{Out[11]}= \sqrt{2\,\pi}\ \text{DiracDelta}[\omega]$$

The Fourier transform of $\cos(t)$ involves the sum of two delta functions.

```
In[12]:= FourierTransform[Cos[t], t, ω]
```

$$\text{Out[12]}= \sqrt{\frac{\pi}{2}}\ \text{DiracDelta}[-1+\omega] + \sqrt{\frac{\pi}{2}}\ \text{DiracDelta}[1+\omega]$$

Dirac delta functions can be used in `DSolve` to find the impulse response or Green's function of systems represented by linear and certain other differential equations.

This finds the behavior of a harmonic oscillator subjected to an impulse at $t = 0$.

```
In[13]:= DSolve[{x''[t] + r x[t] == DiracDelta[t], x[0] == 0, x'[0] == 1}, x[t], t]
```

$$\text{Out[13]}= \left\{\left\{\text{x[t]} \to \frac{\text{HeavisideTheta[t]}\ \text{Sin}\left[\sqrt{\text{r}}\ \text{t}\right]}{\sqrt{\text{r}}}\right\}\right\}$$

`DiracDelta`$[x_1, x_2, \ldots]$	multidimensional Dirac delta function
`HeavisideTheta`$[x_1, x_2, \ldots]$	multidimensional Heaviside theta function

Multidimensional Dirac delta and Heaviside theta functions.

Multidimensional generalized functions are essentially products of univariate generalized functions.

Here is a derivative of a multidimensional Heaviside function.

```
In[14]:= D[HeavisideTheta[x, y, z], x]
Out[14]= DiracDelta[x] HeavisideTheta[y, z]
```

Related to the multidimensional Dirac delta function are two integer functions: discrete delta and Kronecker delta. Discrete delta $\delta(n_1, n_2, \ldots)$ is 1 if all the $n_i = 0$, and is zero otherwise. Kronecker delta $\delta_{n_1 n_2 \ldots}$ is 1 if all the n_i are equal, and is zero otherwise.

`DiscreteDelta`$[n_1, n_2, \ldots]$	discrete delta $\delta(n_1, n_2, \ldots)$
`KroneckerDelta`$[n_1, n_2, \ldots]$	Kronecker delta $\delta_{n_1 n_2 \ldots}$

Integer delta functions.

Numerical Operations on Functions

Arithmetic

You can do arithmetic with *Mathematica* just as you would on an electronic calculator.

This is the sum of two numbers.

```
In[1]:= 2.3 + 5.63
Out[1]= 7.93
```

Here the / stands for division, and the ^ stands for power.

```
In[2]:= 2.4 / 8.9^2
Out[2]= 0.0302992
```

Spaces denote multiplication in *Mathematica*. The front end automatically replaces spaces between numbers with light gray multiplication signs.

```
In[3]:= 2 × 3 × 4
Out[3]= 24
```

You can use a * for multiplication if you want to.

```
In[4]:= 2 * 3 * 4
Out[4]= 24
```

You can type arithmetic expressions with parentheses.

```
In[5]:= (3 + 4)^2 - 2 (3 + 1)
Out[5]= 41
```

Spaces are not needed, though they often make your input easier to read.

```
In[6]:= (3 + 4)^2 - 2 (3 + 1)
Out[6]= 41
```

$x\text{^}y$	power
$-x$	minus
x/y	divide
$x\ y\ z$ or $x*y*z$	multiply
$x+y+z$	add

Arithmetic operations in *Mathematica*.

Arithmetic operations in *Mathematica* are grouped according to the standard mathematical conventions. As usual, `2^3+4`, for example, means `(2^3)+4`, and not `2^(3+4)`. You can always control grouping by explicitly using parentheses.

This result is given in scientific notation.

```
In[7]:= 2.4^45
```

$$\text{Out[7]= } 1.28678\times 10^{17}$$

You can enter numbers in scientific notation like this.

```
In[8]:= 2.3×10^70
```

$$\text{Out[8]= } 2.3\times 10^{70}$$

Or like this.

```
In[9]:= 2.3*^70
```

$$\text{Out[9]= } 2.3\times 10^{70}$$

With *Mathematica*, you can perform calculations with a particular precision, usually higher than an ordinary calculator. When given precise numbers, *Mathematica* does not convert them to an approximate representation, but gives a precise result.

This gives the result in terms of rational numbers.

```
In[10]:= 1/3+2/7
```

$$\text{Out[10]= } \frac{13}{21}$$

This gives the approximate numerical result.

```
In[11]:= 1/3+2/7 //N
Out[11]= 0.619048
```

This gives the approximate numerical result with 40 significant digits.

```
In[12]:= N[1 / 3 + 2 / 7, 40]
Out[12]= 0.6190476190476190476190476190476190476190
```

Numerical Mathematics in *Mathematica*

One of the important features of *Mathematica* is its ability to give you exact, symbolic, results for computations. There are, however, computations where it is just mathematically impossible to get exact "closed form" results. In such cases, you can still often get approximate numerical results.

There is no "closed form" result for $\int_0^1 \sin(\sin(x))\,dx$. *Mathematica* returns the integral in symbolic form.

```
In[1]:= Integrate[Sin[Sin[x]], {x, 0, 1}]
Out[1]= ∫_0^1 Sin[Sin[x]] dx
```

You can now take the symbolic form of the integral, and ask for its approximate numerical value.

```
In[2]:= N[%]
Out[2]= 0.430606
```

When *Mathematica* cannot find an explicit result for something like a definite integral, it returns a symbolic form. You can take this symbolic form, and try to get an approximate numerical value by applying `N`.

By giving a second argument to `N`, you can specify the numerical precision to use.

```
In[3]:= N[Integrate[Sin[Sin[x]], {x, 0, 1}], 40]
Out[3]= 0.4306061031206906049123773552484657864336
```

If you want to evaluate an integral numerically in *Mathematica*, then using `Integrate` and applying `N` to the result is not the most efficient way to do it. It is better instead to use the function `NIntegrate`, which immediately gives a numerical answer, without first trying to get an exact, symbolic, result. You should realize that even when `Integrate` does not in the end manage to give you an exact result, it may spend a lot of time trying to do so.

`NIntegrate` evaluates numerical integrals directly, without first trying to get a symbolic result.

```
In[4]:= NIntegrate[Sin[Sin[x]], {x, 0, 1}]
Out[4]= 0.430606
```

`Integrate`	`NIntegrate`	definite integrals
`Sum`	`NSum`	sums
`Product`	`NProduct`	products
`Solve`	`NSolve`	solutions of algebraic equations
`DSolve`	`NDSolve`	solutions of differential equations
`Maximize`	`NMaximize`	maximization

Symbolic and numerical versions of some *Mathematica* functions.

The Uncertainties of Numerical Mathematics

Mathematica does operations like numerical integration very differently from the way it does their symbolic counterparts.

When you do a symbolic integral, *Mathematica* takes the functional form of the integrand you have given, and applies a sequence of exact symbolic transformation rules to it, to try and evaluate the integral.

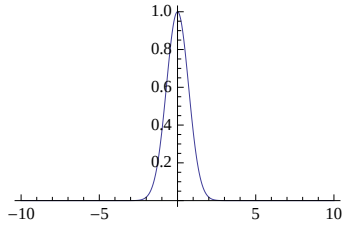
However, when *Mathematica* does a numerical integral, after some initial symbolic preprocessing, the only information it has about your integrand is a sequence of numerical values for it. To get a definite result for the integral, *Mathematica* then effectively has to make certain assumptions about the smoothness and other properties of your integrand. If you give a sufficiently pathological integrand, these assumptions may not be valid, and as a result, *Mathematica* may simply give you the wrong answer for the integral.

This problem may occur, for example, if you try to integrate numerically a function which has a very thin spike at a particular position. *Mathematica* samples your function at a number of points, and then assumes that the function varies smoothly between these points. As a result, if none of the sample points come close to the spike, then the spike will go undetected, and its contribution to the numerical integral will not be correctly included.

Here is a plot of the function $\exp(-x^2)$.

```
In[1]:= Plot[Exp[-x^2], {x, -10, 10}, PlotRange -> All]
```

Out[1]=



NIntegrate gives the correct answer for the numerical integral of this function from -10 to +10.

```
In[2]:= NIntegrate[Exp[-x^2], {x, -10, 10}]
Out[2]= 1.77245
```

If, however, you ask for the integral from -10000 to 10000, with its default settings NIntegrate will miss the peak near $x = 0$, and give the wrong answer.

```
In[3]:= NIntegrate[Exp[-x^2], {x, -10 000, 10 000}]

NIntegrate::ncvb: NIntegrate failed to converge to prescribed accuracy after 10 recursive bisections in x
near {x} = {6670.}. NIntegrate obtained 0.` and 0.` for the integral and error estimates. ≫

Out[3]= 0.
```

NIntegrate tries to make the best possible use of the information that it can get about the numerical values of the integrand. Thus, for example, by default, if NIntegrate notices that the estimated error in the integral in a particular region is large, it will take more samples in that region. In this way, NIntegrate tries to "adapt" its operation to the particular integrand you have given.

The kind of adaptive procedure that NIntegrate uses is similar, at least in spirit, to what Plot does in trying to draw smooth curves for functions. In both cases, *Mathematica* tries to go on taking more samples in a particular region until it has effectively found a smooth approximation to the function in that region.

The kinds of problems that can appear in numerical integration can also arise in doing other numerical operations on functions.

For example, if you ask for a numerical approximation to the sum of an infinite series, *Mathematica* samples a certain number of terms in the series, and then does an extrapolation to estimate the contributions of other terms. If you insert large terms far out in the series, they may not be detected when the extrapolation is done, and the result you get for the sum may be incorrect.

A similar problem arises when you try to find a numerical approximation to the minimum of a function. *Mathematica* samples only a finite number of values, then effectively assumes that the actual function interpolates smoothly between these values. If in fact the function has a sharp dip in a particular region, then *Mathematica* may miss this dip, and you may get the wrong answer for the minimum.

If you work only with numerical values of functions, there is simply no way to avoid the kinds of problems we have been discussing. Exact symbolic computation, of course, allows you to get around these problems.

In many calculations, it is therefore worthwhile to go as far as you can symbolically, and then resort to numerical methods only at the very end. This gives you the best chance of avoiding the problems that can arise in purely numerical computations.

Introduction to Numerical Sums, Products and Integrals

`NSum[`f`, {`i`, `i_{min}`, Infinity}]`	numerical approximation to $\sum_{i_{min}}^{\infty} f$
`NProduct[`f`, {`i`, `i_{min}`, Infinity}]`	numerical approximation to $\prod_{i_{min}}^{\infty} f$
`NIntegrate[`f`, {`x`, `x_{min}`, `x_{max}`}]`	numerical approximation to $\int_{x_{min}}^{x_{max}} f\, dx$
`NIntegrate[`f`, {`x`, `x_{min}`, `x_{max}`}, {`y`, `y_{min}`, `y_{max}`}]`	the multiple integral $\int_{x_{min}}^{x_{max}} dx \int_{y_{min}}^{y_{max}} dy\, f$

Numerical sums, products, and integrals.

Here is a numerical approximation to $\sum_{i=1}^{\infty} \frac{1}{i^3}$.

```
In[1]:= NSum[1 / i^3, {i, 1, Infinity}]
Out[1]= 1.20206
```

NIntegrate can handle singularities in the integration region.

```
In[2]:= NIntegrate[1 / Sqrt[x (1 - x)], {x, 0, 1}]
Out[2]= 3.14159
```

You can do numerical integrals over infinite regions.

```
In[3]:= NIntegrate[Exp[-x^2], {x, -Infinity, Infinity}]
Out[3]= 1.77245
```

Here is a double integral over a triangular domain. Note the order in which the variables are given.

```
In[4]:= NIntegrate[Sin[x y], {x, 0, 1}, {y, 0, x}]
Out[4]= 0.119906
```

Here is a double integral over a more complicated domain.

```
In[5]:= NIntegrate[ Sin[x y], {x, 0, 1}, {y, 0, Sqrt[x^3 + 3]}]
Out[5]= 0.727332
```

Numerical Integration

N[Integrate[*expr*, {*x*, x_{min}, x_{max}}]]	try to perform an integral exactly, then find numerical approximations to the parts that remain
NIntegrate[*expr*, {*x*, x_{min}, x_{max}}]	find a numerical approximation to an integral
NIntegrate[*expr*, {*x*, x_{min}, x_{max}}, {*y*, y_{min}, y_{max}}, …]	multidimensional numerical integral $\int_{x_{min}}^{x_{max}} dx \int_{y_{min}}^{y_{max}} dy \ldots expr$
NIntegrate[*expr*, {*x*, x_{min}, x_1, x_2, …, x_{max}}]	do a numerical integral along a line, starting at x_{min}, going through the points x_i, and ending at x_{max}

Numerical integration functions.

This finds a numerical approximation to the integral $\int_0^\infty e^{-x^3}\, dx$.

```
In[1]:= NIntegrate[Exp[-x^3], {x, 0, Infinity}]
Out[1]= 0.89298
```

Here is the numerical value of the double integral $\int_{-1}^{1} dx \int_{-1}^{1} dy \left(x^2 + y^2\right)$.

```
In[2]:= NIntegrate[x^2 + y^2, {x, -1, 1}, {y, -1, 1}]
Out[2]= 2.66667
```

An important feature of `NIntegrate` is its ability to deal with functions that "blow up" at known points. `NIntegrate` automatically checks for such problems at the endpoints of the integration region.

The function $1/\sqrt{x}$ blows up at $x = 0$, but `NIntegrate` still succeeds in getting the correct value for the integral.

```
In[3]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}]
Out[3]= 2.
```

Mathematica can find the integral of $1/\sqrt{x}$ exactly.

```
In[4]:= Integrate[1 / Sqrt[x], {x, 0, 1}]
Out[4]= 2
```

`NIntegrate` detects that the singularity in $1/x$ at $x = 0$ is not integrable.

```
In[5]:= NIntegrate[1 / x, {x, 0, 1}]

NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
    of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. >>

NIntegrate::ncvb :
  NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
    {x} = {1.22413×10^-225}. NIntegrate obtained 191612.2902185145`
    and 160378.51781028978` for the integral and error estimates. >>

Out[5]= 191612.
```

`NIntegrate` does not automatically look for singularities except at the endpoints of your integration region. When other singularities are present, `NIntegrate` may not give you the right answer for the integral. Nevertheless, in following its adaptive procedure, `NIntegrate` will often detect the presence of potentially singular behavior, and will warn you about it.

`NIntegrate` warns you of a possible problem due to the singularity in the middle of the integration region. The final result is numerically quite close to the correct answer.

```
In[6]:= NIntegrate[x^2 Sin[1 / x], {x, -1, 2}]

        NIntegrate::slwcon :
          Numerical integration converging too slowly; suspect one of the following: singularity, value
            of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. >>

Out[6]= 1.38755
```

If you know that your integrand has singularities at particular points, you can explicitly tell `NIntegrate` to deal with them. `NIntegrate[`*expr*`, {`*x*`, `x_{min}`, `x_1`, `x_2`, …, `x_{max}`}]` integrates *expr* from x_{min} to x_{max}, looking for possible singularities at each of the intermediate points x_i.

This gives the same integral, but now explicitly deals with the singularity at $x = 0$.

```
In[7]:= NIntegrate[x^2 Sin[1 / x], {x, -1, 0, 2}]
Out[7]= 1.38755
```

You can also use the list of intermediate points x_i in `NIntegrate` to specify an integration contour to follow in the complex plane. The contour is taken to consist of a sequence of line segments, starting at x_{min}, going through each of the x_i, and ending at x_{max}.

This integrates $1/x$ around a closed contour in the complex plane, going from -1, through the points $-i$, 1 and i, then back to -1.

```
In[8]:= NIntegrate[1 / x, {x, -1, -I, 1, I, -1}]
Out[8]= 0. + 6.28319 i
```

The integral gives $2\pi i$, as expected from Cauchy's theorem.

```
In[9]:= N[2 Pi I]
Out[9]= 0. + 6.28319 i
```

option name	*default value*	
`MinRecursion`	`0`	minimum number of recursions for the integration method
`MaxRecursion`	`Automatic`	maximum number of recursions for the integration method
`MaxPoints`	`Automatic`	maximum total number of times to sample the integrand

Special options for `NIntegrate`.

When NIntegrate tries to evaluate a numerical integral, it samples the integrand at a sequence of points. If it finds that the integrand changes rapidly in a particular region, then it recursively takes more sample points in that region. The parameters MinRecursion and MaxRecursion specify the minimum and maximum number of recursions to use. Increasing the value of MinRecursion guarantees that NIntegrate will use a larger number of sample points. MaxPoints and MaxRecursion limit the number of sample points which NIntegrate will ever try to use. Increasing MinRecursion or MaxRecursion will make NIntegrate work more slowly.

With the default settings for all options, NIntegrate misses the peak in $\exp\left(-(x-1)^2\right)$ near $x = 1$, and gives the wrong answer for the integral.

```
In[10]:= NIntegrate[Exp[-(x - 1)^2], {x, -1000, 1000}]

NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
    of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. >>

NIntegrate::ncvb :
  NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
    {x} = {3.87517}. NIntegrate obtained 1.6330510571683285` and
    0.004736564243403896` for the integral and error estimates. >>

Out[10]= 1.63305
```

With the option MinRecursion -> 3, NIntegrate samples enough points that it notices the peak around $x = 1$. With the default setting of MaxRecursion, however, NIntegrate cannot use enough sample points to be able to expect an accurate answer.

```
In[11]:= NIntegrate[Exp[-(x - 1)^2], {x, -50 000, 1000}, MinRecursion → 3]

NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
    of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. >>

NIntegrate::ncvb :
  NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
    {x} = {-8.44584}. NIntegrate obtained 1.8181913371063452`
    and 1.165089629798181` for the integral and error estimates. >>

Out[11]= 1.81819
```

With this setting of MaxRecursion, NIntegrate can get an accurate answer for the integral.

```
In[12]:= NIntegrate[Exp[-(x - 1)^2], {x, -50 000, 1000}, MinRecursion → 3, MaxRecursion → 20]

NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
    of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. >>

Out[12]= 1.77242
```

Another way to solve the problem is to make NIntegrate break the integration region into several pieces, with a small piece that explicitly covers the neighborhood of the peak.

```
In[13]:= NIntegrate[Exp[-(x-1)^2], {x, -1000, -10, 10, 1000}]
Out[13]= 1.77245
```

For integrals in many dimensions, it can take a long time for NIntegrate to get a precise answer. However, by setting the option MaxPoints, you can tell NIntegrate to give you just a rough estimate, sampling the integrand only a limited number of times.

Here is a way to get a rough estimate for an integral that takes a long time to compute.

```
In[14]:= NIntegrate[1 / Sqrt[x + Log[y + z]^2],
           {x, 0, 1}, {y, 0, 1}, {z, 0, 1}, MaxPoints -> 1000]
NIntegrate::maxp: The integral failed to converge after 1023 integrand evaluations. NIntegrate obtained
    1.4548878649546768` and 0.03247010762528413` for the integral and error estimates.
Out[14]= 1.45489
```

Numerical Evaluation of Sums and Products

NSum[f, {i, i_{min}, i_{max}}]	find a numerical approximation to the sum $\sum_{i=i_{min}}^{i_{max}} f$
NSum[f, {i, i_{min}, i_{maxdi}}]	use step di in the sum
NProduct[f, {i, i_{min}, i_{max}}]	find a numerical approximation to the product $\prod_{i=i_{min}}^{i_{max}} f$

Numerical sums and products.

This gives a numerical approximation to $\sum_{i=1}^{\infty} \frac{1}{i^3+i!}$.

```
In[1]:= NSum[1 / (i^3 + i!), {i, 1, Infinity}]
Out[1]= 0.64703
```

There is no exact result for this sum, so *Mathematica* leaves it in a symbolic form.

```
In[2]:= Sum[1 / (i^3 + i!), {i, 1, Infinity}]
```

$$\text{Out[2]=} \sum_{i=1}^{\infty} \frac{1}{i^3 + i!}$$

You can apply N explicitly to get a numerical result.

```
In[3]:= N[%]
Out[3]= 0.64703
```

The way NSum works is to include a certain number of terms explicitly, and then to try and estimate the contribution of the remaining ones. There are three approaches to estimating this contribution. The first uses the Euler-Maclaurin method, and is based on approximating the sum by an integral. The second method, known as the Wynn epsilon method, samples a number of additional terms in the sum, and then tries to fit them to a polynomial multiplied by a decaying exponential. The third approach, useful for alternating series, uses an alternating signs method; it also samples a number of additional terms and approximates their sum by the ratio of two polynomials (Padé approximation).

option name	*default value*	
Method	Automatic	Automatic, "EulerMaclaurin", "WynnEpsilon", or "AlternatingSigns"
NSumTerms	15	number of terms to include explicitly
VerifyConvergence	True	whether the convergence of the series should be verified

Special options for NSum.

If you do not explicitly specify the method to use, NSum will try to choose between the EulerMaclaurin or WynnEpsilon methods. In any case, some implicit assumptions about the functions you are summing have to be made. If these assumptions are not correct, you may get inaccurate answers.

The most common place to use NSum is in evaluating sums with infinite limits. You can, however, also use it for sums with finite limits. By making implicit assumptions about the objects you are evaluating, NSum can often avoid doing as many function evaluations as an explicit Sum computation would require.

This finds the numerical value of $\sum_{n=0}^{100} e^{-n}$ by extrapolation techniques.

```
In[4]:= NSum[Exp[-n], {n, 0, 100}]
Out[4]= 1.58198
```

You can also get the result, albeit much less efficiently, by constructing the symbolic form of the sum, then evaluating it numerically.

```
In[5]:= Sum[Exp[-n], {n, 0, 100}] // N
Out[5]= 1.58198
```

`NProduct` works in essentially the same way as `NSum`, with analogous options.

Numerical Equation Solving

`NSolve[`*lhs*`==`*rhs*`,`*x*`]`	solve a polynomial equation numerically
`NSolve[{`*lhs*$_1$`==`*rhs*$_1$`,`*lhs*$_2$`==`*rhs*$_2$`,…},{`*x*`,`*y*`,…}]`	solve a system of polynomial equations numerically
`FindRoot[`*lhs*`==`*rhs*`,{`*x*`,`x_0`}]`	search for a numerical solution to an equation, starting at $x = x_0$
`FindRoot[{`*lhs*$_1$`==`*rhs*$_1$`,`*lhs*$_2$`==`*rhs*$_2$`,…},{{`*x*`,`x_0`},{`*y*`,`y_0`},…}]`	search for numerical solutions to simultaneous equations

Numerical root finding.

`NSolve` gives you numerical approximations to all the roots of a polynomial equation.

```
In[1]:= NSolve[x^5 + x + 1 == 0, x]
Out[1]= {{x → -0.754878}, {x → -0.5 - 0.866025 i}, {x → -0.5 + 0.866025 i},
  {x → 0.877439 - 0.744862 i}, {x → 0.877439 + 0.744862 i}}
```

You can also use `NSolve` to solve sets of simultaneous equations numerically.

```
In[2]:= NSolve[{x + y == 2, x - 3 y + z == 3, x - y + z == 0}, {x, y, z}]
Out[2]= {{x → 3.5, y → -1.5, z → -5.}}
```

If your equations involve only linear functions or polynomials, then you can use `NSolve` to get numerical approximations to all the solutions. However, when your equations involve more complicated functions, there is in general no systematic procedure for finding all solutions, even numerically. In such cases, you can use `FindRoot` to search for solutions. You have to give `FindRoot` a place to start its search.

This searches for a numerical solution, starting at $x = 1$.

```
In[3]:= FindRoot[3 Cos[x] == Log[x], {x, 1}]
Out[3]= {x → 1.44726}
```

The equation has several solutions. If you start at a different x, FindRoot may return a different solution.

```
In[4]:= FindRoot[3 Cos[x] == Log[x], {x, 10}]
Out[4]= {x → 13.1064}
```

You can search for solutions to sets of equations. Here the solution involves complex numbers.

```
In[5]:= FindRoot[{x == Log[y], y == Log[x]}, {{x, I}, {y, 2}}]
Out[5]= {x → 0.318132 + 1.33724 i, y → 0.318132 + 1.33724 i}
```

Numerical Solution of Polynomial Equations

When Solve cannot find solutions in terms of radicals to polynomial equations, it returns a symbolic form of the result in terms of Root objects.

```
In[1]:= Solve[x^5 + 7 x + 1 == 0, x]
Out[1]= {{x → Root[1 + 7 #1 + #1^5 &, 1]}, {x → Root[1 + 7 #1 + #1^5 &, 2]},
  {x → Root[1 + 7 #1 + #1^5 &, 3]}, {x → Root[1 + 7 #1 + #1^5 &, 4]}, {x → Root[1 + 7 #1 + #1^5 &, 5]}}
```

You can get numerical solutions by applying N.

```
In[2]:= N[%]
Out[2]= {{x → -0.142849}, {x → -1.11308 - 1.15173 i},
  {x → -1.11308 + 1.15173 i}, {x → 1.1845 - 1.15139 i}, {x → 1.1845 + 1.15139 i}}
```

This gives the numerical solutions to 25-digit precision.

```
In[3]:= N[%%, 25]
Out[3]= {{x → -0.1428486455250044341134116},
  {x → -1.1130779765477107356003 98 - 1.1517343621516743050467 70 i},
  {x → -1.1130779765477107356003 98 + 1.1517343621516743050467 70 i},
  {x → 1.1845022993102129526571 04 - 1.1513900754088370746991 47 i},
  {x → 1.1845022993102129526571 04 + 1.1513900754088370746991 47 i}}
```

You can use NSolve to get numerical solutions to polynomial equations directly, without first trying to find exact results.

```
In[4]:= NSolve[x^7 + x + 1 == 0, x]
Out[4]= {{x → -0.796544}, {x → -0.705298 - 0.637624 i},
  {x → -0.705298 + 0.637624 i}, {x → 0.123762 - 1.05665 i},
  {x → 0.123762 + 1.05665 i}, {x → 0.979808 - 0.516677 i}, {x → 0.979808 + 0.516677 i}}
```

NSolve[*poly*==0,*x*]	get approximate numerical solutions to a polynomial equation
NSolve[*poly*==0,*x*,*n*]	get solutions using *n*-digit precision arithmetic
NSolve[{*eqn*$_1$,*eqn*$_2$,…},{*var*$_1$,*var*$_2$,…}]	get solutions to a polynomial system

Numerical solution of polynomial equations and systems.

NSolve will give you the complete set of numerical solutions to any polynomial equation or system of polynomial equations.

NSolve can find solutions to sets of simultaneous polynomial equations.

```
In[5]:= NSolve[{x^2 + y^2 == 1, x^3 + y^3 == 2}, {x, y}]

Out[5]= {{x → -1.09791 - 0.839887 i, y → -1.09791 + 0.839887 i},
         {x → -1.09791 + 0.839887 i, y → -1.09791 - 0.839887 i},
         {x → 1.22333 + 0.0729987 i, y → -0.125423 + 0.712005 i},
         {x → 1.22333 - 0.0729987 i, y → -0.125423 - 0.712005 i},
         {x → -0.125423 - 0.712005 i, y → 1.22333 - 0.0729987 i},
         {x → -0.125423 + 0.712005 i, y → 1.22333 + 0.0729987 i}}
```

Numerical Root Finding

NSolve gives you a general way to find numerical approximations to the solutions of polynomial equations. Finding numerical solutions to more general equations, however, can be much more difficult, as discussed in "Equations in One Variable". FindRoot gives you a way to search for a numerical root of a function or a numerical solution to an arbitrary equation, or set of equations.

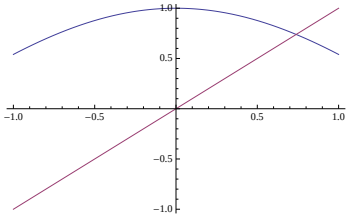
FindRoot[*f*,{*x*,*x*$_0$}]	search for a numerical root of *f*, starting with *x* = *x*$_0$
FindRoot[*lhs*==*rhs*,{*x*,*x*$_0$}]	search for a numerical solution to the equation *lhs* == *rhs*, starting with *x* = *x*$_0$
FindRoot[*f*$_1$,*f*$_2$,…,{{*x*,*x*$_0$},{*y*,*y*$_0$},…}]	search for a simultaneous numerical root of all the *f*$_i$
FindRoot[{*eqn*$_1$,*eqn*$_2$,…},{{*x*,*x*$_0$},{*y*,*y*$_0$},…}]	search for a numerical solution to the simultaneous equations *eqn*$_i$

Numerical root finding.

The curves for $\cos(x)$ and x intersect at one point.

```
In[1]:= Plot[{Cos[x], x}, {x, -1, 1}]
```

Out[1]=



This finds a numerical approximation to the value of x at which the intersection occurs. The `0` tells `FindRoot` what value of x to try first.

```
In[2]:= FindRoot[Cos[x] == x, {x, 0}]
Out[2]= {x → 0.739085}
```

In trying to find a solution to your equation, `FindRoot` starts at the point you specify, and then progressively tries to get closer and closer to a solution. Even if your equations have several solutions, `FindRoot` always returns the first solution it finds. Which solution this is will depend on what starting point you chose. If you start sufficiently close to a particular solution, `FindRoot` will usually return that solution.

The function $\sin(x)$ has an infinite number of roots of the form $x = n\pi$. If you start sufficiently close to a particular root, `FindRoot` will give you that root.

```
In[3]:= FindRoot[Sin[x], {x, 3}]
Out[3]= {x → 3.14159}
```

If you start with $x = 6$, you get a numerical approximation to the root $x = 2\pi$.

```
In[4]:= FindRoot[Sin[x], {x, 6}]
Out[4]= {x → 6.28319}
```

If you want `FindRoot` to search for complex solutions, then you have to give a complex starting value.

```
In[5]:= FindRoot[Sin[x] == 2, {x, I}]
Out[5]= {x → 1.5708 + 1.31696 i}
```

This finds a zero of the Riemann zeta function.

```
In[6]:= FindRoot[Zeta[1 / 2 + I t], {t, 12}]
Out[6]= {t → 14.1347 - 2.54839 × 10^-15 i}
```

This finds a solution to a set of simultaneous equations.

```
In[7]:= FindRoot[{Sin[x] == Cos[y], x + y == 1}, {{x, 1}, {y, 1}}]
Out[7]= {x → -1.85619, y → 2.85619}
```

The variables used by `FindRoot` can have values that are lists. This allows you to find roots of functions that take vectors as arguments.

This is a way to solve a linear equation for the variable x.

```
In[8]:= FindRoot[{{1, 2}, {3, 4}}.x == {5, 6}, {x, {1, 1}}]
Out[8]= {x → {-4., 4.5}}
```

This finds a normalized eigenvector x and eigenvalue a.

```
In[9]:= FindRoot[{{{1, 2}, {3, 4}}.x == a x, x.x == 1}, {{x, {1, 1}}, {a, 1}}]
Out[9]= {x → {0.415974, 0.909377}, a → 5.37228}
```

Introduction to Numerical Differential Equations

`NDSolve[`$eqns$`,`y`,{`x`,`x_{min}`,`x_{max}`}]`	solve numerically for the function y, with the independent variable x in the range x_{min} to x_{max}
`NDSolve[`$eqns$`,{`y_1`,`y_2`,...},{`x`,`x_{min}`,`x_{max}`}]`	solve a system of equations for the y_i

Numerical solution of differential equations.

This generates a numerical solution to the equation $y'(x) = y(x)$ with $0 < x < 2$. The result is given in terms of an `InterpolatingFunction`.

```
In[1]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x, 0, 2}]
Out[1]= {{y → InterpolatingFunction[{{0., 2.}}, <>]}}
```

Here is the value of $y(1.5)$.

```
In[2]:= y[1.5] /. %
Out[2]= {4.48169}
```

With an algebraic equation such as $x^2 + 3x + 1 = 0$, each solution for x is simply a single number. For a differential equation, however, the solution is a *function*, rather than a single number. For example, in the equation $y'(x) = y(x)$, you want to get an approximation to the function $y(x)$ as the independent variable x varies over some range.

Mathematica represents numerical approximations to functions as `InterpolatingFunction` objects. These objects are functions which, when applied to a particular x, return the approximate value of $y(x)$ at that point. The `InterpolatingFunction` effectively stores a table of values for $y(x_i)$, then interpolates this table to find an approximation to $y(x)$ at the particular x you request.

$y[x]$ `/.` *solution*	use the list of rules for the function y to get values for $y[x]$
`InterpolatingFunction[`*data*`][`x`]`	evaluate an interpolated function at the point x
`Plot[Evaluate[`$y[x]$ `/.` *solution*`],{`x`,`x_{min}`,`x_{max}`}]`	plot the solution to a differential equation

Using results from `NDSolve`.

This solves a system of two coupled differential equations.

```
In[3]:= NDSolve[{y'[x] == z[x], z'[x] == -y[x], y[0] == 0, z[0] == 1}, {y, z}, {x, 0, Pi}]
Out[3]= {{y → InterpolatingFunction[{{0., 3.14159}}, <>], z → InterpolatingFunction[{{0., 3.14159}}, <>]}}
```

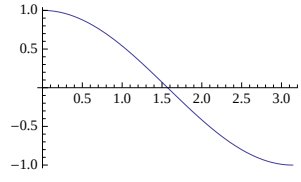
Here is the value of `z[2]` found from the solution.

```
In[4]:= z[2] /. %
Out[4]= {-0.416147}
```

Here is a plot of the solution for `z[x]` found on line 3. `Plot` is discussed in "Basic Plotting".

```
In[5]:= Plot[Evaluate[z[x] /. %3], {x, 0, Pi}]
```

Out[5]=



`NDSolve[`eqn`,`u`,{`x`,`x_{min}`,`x_{max}`},{`t`,`t_{min}`,`t_{max}`},...]`	solve a partial differential equation

Numerical solution of partial differential equations.

Numerical Solution of Differential Equations

The function NDSolve discussed in "Numerical Differential Equations" allows you to find numerical solutions to differential equations. NDSolve handles both single differential equations, and sets of simultaneous differential equations. It can handle a wide range of *ordinary differential equations* as well as some *partial differential equations*. In a system of ordinary differential equations there can be any number of unknown functions y_i, but all of these functions must depend on a single "independent variable" x, which is the same for each function. Partial differential equations involve two or more independent variables. NDSolve can also handle *differential-algebraic equations* that mix differential equations with algebraic ones.

NDSolve[{eqn_1, eqn_2, ...}, y, {x, x_{min}, x_{max}}]
find a numerical solution for the function y with x in the range x_{min} to x_{max}

NDSolve[{eqn_1, eqn_2, ...}, {y_1, y_2, ...}, {x, x_{min}, x_{max}}]
find numerical solutions for several functions y_i

Finding numerical solutions to ordinary differential equations.

NDSolve represents solutions for the functions y_i as InterpolatingFunction objects. The InterpolatingFunction objects provide approximations to the y_i over the range of values x_{min} to x_{max} for the independent variable x.

NDSolve finds solutions iteratively. It starts at a particular value of x, then takes a sequence of steps, trying eventually to cover the whole range x_{min} to x_{max}.

In order to get started, NDSolve has to be given appropriate initial or boundary conditions for the y_i and their derivatives. These conditions specify values for $y_i[x]$, and perhaps derivatives $y_i'[x]$, at particular points x. In general, at least for ordinary differential equations, the conditions you give can be at any x: NDSolve will automatically cover the range x_{min} to x_{max}.

This finds a solution for y with x in the range 0 to 2, using an initial condition for y[0].

```
In[1]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x, 0, 2}]
Out[1]= {{y → InterpolatingFunction[{{0., 2.}}, <>]}}
```

This still finds a solution with x in the range 0 to 2, but now the initial condition is for y[3].

```
In[2]:= NDSolve[{y'[x] == y[x], y[3] == 1}, y, {x, 0, 2}]
Out[2]= {{y → InterpolatingFunction[{{0., 2.}}, <>]}}
```

Here is a simple boundary value problem.

```
In[3]:= NDSolve[{y''[x] + x y[x] == 0, y[0] == 1, y[1] == -1}, y, {x, 0, 1}]
Out[3]= {{y → InterpolatingFunction[{{0., 1.}}, <>]}}
```

When you use NDSolve, the initial or boundary conditions you give must be sufficient to determine the solutions for the y_i completely. When you use DSolve to find symbolic solutions to differential equations, you can get away with specifying fewer initial conditions. The reason is that DSolve automatically inserts arbitrary constants C[i] to represent degrees of freedom associated with initial conditions that you have not specified explicitly. Since NDSolve must give a numerical solution, it cannot represent these kinds of additional degrees of freedom. As a result, you must explicitly give all the initial or boundary conditions that are needed to determine the solution.

In a typical case, if you have differential equations with up to n^{th} derivatives, then you need to give initial conditions for up to $(n-1)^{\text{th}}$ derivatives, or give boundary conditions at n points.

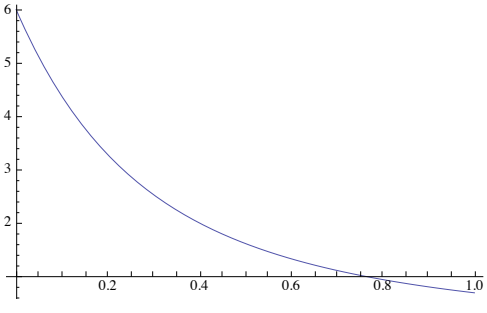
With a third-order equation, you need to give initial conditions for up to second derivatives.

```
In[4]:= NDSolve[{y'''[x] + 8 y''[x] + 17 y'[x] + 10 y[x] == 0,
          y[0] == 6, y'[0] == -20, y''[0] == 84}, y, {x, 0, 1}]
Out[4]= {{y → InterpolatingFunction[{{0., 1.}}, <>]}}
```

This plots the solution obtained.

```
In[5]:= Plot[Evaluate[y[x] /. %], {x, 0, 1}]
```

Out[5]=



With a third-order equation, you can also give boundary conditions at three points.

```
In[6]:= NDSolve[{y'''[x] + Sin[x] == 0, y[0] == 4, y[1] == 7, y[2] == 0}, y, {x, 0, 2}]
Out[6]= {{y → InterpolatingFunction[{{0., 2.}}, <>]}}
```

Mathematica allows you to use any appropriate linear combination of function values and derivatives as boundary conditions.

```
In[7]:= NDSolve[{y''[x] + y[x] == 12 x, 2 y[0] - y'[0] == -1, 2 y[1] + y'[1] == 9}, y, {x, 0, 1}]
Out[7]= {{y → InterpolatingFunction[{{0., 1.}}, <>]}}
```

In most cases, all the initial conditions you give must involve the same value of x, say x_0. As a result, you can avoid giving both x_{min} and x_{max} explicitly. If you specify your range of x as $\{x, x_1\}$, then *Mathematica* will automatically generate a solution over the range x_0 to x_1.

This generates a solution over the range 0 to 2.

```
In[8]:= NDSolve[{y'[x] == y[x], y[0] == 1}, y, {x, 2}]
Out[8]= {{y → InterpolatingFunction[{{0., 2.}}, <>]}}
```

You can give initial conditions as equations of any kind. In some cases, these equations may have multiple solutions. In such cases, `NDSolve` will correspondingly generate multiple solutions.

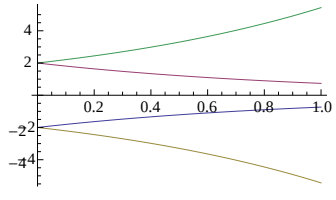
The initial conditions in this case lead to multiple solutions.

```
In[9]:= NDSolve[{y'[x]^2 - y[x]^2 == 0, y[0]^2 == 4}, y[x], {x, 1}]
Out[9]= {{y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]},
         {y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]},
         {y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]},
         {y[x] → InterpolatingFunction[{{0., 1.}}, <>][x]}}
```

Here is a plot of all the solutions.

```
In[10]:= Plot[Evaluate[y[x] /. %], {x, 0, 1}]
```

Out[10]=



You can use `NDSolve` to solve systems of coupled differential equations.

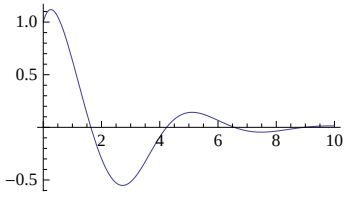
This finds a numerical solution to a pair of coupled equations.

```
In[11]:= sol = NDSolve[
           {x'[t] == -y[t] - x[t]^2, y'[t] == 2 x[t] - y[t], x[0] == y[0] == 1}, {x, y}, {t, 10}]
Out[11]= {{x → InterpolatingFunction[{{0., 10.}}, <>], y → InterpolatingFunction[{{0., 10.}}, <>]}}
```

This plots the solution for y from these equations.

```
In[12]:= Plot[Evaluate[y[t] /. sol], {t, 0, 10}, PlotRange → All]
```

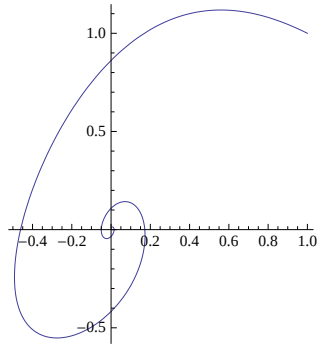
Out[12]=



This generates a parametric plot using both x and y.

```
In[13]:= ParametricPlot[Evaluate[{x[t], y[t]} /. sol], {t, 0, 10}, PlotRange -> All]
```

Out[13]=



Unknown functions in differential equations do not necessarily have to be represented by single symbols. If you have a large number of unknown functions, you will often find it more convenient, for example, to give the functions names like $y[i]$.

This constructs a set of five coupled differential equations and initial conditions.

```
In[14]:= eqns = Join[Table[y[i]'[x] == y[i - 1][x] - y[i][x], {i, 2, 4}],
            {y[1]'[x] == -y[1][x], y[5]'[x] == y[4][x], y[1][0] == 1},
            Table[y[i][0] == 0, {i, 2, 5}]]
Out[14]= {y[2]′[x] == y[1][x] - y[2][x], y[3]′[x] == y[2][x] - y[3][x],
          y[4]′[x] == y[3][x] - y[4][x], y[1]′[x] == -y[1][x], y[5]′[x] == y[4][x],
          y[1][0] == 1, y[2][0] == 0, y[3][0] == 0, y[4][0] == 0, y[5][0] == 0}
```

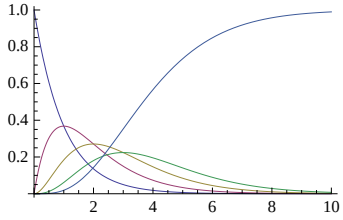
This solves the equations.

```
In[15]:= NDSolve[eqns, Table[y[i], {i, 5}], {x, 10}]
Out[15]= {{y[1] → InterpolatingFunction[{{0., 10.}}, <>],
           y[2] → InterpolatingFunction[{{0., 10.}}, <>], y[3] → InterpolatingFunction[{{0., 10.}}, <>],
           y[4] → InterpolatingFunction[{{0., 10.}}, <>], y[5] → InterpolatingFunction[{{0., 10.}}, <>]}}
```

Here is a plot of the solutions.

```
In[16]:= Plot[Evaluate[Table[y[i][x], {i, 5}] /. %], {x, 0, 10}]
```

Out[16]=



`NDSolve` can handle functions whose values are lists or arrays. If you give initial conditions like `y[0] == {`v_1`, `v_2`, ..., `v_n`}`, then `NDSolve` will assume that y is a function whose values are lists of length n.

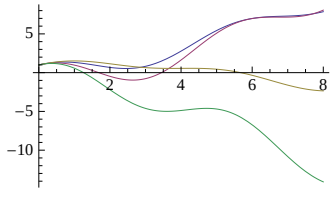
This solves a system of four coupled differential equations.

```
In[17]:= NDSolve[{y''[x] == -RandomReal[{0, 1}, {4, 4}].y[x],
           y[0] == y'[0] == Table[1, {4}]}, y, {x, 0, 8}]
Out[17]= {{y → InterpolatingFunction[{{0., 8.}}, <>]}}
```

Here are the solutions.

```
In[18]:= With[{s = y[x] /. First[%]},
          Plot[{s[[1]], s[[2]], s[[3]], s[[4]]}, {x, 0, 8}, PlotRange -> All]]
```

Out[18]=



option name	*default value*	
`MaxSteps`	`Automatic`	maximum number of steps in x to take
`StartingStepSize`	`Automatic`	starting size of step in x to use
`MaxStepSize`	`Automatic`	maximum size of step in x to use
`NormFunction`	`Automatic`	the norm to use for error estimation

Special options for `NDSolve`.

`NDSolve` has many methods for solving equations, but essentially all of them at some level work by taking a sequence of steps in the independent variable x, and using an adaptive procedure to determine the size of these steps. In general, if the solution appears to be varying rapidly in a particular region, then `NDSolve` will reduce the step size or change the method so as to be able to track the solution better.

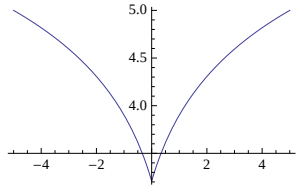
This solves a differential equation in which the derivative has a discontinuity.

```
In[19]:= NDSolve[{y'[x] == If[x < 0, 1 / (x - 1), 1 / (x + 1)], y[-5] == 5}, y, {x, -5, 5}]
Out[19]= {{y → InterpolatingFunction[{{-5., 5.}}, <>]}}
```

NDSolve reduced the step size around $x = 0$ so as to reproduce the kink accurately.

```
In[20]:= Plot[Evaluate[y[x] /. %], {x, -5, 5}]
```

Out[20]=



Through its adaptive procedure, NDSolve is able to solve "stiff" differential equations in which there are several components which vary with x at very different rates.

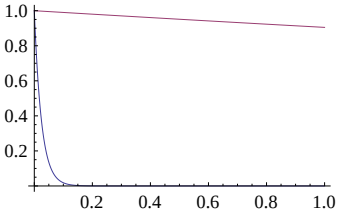
In these equations, y varies much more rapidly than z.

```
In[21]:= sol =
          NDSolve[{y'[x] == -40 y[x], z'[x] == -z[x] / 10, y[0] == z[0] == 1}, {y, z}, {x, 0, 1}]
Out[21]= {{y → InterpolatingFunction[{{0., 1.}}, <>], z → InterpolatingFunction[{{0., 1.}}, <>]}}
```

NDSolve nevertheless tracks both components successfully.

```
In[22]:= Plot[Evaluate[{y[x], z[x]} /. sol], {x, 0, 1}, PlotRange -> All]
```

Out[22]=



NDSolve follows the general procedure of reducing step size until it tracks solutions accurately. There is a problem, however, when the true solution has a singularity. In this case, NDSolve might go on reducing the step size forever, and never terminate. To avoid this problem, the option MaxSteps specifies the maximum number of steps that NDSolve will ever take in attempting to find a solution. For ordinary differential equations the default setting is MaxSteps -> 10 000.

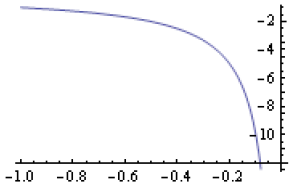
`NDSolve` stops after taking 10000 steps.

```
In[23]:= NDSolve[{y'[x] == -1 / x^2, y[-1] == -1}, y[x], {x, -1, 0}]

NDSolve::mxst: Maximum number of 10000 steps reached at the point x == -1.00413×10^-172. >>

Out[23]= {{y[x] → InterpolatingFunction[{{-1., -1.00413×10^-172}}, <>][x]}}
```

There is in fact a singularity in the solution at $x = 0$.

```
In[24]:= Plot[Evaluate[y[x] /. %], {x, -1, 0}]
```

Out[24]=



The default setting for `MaxSteps` should be sufficient for most equations with smooth solutions. When solutions have a complicated structure, however, you may occasionally have to choose larger settings for `MaxSteps`. With the setting `MaxSteps -> Infinity` there is no upper limit on the number of steps used.

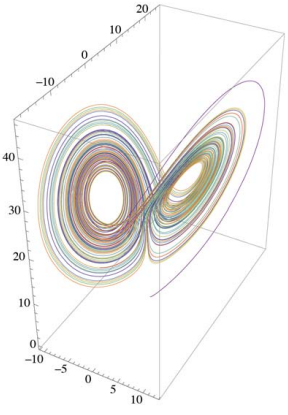
To take the solution to the Lorenz equations this far, you need to remove the default bound on `MaxSteps`.

```
In[25]:= NDSolve[{x'[t] == -3 (x[t] - y[t]), y'[t] == -x[t] z[t] + 26.5 x[t] - y[t],
    z'[t] == x[t] y[t] - z[t], x[0] == z[0] == 0, y[0] == 1},
   {x, y, z}, {t, 0, 200}, MaxSteps -> Infinity]

Out[25]= {{x → InterpolatingFunction[{{0., 200.}}, <>],
   y → InterpolatingFunction[{{0., 200.}}, <>], z → InterpolatingFunction[{{0., 200.}}, <>]}}
```

Here is a parametric plot of the solution in three dimensions.

```
In[26]:= ParametricPlot3D[Evaluate[{x[t], y[t], z[t]} /. %], {t, 0, 200},
  PlotPoints -> 10 000, ColorFunction → (ColorData["Rainbow"][#4] &)]
```

Out[26]=



When NDSolve solves a particular set of differential equations, it always tries to choose a step size appropriate for those equations. In some cases, the very first step that NDSolve makes may be too large, and it may miss an important feature in the solution. To avoid this problem, you can explicitly set the option StartingStepSize to specify the size to use for the first step.

The equations you give to NDSolve do not necessarily all have to involve derivatives; they can also just be algebraic. You can use NDSolve to solve many such *differential-algebraic equations*.

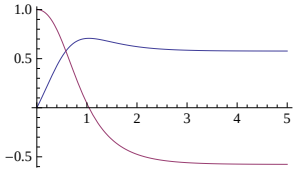
This solves a system of differential-algebraic equations.

```
In[27]:= NDSolve[{x'[t] == y[t]^2 + x[t] y[t],
    2 x[t]^2 + y[t]^2 == 1, x[0] == 0, y[0] == 1}, {x, y}, {t, 0, 5}]
Out[27]= {{x → InterpolatingFunction[{{0., 5.}}, <>], y → InterpolatingFunction[{{0., 5.}}, <>]}}
```

Here is the solution.

```
In[28]:= Plot[Evaluate[{x[t], y[t]} /. %], {t, 0, 5}]
```

Out[28]=



NDSolve[{*eqn*$_1$, *eqn*$_2$, ...}, *u*, {*t*, *t*$_{min}$, *t*$_{max}$}, {*x*, *x*$_{min}$, *x*$_{max}$}, ...]

solve a system of partial differential equations for *u*

NDSolve[{*eqn*$_1$, *eqn*$_2$, ...}, {*u*$_1$, *u*$_2$, ...}, {*t*, *t*$_{min}$, *t*$_{max}$}, {*x*, *x*$_{min}$, *x*$_{max}$}, ...]

solve a system of partial differential equations for several functions u_i

Finding numerical solutions to partial differential equations.

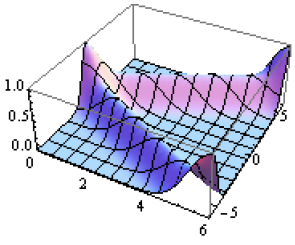
This finds a numerical solution to the wave equation. The result is a two-dimensional interpolating function.

```
In[29]:= NDSolve[{D[u[t, x], t, t] == D[u[t, x], x, x], u[0, x] == Exp[-x^2],
    Derivative[1, 0][u][0, x] == 0, u[t, -6] == u[t, 6]}, u, {t, 0, 6}, {x, -6, 6}]
Out[29]= {{u → InterpolatingFunction[{{0., 6.}, {..., -6., 6., ...}}, <>]}}
```

This generates a plot of the result.

In[30]:=
```
Plot3D[Evaluate[u[t, x] /. First[%]], {t, 0, 6}, {x, -6, 6}, PlotPoints -> 50]
```

Out[30]=



This finds a numerical solution to a nonlinear wave equation.

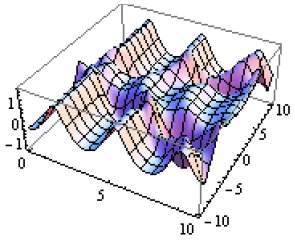
In[31]:=
```
NDSolve[{D[u[t, x], t, t] == D[u[t, x], x, x] + (1 - u[t, x]^2) (1 + 2 u[t, x]),
   u[0, x] == Exp[-x^2], Derivative[1, 0][u][0, x] == 0,
   u[t, -10] == u[t, 10]}, u, {t, 0, 10}, {x, -10, 10}]
```

Out[31]=
```
{{u → InterpolatingFunction[{{0., 10.}, {..., -10., 10., ...}}, <>]}}
```

Here is a 3D plot of the result.

In[32]:=
```
Plot3D[Evaluate[u[t, x] /. First[%]], {t, 0, 10}, {x, -10, 10}]
```

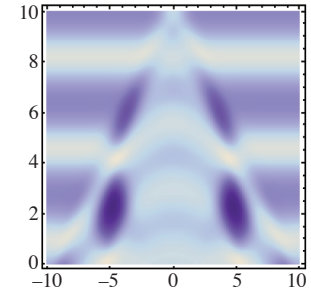
Out[32]=



This is a higher-resolution density plot of the solution.

In[33]:=
```
DensityPlot[Evaluate[u[10 - t, x] /. First[%%]],
 {x, -10, 10}, {t, 0, 10}, PlotPoints -> 200, Mesh -> False]
```

Out[33]=



Here is a version of the equation in 2+1 dimensions.

In[34]:=
```
eqn = D[u[t, x, y], t, t] ==
  D[u[t, x, y], x, x] + D[u[t, x, y], y, y] / 2 + (1 - u[t, x, y]^2) (1 + 2 u[t, x, y])
```

Out[34]=
$$u^{(2,0,0)}[t, x, y] == (1 + 2\,u[t, x, y])\left(1 - u[t, x, y]^2\right) + \frac{1}{2}u^{(0,0,2)}[t, x, y] + u^{(0,2,0)}[t, x, y]$$

This solves the equation.

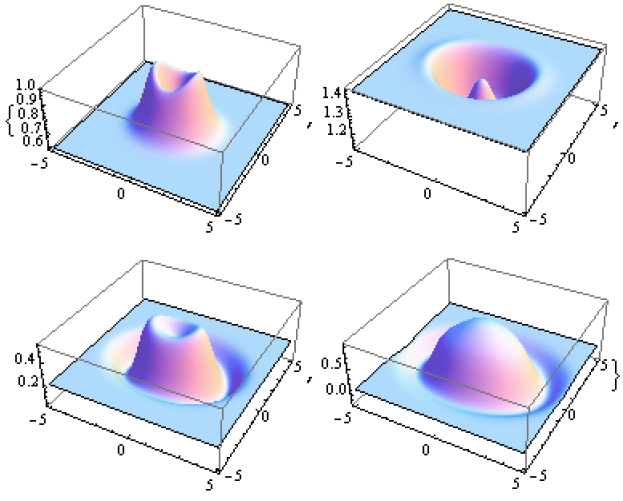
In[35]:=
```
NDSolve[{eqn, u[0, x, y] == Exp[-(x^2 + y^2)], u[t, -5, y] == u[t, 5, y],
  u[t, x, -5] == u[t, x, 5], Derivative[1, 0, 0][u][0, x, y] == 0},
 u, {t, 0, 4}, {x, -5, 5}, {y, -5, 5}]
```

Out[35]=
```
{{u → InterpolatingFunction[{{0., 4.}, {..., -5., 5., ...}, {..., -5., 5., ...}}, <>]}}
```

This generates a list of plots of the solution.

In[36]:=
```
Table[Plot3D[Evaluate[u[t, x, y] /. First[%]], {x, -5, 5}, {y, -5, 5},
  PlotRange -> All, PlotPoints -> 100, Mesh -> False], {t, 1, 4}]
```

Out[36]=



Numerical Optimization

FindMinimum[f, {x, x_0}]	search for a local minimum of f, starting at $x = x_0$
FindMinimum[f, x]	search for a local minimum of f
FindMinimum[f, {{x, x_0}, {y, y_0}, …}]	search for a local minimum in several variables
FindMinimum[{f, $cons$}, {{x, x_0}, {y, y_0}, …}]	search for a local minimum subject to the constraints $cons$ starting at $x = x_0$, $y = y_0$, …
FindMinimum[{f, $cons$}, {x, y, …}]	search for a local minimum subject to the constraints $cons$
FindMaximum[f, x], etc.	search for a local maximum

Searching for local minima and maxima.

This finds the value of x which minimizes $\Gamma(x)$, starting at $x = 2$.

```
In[1]:= FindMinimum[Gamma[x], {x, 2}]
Out[1]= {0.885603, {x → 1.46163}}
```

The last element of the list gives the value at which the minimum is achieved.

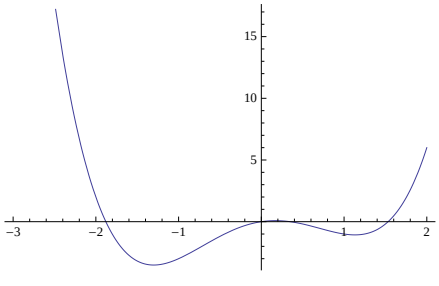
```
In[2]:= Gamma[x] /. Last[%]
Out[2]= 0.885603
```

Like FindRoot, FindMinimum and FindMaximum work by starting from a point, then progressively searching for a minimum or maximum. But since they return a result as soon as they find anything, they may give only a local minimum or maximum of your function, not a global one.

This curve has two local minima.

```
In[3]:= Plot[x^4 - 3 x^2 + x, {x, -3, 2}]
```

Out[3]=



Starting at $x = 1$, you get the local minimum on the right.

```
In[4]:= FindMinimum[x^4 - 3 x^2 + x, {x, 1}]
Out[4]= {-1.07023, {x → 1.1309}}
```

This gives the local minimum on the left, which in this case is also the global minimum.

```
In[5]:= FindMinimum[x^4 - 3 x^2 + x, {x, -1}]
Out[5]= {-3.51391, {x → -1.30084}}
```

You can specify variables without initial values.

```
In[6]:= FindMinimum[x^4 - 3 x^2 + x, x]
Out[6]= {-1.07023, {x → 1.1309}}
```

You can specify a constraint.

```
In[7]:= FindMinimum[{x^4 - 3 x^2 + x, x < 0}, x]
Out[7]= {-3.51391, {x → -1.30084}}
```

NMinimize[f, x]	try to find the global minimum of f
NMinimize[f, {x, y, ...}]	try to find the global minimum over several variables
NMaximize[f, x]	try to find the global maximum of f
NMaximize[f, {x, y, ...}]	try to find the global maximum over several variables

Finding global minima and maxima.

This immediately finds the global minimum.

```
In[8]:= NMinimize[x^4 - 3 x^2 + x, x]
Out[8]= {-3.51391, {x → -1.30084}}
```

NMinimize and NMaximize are numerical analogs of Minimize and Maximize. But unlike Minimize and Maximize they usually cannot guarantee to find absolute global minima and maxima. Nevertheless, they typically work well when the function f is fairly smooth, and has a limited number of local minima and maxima.

NMinimize[{*f*, *cons*}, {*x*, *y*, ...}]	try to find the global minimum of *f* subject to constraints *cons*
NMaximize[{*f*, *cons*}, {*x*, *y*, ...}]	try to find the global maximum of *f* subject to constraints *cons*

Finding global minima and maxima subject to constraints.

With the constraint $x > 0$, NMinimize will give the local minimum on the right.

```
In[9]:= NMinimize[{x^4 - 3 x^2 + x, x > 0}, x]
Out[9]= {-1.07023, {x → 1.1309}}
```

This finds the minimum of $x + 2y$ within the unit circle.

```
In[10]:= NMinimize[{x + 2 y, x^2 + y^2 <= 1}, {x, y}]
Out[10]= {-2.23607, {x → -0.447214, y → -0.894427}}
```

In this case Minimize can give an exact result.

```
In[11]:= Minimize[{x + 2 y, x^2 + y^2 <= 1}, {x, y}]
```

$$\text{Out[11]= } \left\{-\sqrt{5}\,,\ \left\{x \to \frac{4}{\sqrt{5}} - \sqrt{5}\,,\ y \to -\frac{2}{\sqrt{5}}\right\}\right\}$$

But in this case it cannot.

```
In[12]:= Minimize[{Cos[x + 2 y], x^2 + y^2 <= 1}, {x, y}]
Out[12]= Minimize[{Cos[x + 2 y], x^2 + y^2 ≤ 1}, {x, y}]
```

This gives a numerical approximation, effectively using NMinimize.

```
In[13]:= N[%]
Out[13]= {-0.617273, {x → 0.447214, y → 0.894427}}
```

If both the objective function f and the constraints *cons* are linear in all variables, then minimization and maximization correspond to a *linear programming problem*. Sometimes it is convenient to state such problems not in terms of explicit equations, but instead in terms of matrices and vectors.

LinearProgramming[*c*, *m*, *b*]	find the vector x which minimizes $c.x$ subject to the constraints $m.x \geq b$ and $x \geq 0$
LinearProgramming[*c*, *m*, *b*, *l*]	use the constraints $m.x \geq b$ and $x \geq l$

Linear programming in matrix form.

Here is a linear programming problem in equation form.

```
In[14]:= Minimize[{2 x + 3 y, x + 5 y >= 10, x - y >= 2, x >= 1}, {x, y}]
```

$$\text{Out[14]}= \left\{\frac{32}{3}, \left\{x \to \frac{10}{3}, y \to \frac{4}{3}\right\}\right\}$$

Here is the corresponding problem in matrix form.

```
In[15]:= LinearProgramming[{2, 3}, {{1, 5}, {1, -1}, {1, 0}}, {10, 2, 1}]
```

$$\text{Out[15]}= \left\{\frac{10}{3}, \frac{4}{3}\right\}$$

You can specify a mixture of equality and inequality constraints by making the list b be a sequence of pairs $\{b_i, s_i\}$. If s_i is 1, then the i^{th} constraint is $m_i . x \geq b_i$. If s_i is 0 then it is $m_i . x == b_i$, and if s_i is -1 then it is $m_i . x \leq b_i$.

This makes the first inequality use $\leq$.

```
In[16]:= LinearProgramming[{2, 3}, {{1, 5}, {1, -1}, {1, 0}}, {{10, -1}, {2, 1}, {1, 1}}]
Out[16]= {2, 0}
```

In `LinearProgramming[c, m, b, l]`, you can make l be a list of pairs $\{\{l_1, u_1\}, \{l_2, u_2\}, \ldots\}$ representing lower and upper bounds on the x_i.

In doing large linear programming problems, it is often convenient to give the matrix m as a `SparseArray` object.

Controlling the Precision of Results

In doing numerical operations like `NDSolve` and `NMinimize`, *Mathematica* by default uses machine numbers. But by setting the option `WorkingPrecision -> n` you can tell it to use arbitrary-precision numbers with n-digit precision.

This does a machine-precision computation of a numerical integral.

```
In[1]:= NIntegrate[Sin[Sin[x]], {x, 0, 1}]
Out[1]= 0.430606
```

This does the computation with 30-digit arbitrary-precision numbers.

```
In[2]:= NIntegrate[Sin[Sin[x]], {x, 0, 1}, WorkingPrecision -> 30]
Out[2]= 0.430606103120690604912377355248
```

When you give a setting for `WorkingPrecision`, this typically defines an upper limit on the precision of the results from a computation. But within this constraint you can tell *Mathematica* how much precision and accuracy you want it to try to get. You should realize that for many kinds of numerical operations, increasing precision and accuracy goals by only a few digits can greatly increase the computation time required. Nevertheless, there are many cases where it is important to ensure that high precision and accuracy are obtained.

`WorkingPrecision`	the number of digits to use for computations
`PrecisionGoal`	the number of digits of precision to try to get
`AccuracyGoal`	the number of digits of accuracy to try to get

Options for controlling precision and accuracy.

This gives a result to 25-digit precision.

```
In[3]:= NIntegrate[Sin[Sin[x]], {x, 0, 1}, WorkingPrecision -> 30, PrecisionGoal -> 25]
Out[3]= 0.430606103120690604912377355248
```

50-digit precision cannot be achieved with 30-digit working precision.

```
In[4]:= NIntegrate[Sin[Sin[x]], {x, 0, 1},
         WorkingPrecision -> 30, PrecisionGoal -> 50, MaxRecursion → 20]

   NIntegrate::slwcon:
     Numerical integration converging too slowly; suspect one of the following: singularity, value
       of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small.

   NIntegrate::eincr:
     The global error of the strategy GlobalAdaptive has increased more than 400 times. The global error is
       expected to decrease monotonically after a number of integrand evaluations. Suspect one
       of the following: the difference between the values of PrecisionGoal and WorkingPrecision
       is too small; the integrand is highly oscillatory or it is not a (piecewise) smooth function;
       or the true value of the integral is 0. Increasing the value of the GlobalAdaptive option
       MaxErrorIncreases might lead to a convergent numerical integration. NIntegrate obtained
       0.43060610312069060491237735524846578643219268469700477957788899453862440935`.
         086147`79.99999999999999 and
       5.03891680239785224285840796406833800958006097055414813173023183082827274593`.
         35312`79.99999999999999*^-40 for the integral and error estimates.
Out[4]= 0.430606103120690604912377355248
```

Given a particular setting for WorkingPrecision, each of the functions for numerical operations in *Mathematica* uses certain default settings for PrecisionGoal and AccuracyGoal. Typical is the case of NDSolve, in which these default settings are equal to half the settings given for WorkingPrecision.

The precision and accuracy goals normally apply both to the final results returned, and to various norms or error estimates for them. Functions for numerical operations in *Mathematica* typically try to refine their results until either the specified precision goal or accuracy goal is reached. If the setting for either of these goals is Infinity, then only the other goal is considered.

In doing ordinary numerical evaluation with N[*expr*, *n*], *Mathematica* automatically adjusts its internal computations to achieve n-digit precision in the result. But in doing numerical operations on functions, it is in practice usually necessary to specify WorkingPrecision and PrecisionGoal more explicitly.

Monitoring and Selecting Algorithms

Functions in *Mathematica* are carefully set up so that you normally do not have to know how they work inside. But particularly for numerical functions that use iterative algorithms, it is sometimes useful to be able to monitor the internal progress of these algorithms.

StepMonitor	an expression to evaluate whenever a successful step is taken
EvaluationMonitor	an expression to evaluate whenever functions from the input are evaluated

Options for monitoring progress of numerical functions.

This prints the value of x every time a step is taken.

```
In[1]:= FindRoot[Cos[x] == x, {x, 1}, StepMonitor :> Print[x]]

        0.750364

        0.739113

        0.739085

        0.739085

Out[1]= {x → 0.739085}
```

Note the importance of using *option* :> *expr* rather than *option* -> *expr*. You need a delayed rule :> to make *expr* be evaluated each time it is used, rather than just when the rule is given.

Reap and Sow provide a convenient way to make a list of the steps taken.

```
In[2]:= Reap[FindRoot[Cos[x] == x, {x, 1}, StepMonitor :> Sow[x]]]
Out[2]= {{x → 0.739085}, {{0.750364, 0.739113, 0.739085, 0.739085}}}
```

This counts the steps.

```
In[3]:= Block[{ct = 0}, {FindRoot[Cos[x] == x, {x, 1}, StepMonitor :> ct++], ct}]
Out[3]= {{x → 0.739085}, 4}
```

To take a successful step toward an answer, iterative numerical algorithms sometimes have to do several evaluations of the functions they have been given. Sometimes this is because each step requires, say, estimating a derivative from differences between function values, and sometimes it is because several attempts are needed to achieve a successful step.

This shows the successful steps taken in reaching the answer.

```
In[4]:= Reap[FindRoot[Cos[x] == x, {x, 5}, StepMonitor :> Sow[x]]]
Out[4]= {{x → 0.739085}, {{-1., -0.0283783, 1.02962, 0.752589, 0.739125, 0.739085, 0.739085}}}
```

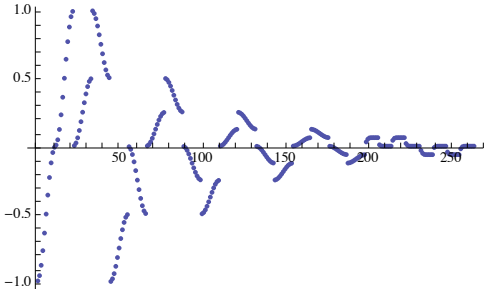
This shows every time the function was evaluated.

```
In[5]:= Reap[FindRoot[Cos[x] == x, {x, 5}, EvaluationMonitor :> Sow[x]]]
Out[5]= {{x → 0.739085},
         {{5., -55., -1., 8.71622, -0.0283783, 1.02962, 0.752589, 0.739125, 0.739085, 0.739085}}}
```

The pattern of evaluations done by algorithms in *Mathematica* can be quite complicated.

```
In[6]:= ListPlot[
          Reap[NIntegrate[1 / Sqrt[x], {x, -1, 0, 1}, EvaluationMonitor :> Sow[x]]][[2, 1]]]
```

Out[6]=



`Method->Automatic`	pick methods automatically (default)
`Method->"`*name*`"`	specify an explicit method to use
`Method->{"`*name*`",{"`par_1`"->`val_1`,...}}`	specify more details of a method

`Method` options.

There are often several different methods known for doing particular types of numerical computations. Typically *Mathematica* supports most generally successful ones that have been discussed in the literature, as well as many that have not. For any specific problem, it goes to considerable effort to pick the best method automatically. But if you have sophisticated knowledge of a problem, or are studying numerical methods for their own sake, you may find it useful to tell *Mathematica* explicitly what method it should use. Function reference pages list some of the methods built into *Mathematica*; others are discussed in "Numerical and Related Functions" or in advanced documentation.

This solves a differential equation using method m, and returns the number of steps and evaluations needed.

```
In[7]:= try[m_] := Block[{s = e = 0}, NDSolve[{y''[x] + Sin[y[x]] == 0, y'[0] == y[0] == 1}, y,
          {x, 0, 100}, StepMonitor :> s++, EvaluationMonitor :> e++, Method -> m]; {s, e}]
```

With the method selected automatically, this is the number of steps and evaluations that are needed.

```
In[8]:= try[Automatic]
Out[8]= {1118, 2329}
```

This shows what happens with several other possible methods. The Adams method that is selected automatically is the fastest.

```
In[9]:= try /@
          {"Adams", "BDF", "ExplicitRungeKutta", "ImplicitRungeKutta", "Extrapolation"}
Out[9]= {{1118, 2329}, {2415, 2861}, {287, 4595}, {882, 13092}, {84, 4146}}
```

This shows what happens with the explicit Runge-Kutta method when the difference order parameter is changed.

```
In[10]:= Table[try[{"ExplicitRungeKutta", "DifferenceOrder" -> n}], {n, 4, 9}]
Out[10]= {{3519, 14078}, {614, 4300}, {849, 6794}, {472, 4722}, {288, 3746}, {287, 4594}}
```

Functions with Sensitive Dependence on Their Input

Functions that are specified by simple algebraic formulas tend to be such that when their input is changed only slightly, their output also changes only slightly. But functions that are instead based on executing procedures quite often show almost arbitrarily sensitive dependence on their input. Typically the reason this happens is that the procedure "excavates" progressively less and less significant digits in the input.

This shows successive steps in a simple iterative procedure with input 0.1111.

```
In[1]:= NestList[FractionalPart[2 #] &, 0.1111, 10]
Out[1]= {0.1111, 0.2222, 0.4444, 0.8888, 0.7776, 0.5552, 0.1104, 0.2208, 0.4416, 0.8832, 0.7664}
```

Here is the result with input 0.1112. Progressive divergence from the result with input 0.1111 is seen.

```
In[2]:= NestList[FractionalPart[2 #] &, 0.1112, 10]
Out[2]= {0.1112, 0.2224, 0.4448, 0.8896, 0.7792, 0.5584, 0.1168, 0.2336, 0.4672, 0.9344, 0.8688}
```

The action of `FractionalPart[2 x]` is particularly simple in terms of the binary digits of the number x: it just drops the first one, and shifts the remaining ones to the left. After several steps, this means that the results one gets are inevitably sensitive to digits that are far to the right, and have an extremely small effect on the original value of x.

This shows the shifting process achieved by `FractionalPart[2 x]` in the first 8 binary digits of x.

```
In[3]:= RealDigits[Take[%, 5], 2, 8, -1]
Out[3]= {{{0, 0, 0, 1, 1, 1, 0, 0}, 0}, {{0, 0, 1, 1, 1, 0, 0, 0}, 0},
  {{0, 1, 1, 1, 0, 0, 0, 1}, 0}, {{1, 1, 1, 0, 0, 0, 1, 1}, 0}, {{1, 1, 0, 0, 0, 1, 1, 1}, 0}}
```

If you give input only to a particular precision, you are effectively specifying only a certain number of digits. And once all these digits have been "excavated" you can no longer get accurate results, since to do so would require knowing more digits of your original input. So long as you use arbitrary-precision numbers, *Mathematica* automatically keeps track of this kind of degradation in precision, indicating a number with no remaining significant digits by $0. \times 10^e$, as discussed in "Arbitrary-Precision Numbers".

Successive steps yield numbers of progressively lower precision, and eventually no precision at all.

```
In[4]:= NestList[FractionalPart[40 #] &, N[1 / 9, 20], 20]

Out[4]= {0.11111111111111111111, 0.44444444444444444444, 0.77777777777777778, 0.11111111111111111,
 0.44444444444444, 0.777777777778, 0.11111111111, 0.444444444, 0.77777778, 0.111111,
 0.4444, 0.778, 0.1, 0.×10^-1, 0.×10^1, 0.×10^3, 0.×10^4, 0.×10^6, 0.×10^7, 0.×10^9, 0.×10^11}
```

This asks for the precision of each number. Zero precision indicates that there are no correct significant digits.

```
In[5]:= Map[Precision, %]

Out[5]= {20., 19., 17.641, 15.1938, 14.1938, 12.8348, 10.3876, 9.38764,
 8.02862, 5.58146, 4.58146, 3.22244, 0.77528, 0., 0., 0., 0., 0., 0., 0.}
```

This shows that the exact result is a periodic sequence.

```
In[6]:= NestList[FractionalPart[40 #] &, 1 / 9, 10]

Out[6]= {1/9, 4/9, 7/9, 1/9, 4/9, 7/9, 1/9, 4/9, 7/9, 1/9, 4/9}
```

It is important to realize that if you use approximate numbers of any kind, then in an example like the one above you will always eventually run out of precision. But so long as you use arbitrary-precision numbers, *Mathematica* will explicitly show you any decrease in precision that is occurring. However, if you use machine-precision numbers, then *Mathematica* will not keep track of precision, and you cannot tell when your results become meaningless.

If you use machine-precision numbers, *Mathematica* will no longer keep track of any degradation in precision.

```
In[7]:= NestList[FractionalPart[40 #] &, N[1 / 9], 20]

Out[7]= {0.111111, 0.444444, 0.777778, 0.111111, 0.444444, 0.777778, 0.111111, 0.444445, 0.77781,
 0.112405, 0.496185, 0.847383, 0.89534, 0.813599, 0.543945, 0.757813, 0.3125, 0.5, 0., 0., 0.}
```

By iterating the operation `FractionalPart[2 x]` you extract successive binary digits in whatever number you start with. And if these digits are apparently random—as in a number like π—then the results will be correspondingly random. But if the digits have a simple pattern—as in any rational number—then the results you get will be correspondingly simple.

By iterating an operation such as `FractionalPart[3 / 2 x]` it turns out however to be possible to get seemingly random sequences even from very simple input. This is an example of a very general phenomenon first identified by Stephen Wolfram in the mid-1980s, which has nothing directly to do with sensitive dependence on input.

This generates a seemingly random sequence, even starting from simple input.

```
In[8]:= NestList[FractionalPart[3 / 2 #] &, 1, 15]
```

$$\text{Out[8]}= \left\{1, \frac{1}{2}, \frac{3}{4}, \frac{1}{8}, \frac{3}{16}, \frac{9}{32}, \frac{27}{64}, \frac{81}{128}, \frac{243}{256}, \frac{217}{512}, \frac{651}{1024}, \frac{1953}{2048}, \frac{1763}{4096}, \frac{5289}{8192}, \frac{15\,867}{16\,384}, \frac{14\,833}{32\,768}\right\}$$

After the values have been computed, one can safely find numerical approximations to them.

```
In[9]:= N[%]
Out[9]= {1., 0.5, 0.75, 0.125, 0.1875, 0.28125, 0.421875, 0.632813, 0.949219,
  0.423828, 0.635742, 0.953613, 0.43042, 0.64563, 0.968445, 0.452667}
```

Here are the last 5 results after 1000 iterations, computed using exact numbers.

```
In[10]:= Take[N[NestList[FractionalPart[3 / 2 #] &, 1, 1000]], -5]
Out[10]= {0.0218439, 0.0327659, 0.0491488, 0.0737233, 0.110585}
```

Using machine-precision numbers gives completely incorrect results.

```
In[11]:= Take[NestList[FractionalPart[3 / 2 #] &, 1., 1000], -5]
Out[11]= {0.670664, 0.0059966, 0.0089949, 0.0134924, 0.0202385}
```

Many kinds of iterative procedures yield functions that depend sensitively on their input. Such functions also arise when one looks at solutions to differential equations. In effect, varying the independent parameter in the differential equation is a continuous analog of going from one step to the next in an iterative procedure.

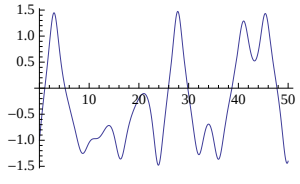
This finds a solution to the Duffing equation with initial condition 1.

```
In[12]:= NDSolve[{x''[t] + 0.15 x'[t] - x[t] + x[t]^3 == 0.3 Cos[t],
    x[0] == -1, x'[0] == 1}, x, {t, 0, 50}]
Out[12]= {{x → InterpolatingFunction[{{0., 50.}}, <>]}}
```

Here is a plot of the solution.

```
In[13]:= Plot[Evaluate[x[t] /. %], {t, 0, 50}]
```

Out[13]=



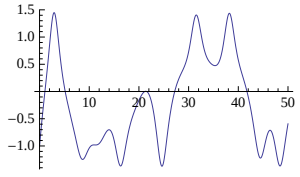
Here is the same equation with initial condition 1.001.

```
In[14]:= NDSolve[{x''[t] + 0.15 x'[t] - x[t] + x[t]^3 == 0.3 Cos[t],
           x[0] == -1, x'[0] == 1.001}, x, {t, 0, 50}]
Out[14]= {{x → InterpolatingFunction[{{0., 50.}}, <>]}}
```

The solution progressively diverges from the one shown above.

```
In[15]:= Plot[Evaluate[x[t] /. %], {t, 0, 50}]
```

Out[15]=



Numerical Operations on Data

Basic Statistics

`Mean[`*list*`]`	mean (average)
`Median[`*list*`]`	median (central value)
`Max[`*list*`]`	maximum value
`Variance[`*list*`]`	variance
`StandardDeviation[`*list*`]`	standard deviation
`Quantile[`*list*`,`*q*`]`	q^{th} quantile
`Total[`*list*`]`	total

Basic descriptive statistics operations.

Given a list with n elements x_i, the *mean* `Mean[`*list*`]` is defined to be $\mu(x) = \overline{x} = \sum x_i / n$.

The *variance* `Variance[`*list*`]` is defined to be $\text{var}(x) = \sigma^2(x) = \sum (x_i - \mu(x))^2 \big/ (n-1)$, for real data. (For complex data $\text{var}(x) = \sigma^2(x) = \sum (x_i - \mu(x))\,(\overline{x_i - \mu(x)}) \big/ (n-1)$.)

The *standard deviation* `StandardDeviation[`*list*`]` is defined to be $\sigma(x) = \sqrt{\text{var}(x)}$.

If the elements in *list* are thought of as being selected at random according to some probability distribution, then the mean gives an estimate of where the center of the distribution is located, while the standard deviation gives an estimate of how wide the dispersion in the distribution is.

The *median* `Median[`*list*`]` effectively gives the value at the halfway point in the sorted version of *list*. It is often considered a more robust measure of the center of a distribution than the mean, since it depends less on outlying values.

The q^{th} *quantile* `Quantile[`*list*`, `*q*`]` effectively gives the value that is q of the way through the sorted version of *list*.

For a list of length n, *Mathematica* defines `Quantile[`*list*`, `*q*`]` to be *s*`[[Ceiling[`*n q*`]]]`, where *s* is `Sort[`*list*`, Less]`.

There are, however, about ten other definitions of quantile in use, all potentially giving slightly different results. *Mathematica* covers the common cases by introducing four *quantile parameters* in the form `Quantile[list, q, {{a, b}, {c, d}}]`. The parameters a and b in effect define where in the list should be considered a fraction q of the way through. If this corresponds to an integer position, then the element at that position is taken to be the q^{th} quantile. If it is not an integer position, then a linear combination of the elements on either side is used, as specified by c and d.

The position in a sorted list s for the q^{th} quantile is taken to be $k = a + (n + b)\,q$. If k is an integer, then the quantile is s_k. Otherwise, it is $s_{\lfloor k \rfloor} + (s_{\lceil k \rceil} - s_{\lfloor k \rfloor})\,(c + d\,(k - \lfloor k \rfloor))$, with the indices taken to be 1 or n if they are out of range.

`{{0,0},{1,0}}`	inverse empirical CDF (default)
`{{0,0},{0,1}}`	linear interpolation (California method)
`{{1/2,0},{0,0}}`	element numbered closest to qn
`{{1/2,0},{0,1}}`	linear interpolation (hydrologist method)
`{{0,1},{0,1}}`	mean-based estimate (Weibull method)
`{{1,-1},{0,1}}`	mode-based estimate
`{{1/3,1/3},{0,1}}`	median-based estimate
`{{3/8,1/4},{0,1}}`	normal distribution estimate

Common choices for quantile parameters.

Whenever $d = 0$, the value of the q^{th} quantile is always equal to some actual element in *list*, so that the result changes discontinuously as q varies. For $d = 1$, the q^{th} quantile interpolates linearly between successive elements in *list*. `Median` is defined to use such an interpolation.

Note that `Quantile[list, q]` yields *quartiles* when $q = m/4$ and *percentiles* when $q = m/100$.

`Mean[{x1,x2,...}]`	the mean of the x_i
`Mean[{{x1,y1,...},{x2,y2,...},...}]`	a list of the means of the x_i, y_i, ...

Handling multidimensional data.

Sometimes each item in your data may involve a list of values. The basic statistics functions in *Mathematica* automatically apply to all corresponding elements in these lists.

This separately finds the mean of each "column" of data.

```
In[1]:= Mean[{{x1, y1}, {x2, y2}, {x3, y3}}]
Out[1]= {1/3 (x1 + x2 + x3), 1/3 (y1 + y2 + y3)}
```

Note that you can extract the elements in the i^{th} "column" of a multidimensional list using *list*`[[All, i]]`.

Descriptive Statistics

Descriptive statistics refers to properties of distributions, such as location, dispersion, and shape. The functions described here compute descriptive statistics of lists of data. You can calculate some of the standard descriptive statistics for various known distributions by using the functions described in "Continuous Distributions" and "Discrete Distributions".

The statistics are calculated assuming that each value of data x_i has probability equal to $\frac{1}{n}$, where n is the number of elements in the data.

`Mean[`*data*`]`	average value $\frac{1}{n}\sum_i x_i$
`Median[`*data*`]`	median (central value)
`Commonest[`*data*`]`	list of the elements with highest frequency
`GeometricMean[`*data*`]`	geometric mean $\left(\prod_i x_i\right)^{\frac{1}{n}}$
`HarmonicMean[`*data*`]`	harmonic mean $n \Big/ \sum_i \frac{1}{x_i}$
`RootMeanSquare[`*data*`]`	root mean square $\sqrt{\frac{1}{n}\sum_i x_i^2}$
`TrimmedMean[`*data*`,`f`]`	mean of remaining entries, when a fraction f is removed from each end of the sorted list of data
`TrimmedMean[`*data*`,{`f_1`,`f_2`}]`	mean of remaining entries, when fractions f_1 and f_2 are dropped from each end of the sorted data
`Quantile[`*data*`,`q`]`	q^{th} quantile
`Quartiles[`*data*`]`	list of the $\frac{1}{4}^{\text{th}}$, $\frac{1}{2}^{\text{th}}$, $\frac{3}{4}^{\text{th}}$ quantiles of the elements in *list*

Location statistics.

Location statistics describe where the data are located. The most common functions include measures of central tendency like the mean, median, and mode. `Quantile[data, q]` gives the location before which $(100\,q)$ percent of the data lie. In other words, `Quantile` gives a value z such that the probability that $(x_i < z)$ is less than or equal to q and the probability that $(x_i \le z)$ is greater than or equal to q.

Here is a dataset.

```
In[1]:= data = {6.5, 3.8, 6.6, 5.7, 6.0, 6.4, 5.3}
Out[1]= {6.5, 3.8, 6.6, 5.7, 6., 6.4, 5.3}
```

This finds the mean and median of the data.

```
In[2]:= {Mean[data], Median[data]}
Out[2]= {5.75714, 6.}
```

This is the mean when the smallest entry in the list is excluded. `TrimmedMean` allows you to describe the data with removed outliers.

```
In[3]:= TrimmedMean[data, {1/7, 0}]
Out[3]= 6.08333
```

`Variance[data]`	unbiased estimate of variance, $\frac{1}{n-1}\sum_i (x_i - \bar{x})^2$
`StandardDeviation[data]`	unbiased estimate of standard deviation
`MeanDeviation[data]`	mean absolute deviation, $\frac{1}{n}\sum_i \left\lvert x_i - \bar{x} \right\rvert$
`MedianDeviation[data]`	median absolute deviation, median of $\lvert x_i - median \rvert$ values
`InterquartileRange[data]`	difference between the first and third quartiles
`QuartileDeviation[data]`	half the interquartile range

Dispersion statistics.

Dispersion statistics summarize the scatter or spread of the data. Most of these functions describe deviation from a particular location. For instance, variance is a measure of deviation from the mean, and standard deviation is just the square root of the variance.

This gives an unbiased estimate for the variance of the data with $n - 1$ as the divisor.

```
In[4]:= Variance[data]
Out[4]= 0.962857
```

This compares three types of deviation.

```
In[5]:= {StandardDeviation[data], MeanDeviation[data], MedianDeviation[data]}
Out[5]= {0.981253, 0.706122, 0.5}
```

`Covariance[`v_1`,`v_2`]`	covariance coefficient between lists v_1 and v_2
`Covariance[`m`]`	covariance matrix for the matrix m
`Covariance[`m_1`,`m_2`]`	covariance matrix for the matrices m_1 and m_2
`Correlation[`v_1`,`v_2`]`	correlation coefficient between lists v_1 and v_2
`Correlation[`m`]`	correlation matrix for the matrix m
`Correlation[`m_1`,`m_2`]`	correlation matrix for the matrices m_1 and m_2

Covariance and correlation statistics.

Covariance is the multivariate extension of variance. For two vectors of equal length, the covariance is a number. For a single matrix m, the i, j^{th} element of the covariance matrix is the covariance between the i^{th} and j^{th} columns of m. For two matrices m_1 and m_2, the i, j^{th} element of the covariance matrix is the covariance between the i^{th} column of m_1 and the j^{th} column of m_2.

While covariance measures dispersion, correlation measures association. The correlation between two vectors is equivalent to the covariance between the vectors divided by the standard deviations of the vectors. Likewise, the elements of a correlation matrix are equivalent to the elements of the corresponding covariance matrix scaled by the appropriate column standard deviations.

This gives the covariance between *data* and a random vector.

```
In[6]:= Covariance[data, RandomReal[1, Length[data]]]
Out[6]= 0.0258505
```

Here is a random matrix.

```
In[7]:= m = RandomReal[10, {20, 2}]
Out[7]= {{8.01573, 5.3642}, {6.70564, 0.352495}, {2.17328, 7.48353}, {1.33259, 3.27026},
          {9.54907, 8.35172}, {1.56138, 9.4684}, {5.76737, 4.42373}, {8.65789, 6.66041},
          {6.65159, 7.40813}, {3.38061, 6.22431}, {0.269599, 9.76406}, {5.23322, 4.58995},
          {3.3881, 1.66902}, {5.66131, 6.06514}, {7.50919, 8.17705}, {5.92976, 0.803385},
          {9.96, 1.18177}, {2.14364, 5.8279}, {8.13317, 8.79128}, {3.51722, 3.08246}}
```

This is the correlation matrix for the matrix m.

```
In[8]:= Correlation[m]
Out[8]= {{1., -0.132314}, {-0.132314, 1.}}
```

This is the covariance matrix.

```
In[9]:= Covariance[m]
Out[9]= {{8.48155, -1.13411}, {-1.13411, 8.66215}}
```

Scaling the covariance matrix terms by the appropriate standard deviations gives the correlation matrix.

```
In[10]:= With[{sd = StandardDeviation[m]},
          Transpose[Transpose[% / sd] / sd]]
Out[10]= {{1., -0.132314}, {-0.132314, 1.}}
```

`CentralMoment[`*data*`,`*r*`]`	r^{th} central moment $\frac{1}{n}\sum_i (x_i - \overline{x})^r$
`Skewness[`*data*`]`	coefficient of skewness
`Kurtosis[`*data*`]`	kurtosis coefficient
`QuartileSkewness[`*data*`]`	quartile skewness coefficient

Shape statistics.

You can get some information about the shape of a distribution using shape statistics. Skewness describes the amount of asymmetry. Kurtosis measures the concentration of data around the peak and in the tails versus the concentration in the flanks.

`Skewness` is calculated by dividing the third central moment by the cube of the population standard deviation. `Kurtosis` is calculated by dividing the fourth central moment by the square of the population variance of the data, equivalent to `CentralMoment[`*data*`, 2]`. (The population variance is the second central moment, and the population standard deviation is its square root.)

`QuartileSkewness` is calculated from the quartiles of data. It is equivalent to $(q_1 - 2\,q_2 + q_3)/(q_3 - q_1)$, where q_1, q_2, and q_3 are the first, second, and third quartiles respectively.

Here is the second central moment of the data.

```
In[11]:= CentralMoment[data, 2]
Out[11]= 0.825306
```

A negative value for skewness indicates that the distribution underlying the data has a long left-sided tail.

```
In[12]:= Skewness[data]
Out[12]= -1.20108
```

ExpectedValue[*f*, *list*]	expected value of the pure function *f* with respect to the values in *list*
ExpectedValue[*f*[*x*], *list*, *x*]	expected value of the function *f* of *x* with respect to the values of *list*

Expected values.

The expected value of a function f is $\frac{1}{n}\sum_{i=1}^{n} f(x_i)$ for the list of values x_1, x_2, ..., x_n. Many descriptive statistics are expected values. For instance, the mean is the expected value of x, and the r^{th} central moment is the expected value of $(x-\bar{x})^r$ where $\bar{x}$ is the mean of the x_i.

Here is the expected value of the Log of the data.

```
In[13]:= ExpectedValue[Log, data]
Out[13]= 1.73573
```

Discrete Distributions

The functions described here are among the most commonly used discrete statistical distributions. You can compute their densities, means, variances, and other related properties. The distributions themselves are represented in the symbolic form *name*[*param*$_1$, *param*$_2$, ...]. Functions such as Mean, which give properties of statistical distributions, take the symbolic representation of the distribution as an argument. "Continuous Distributions" describes many continuous statistical distributions.

BernoulliDistribution[*p*]	Bernoulli distribution with mean *p*
BetaBinomialDistribution[*α*, *β*, *n*]	binomial distribution where the success probability is a BetaDistribution[*α*, *β*] random variable
BetaNegativeBinomialDistribution[*α*, *β*, *n*]	negative binomial distribution where the success probability is a BetaDistribution[*α*, *β*] random variable
BinomialDistribution[*n*, *p*]	binomial distribution for the number of successes that occur in *n* trials, where the probability of success in a trial is *p*
DiscreteUniformDistribution[{*i*$_{min}$, *i*$_{max}$}]	

	discrete uniform distribution over the integers from i_{min} to i_{max}
`GeometricDistribution[`p`]`	geometric distribution for the number of trials before the first success, where the probability of success in a trial is p
`HypergeometricDistribution[`n`,`n_{succ}`,`n_{tot}`]`	
	hypergeometric distribution for the number of successes out of a sample of size n, from a population of size n_{tot} containing n_{succ} successes
`LogSeriesDistribution[`θ`]`	logarithmic series distribution with parameter θ
`NegativeBinomialDistribution[`n`,`p`]`	
	negative binomial distribution with parameters n and p
`PoissonDistribution[`μ`]`	Poisson distribution with mean μ
`ZipfDistribution[`ρ`]`	Zipf distribution with parameter ρ

Discrete statistical distributions.

Most of the common discrete statistical distributions can be understood by considering a sequence of trials, each with two possible outcomes, for example, success and failure.

The *Bernoulli distribution* `BernoulliDistribution[`p`]` is the probability distribution for a single trial in which success, corresponding to value 1, occurs with probability p, and failure, corresponding to value 0, occurs with probability $1 - p$.

The *binomial distribution* `BinomialDistribution[`n`, `p`]` is the distribution of the number of successes that occur in n independent trials, where the probability of success in each trial is p.

The *negative binomial distribution* `NegativeBinomialDistribution[`n`, `p`]` for positive integer n is the distribution of the number of failures that occur in a sequence of trials before n successes have occurred, where the probability of success in each trial is p. The distribution is defined for any positive n, though the interpretation of n as the number of successes and p as the success probability no longer holds if n is not an integer.

The *beta binomial distribution* `BetaBinomialDistribution[`α`, `β`, `n`]` is a mixture of binomial and beta distributions. A `BetaBinomialDistribution[`α`, `β`, `n`]` random variable follows a `BinomialDistribution[`n`, `p`]` distribution, where the success probability p is itself a random variable following the beta distribution `BetaDistribution[`α`, `β`]`. The *beta negative binomial distribution* `BetaNegativeBinomialDistribution[`α`, `β`, `n`]` is a similar mixture of the beta and negative binomial distributions.

The *geometric distribution* `GeometricDistribution[`p`]` is the distribution of the total number of trials before the first success occurs, where the probability of success in each trial is p.

The *hypergeometric distribution* `HypergeometricDistribution[`n`,` n_{succ}`,` n_{tot}`]` is used in place of the binomial distribution for experiments in which the n trials correspond to sampling without replacement from a population of size n_{tot} with n_{succ} potential successes.

The *discrete uniform distribution* `DiscreteUniformDistribution[{`i_{min}`,` i_{max}`}]` represents an experiment with multiple equally probable outcomes represented by integers i_{min} through i_{max}.

The *Poisson distribution* `PoissonDistribution[`μ`]` describes the number of events that occur in a given time period where μ is the average number of events per period.

The terms in the series expansion of $\log(1-\theta)$ about $\theta = 0$ are proportional to the probabilities of a discrete random variable following the *logarithmic series distribution* `LogSeriesDistribution[`θ`]`. The distribution of the number of items of a product purchased by a buyer in a specified interval is sometimes modeled by this distribution.

The *Zipf distribution* `ZipfDistribution[`ρ`]`, sometimes referred to as the zeta distribution, was first used in linguistics and its use has been extended to model rare events.

`PDF[`*dist*`,`x`]`	probability mass function at x
`CDF[`*dist*`,`x`]`	cumulative distribution function at x
`InverseCDF[`*dist*`,`q`]`	the largest integer x such that `CDF[`*dist*`,` x`]` is at most q
`Quantile[`*dist*`,`q`]`	q^{th} quantile
`Mean[`*dist*`]`	mean
`Variance[`*dist*`]`	variance
`StandardDeviation[`*dist*`]`	standard deviation
`Skewness[`*dist*`]`	coefficient of skewness
`Kurtosis[`*dist*`]`	coefficient of kurtosis
`CharacteristicFunction[`*dist*`,`t`]`	characteristic function $\phi(t)$
`ExpectedValue[`f`,`*dist*`]`	expected value of the pure function f in *dist*
`ExpectedValue[`f`[`x`],`*dist*`,`x`]`	expected value of $f[x]$ for x in *dist*
`Median[`*dist*`]`	median
`Quartiles[`*dist*`]`	list of the $\frac{1}{4}^{\text{th}}$, $\frac{1}{2}^{\text{th}}$, $\frac{3}{4}^{\text{th}}$ quantiles for *dist*
`InterquartileRange[`*dist*`]`	difference between the first and third quartiles
`QuartileDeviation[`*dist*`]`	half the interquartile range

QuartileSkewness[*dist*]	quartile-based skewness measure
RandomInteger[*dist*]	pseudorandom number with specified distribution
RandomInteger[*dist*,*dims*]	pseudorandom array with dimensionality *dims*, and elements from the specified distribution

Functions of statistical distributions.

Distributions are represented in symbolic form. PDF[*dist*, *x*] evaluates the mass function at *x* if *x* is a numerical value, and otherwise leaves the function in symbolic form whenever possible. Similarly, CDF[*dist*, *x*] gives the cumulative distribution and Mean[*dist*] gives the mean of the specified distribution. For a more complete description of the various functions of a statistical distribution, see the description of their continuous analogues in "Continuous Distributions".

Here is a symbolic representation of the binomial distribution for 34 trials, each having probability 0.3 of success.

```
In[1]:= bdist = BinomialDistribution[34, 0.3]
Out[1]= BinomialDistribution[34, 0.3]
```

This is the mean of the distribution.

```
In[2]:= Mean[bdist]
Out[2]= 10.2
```

You can get the expression for the mean by using symbolic variables as arguments.

```
In[3]:= Mean[BinomialDistribution[n, p]]
Out[3]= n p
```

Here is the 50% quantile, which is equal to the median.

```
In[4]:= Quantile[bdist, 0.5]
Out[4]= 10
```

This gives the expected value of x^3 with respect to the binomial distribution.

```
In[5]:= ExpectedValue[x^3, bdist, x]
Out[5]= 1282.55
```

The elements of this matrix are pseudorandom numbers from the binomial distribution.

```
In[6]:= RandomInteger[bdist, {2, 3}]
Out[6]= {{10, 7, 9}, {12, 10, 11}}
```

Continuous Distributions

The functions described here are among the most commonly used continuous statistical distributions. You can compute their densities, means, variances, and other related properties. The distributions themselves are represented in the symbolic form $name[param_1, param_2, \ldots]$. Functions such as `Mean`, which give properties of statistical distributions, take the symbolic representation of the distribution as an argument. "Discrete Distributions" describes many discrete statistical distributions.

`NormalDistribution[`μ`,`σ`]`	normal (Gaussian) distribution with mean μ and standard deviation σ
`HalfNormalDistribution[`θ`]`	half-normal distribution with scale inversely proportional to parameter θ
`LogNormalDistribution[`μ`,`σ`]`	lognormal distribution based on a normal distribution with mean μ and standard deviation σ
`InverseGaussianDistribution[`μ`,`λ`]`	inverse Gaussian distribution with mean μ and scale λ

Distributions related to the normal distribution.

The *lognormal distribution* `LogNormalDistribution[`μ`, `σ`]` is the distribution followed by the exponential of a normally distributed random variable. This distribution arises when many independent random variables are combined in a multiplicative fashion. The *half-normal distribution* `HalfNormalDistribution[`θ`]` is proportional to the distribution `NormalDistribution[0, 1 / (`θ` Sqrt[2 / `π`])]` limited to the domain $[0, \infty)$.

The *inverse Gaussian distribution* `InverseGaussianDistribution[`μ`, `λ`]`, sometimes called the Wald distribution, is the distribution of first passage times in Brownian motion with positive drift.

`ChiSquareDistribution[`ν`]`	χ^2 distribution with ν degrees of freedom
`InverseChiSquareDistribution[`ν`]`	inverse χ^2 distribution with ν degrees of freedom
`FRatioDistribution[`n`,`m`]`	F-ratio distribution with n numerator and m denominator degrees of freedom
`StudentTDistribution[`ν`]`	Student t distribution with ν degrees of freedom

`NoncentralChiSquareDistribution[ν,λ]`	noncentral χ^2 distribution with ν degrees of freedom and noncentrality parameter λ
`NoncentralStudentTDistribution[ν,δ]`	noncentral Student t distribution with ν degrees of freedom and noncentrality parameter δ
`NoncentralFRatioDistribution[n,m,λ]`	noncentral F-ratio distribution with n numerator degrees of freedom and m denominator degrees of freedom and numerator noncentrality parameter λ

Distributions related to normally distributed samples.

If $X_1,\ldots, X_\nu$ are independent normal random variables with unit variance and mean zero, then $\sum_{i=1}^{\nu} X_i^2$ has a χ^2 *distribution* with ν degrees of freedom. If a normal variable is standardized by subtracting its mean and dividing by its standard deviation, then the sum of squares of such quantities follows this distribution. The χ^2 distribution is most typically used when describing the variance of normal samples.

If Y follows a χ^2 *distribution* with ν degrees of freedom, $1/Y$ follows the *inverse* χ^2 *distribution* `InverseChiSquareDistribution[ν]`. A *scaled inverse* χ^2 *distribution* with ν degrees of freedom and scale ξ can be given as `InverseChiSquareDistribution[ν, ξ]`. Inverse χ^2 distributions are commonly used as prior distributions for the variance in Bayesian analysis of normally distributed samples.

A variable that has a *Student* t *distribution* can also be written as a function of normal random variables. Let X and Z be independent random variables, where X is a standard normal distribution and Z is a χ^2 variable with ν degrees of freedom. In this case, $X\Big/\sqrt{Z/\nu}$ has a t distribution with ν degrees of freedom. The Student t distribution is symmetric about the vertical axis, and characterizes the ratio of a normal variable to its standard deviation. Location and scale parameters can be included as μ and σ in `StudentTDistribution[μ, σ, ν]`. When $\nu = 1$, the t distribution is the same as the Cauchy distribution.

The F*-ratio distribution* is the distribution of the ratio of two independent χ^2 variables divided by their respective degrees of freedom. It is commonly used when comparing the variances of two populations in hypothesis testing.

Distributions that are derived from normal distributions with nonzero means are called *noncentral distributions*.

The sum of the squares of ν normally distributed random variables with variance $\sigma^2 = 1$ and nonzero means follows a *noncentral* χ^2 *distribution* `NoncentralChiSquareDistribution[ν, λ]`. The noncentrality parameter λ is the sum of the squares of the means of the random variables in the sum. Note that in various places in the literature, $\lambda/2$ or $\sqrt{\lambda}$ is used as the noncentrality parameter.

The *noncentral Student* t *distribution* `NoncentralStudentTDistribution[ν, δ]` describes the ratio $X \Big/ \sqrt{\chi^2_\nu / \nu}$ where χ^2_ν is a central χ^2 random variable with ν degrees of freedom, and X is an independent normally distributed random variable with variance $\sigma^2 = 1$ and mean δ.

The *noncentral* F*-ratio distribution* `NoncentralFRatioDistribution[n, m, λ]` is the distribution of the ratio of $\frac{1}{n}\chi^2_n(\lambda)$ to $\frac{1}{m}\chi^2_m$, where $\chi^2_n(\lambda)$ is a noncentral χ^2 random variable with noncentrality parameter λ and n_1 degrees of freedom and χ^2_m is a central χ^2 random variable with m degrees of freedom.

`TriangularDistribution[{a,b}]`	symmetric triangular distribution on the interval $\{a, b\}$
`TriangularDistribution[{a,b},c]`	triangular distribution on the interval $\{a, b\}$ with maximum at c
`UniformDistribution[{min,max}]`	uniform distribution on the interval $\{min, max\}$

Piecewise linear distributions.

The *triangular distribution* `TriangularDistribution[{a, b}, c]` is a triangular distribution for $a < X < b$ with maximum probability at c and $a < c < b$. If c is $\frac{a+b}{2}$, `TriangularDistribution[{a, b}, c]` is the symmetric triangular distribution `TriangularDistribution[{a, b}]`.

The *uniform distribution* `UniformDistribution[{min, max}]`, commonly referred to as the rectangular distribution, characterizes a random variable whose value is everywhere equally likely. An example of a uniformly distributed random variable is the location of a point chosen randomly on a line from min to max.

`BetaDistribution[α,β]`	continuous beta distribution with shape parameters α and β
`CauchyDistribution[a,b]`	Cauchy distribution with location parameter a and scale parameter b

ChiDistribution[ν]	χ distribution with ν degrees of freedom
ExponentialDistribution[λ]	exponential distribution with scale inversely proportional to parameter λ
ExtremeValueDistribution[α,β]	extreme maximum value (Fisher-Tippett) distribution with location parameter α and scale parameter β
GammaDistribution[α,β]	gamma distribution with shape parameter α and scale parameter β
GumbelDistribution[α,β]	Gumbel minimum extreme value distribution with location parameter α and scale parameter β
InverseGammaDistribution[α,β]	inverse gamma distribution with shape parameter α and scale parameter β
LaplaceDistribution[μ,β]	Laplace (double exponential) distribution with mean μ and scale parameter β
LevyDistribution[μ,σ]	Lévy distribution with location parameter μ and dispersion parameter σ
LogisticDistribution[μ,β]	logistic distribution with mean μ and scale parameter β
MaxwellDistribution[σ]	Maxwell (Maxwell-Boltzmann) distribution with scale parameter σ
ParetoDistribution[k,α]	Pareto distribution with minimum value parameter k and shape parameter α
RayleighDistribution[σ]	Rayleigh distribution with scale parameter σ
WeibullDistribution[α,β]	Weibull distribution with shape parameter α and scale parameter β

Other continuous statistical distributions.

If X is uniformly distributed on [$-\pi$, π], then the random variable $\tan(X)$ follows a *Cauchy distribution* CauchyDistribution[a, b], with $a = 0$ and $b = 1$.

When $\alpha = n/2$ and $\lambda = 2$, the *gamma distribution* GammaDistribution[α, λ] describes the distribution of a sum of squares of n-unit normal random variables. This form of the gamma distribution is called a χ^2 *distribution* with ν degrees of freedom. When $\alpha = 1$, the gamma distribution takes on the form of the *exponential distribution* ExponentialDistribution[λ], often used in describing the waiting time between events.

If a random variable X follows the *gamma distribution* GammaDistribution[α, β], $1/X$ follows the *inverse gamma distribution* InverseGammaDistribution[α, 1 / β]. If a random variable X follows InverseGammaDistribution[1 / 2, σ / 2], $X + \mu$ follows a *Lévy distribution* LevyDistribution[μ, σ].

When X_1 and X_2 have independent gamma distributions with equal scale parameters, the random variable $\frac{X_1}{X_1+X_2}$ follows the *beta distribution* `BetaDistribution[α, β]`, where α and β are the shape parameters of the gamma variables.

The χ *distribution* `ChiDistribution[ν]` is followed by the square root of a χ^2 random variable. For $n=1$, the χ distribution is identical to `HalfNormalDistribution[θ]` with $\theta=\sqrt{\frac{\pi}{2}}$. For $n=2$, the χ distribution is identical to the *Rayleigh distribution* `RayleighDistribution[σ]` with $\sigma=1$. For $n=3$, the χ distribution is identical to the *Maxwell-Boltzmann distribution* `MaxwellDistribution[σ]` with $\sigma=1$.

The *Laplace distribution* `LaplaceDistribution[μ, β]` is the distribution of the difference of two independent random variables with identical exponential distributions. The *logistic distribution* `LogisticDistribution[μ, β]` is frequently used in place of the normal distribution when a distribution with longer tails is desired.

The *Pareto distribution* `ParetoDistribution[k, α]` may be used to describe income, with k representing the minimum income possible.

The *Weibull distribution* `WeibullDistribution[α, β]` is commonly used in engineering to describe the lifetime of an object. The *extreme value distribution* `ExtremeValueDistribution[α, β]` is the limiting distribution for the largest values in large samples drawn from a variety of distributions, including the normal distribution. The limiting distribution for the smallest values in such samples is the *Gumbel distribution*, `GumbelDistribution[α, β]`. The names "extreme value" and "Gumbel distribution" are sometimes used interchangeably because the distributions of the largest and smallest extreme values are related by a linear change of variable. The extreme value distribution is also sometimes referred to as the log-Weibull distribution because of logarithmic relationships between an extreme value-distributed random variable and a properly shifted and scaled Weibull-distributed random variable.

`PDF[`*dist*`,`*x*`]`	probability density function at *x*
`CDF[`*dist*`,`*x*`]`	cumulative distribution function at *x*
`InverseCDF[`*dist*`,`*q*`]`	the value of *x* such that `CDF[`*dist*`,` *x*`]` equals *q*

`Quantile[dist,q]`	q^{th} quantile
`Mean[dist]`	mean
`Variance[dist]`	variance
`StandardDeviation[dist]`	standard deviation
`Skewness[dist]`	coefficient of skewness
`Kurtosis[dist]`	coefficient of kurtosis
`CharacteristicFunction[dist,t]`	characteristic function $\phi(t)$
`ExpectedValue[f,dist]`	expected value of the pure function f in *dist*
`ExpectedValue[f[x],dist,x]`	expected value of $f[x]$ for x in *dist*
`Median[dist]`	median
`Quartiles[dist]`	list of the $\frac{1}{4}$th, $\frac{1}{2}$th, $\frac{3}{4}$th quantiles for *dist*
`InterquartileRange[dist]`	difference between the first and third quartiles
`QuartileDeviation[dist]`	half the interquartile range
`QuartileSkewness[dist]`	quartile-based skewness measure
`RandomReal[dist]`	pseudorandom number with specified distribution
`RandomReal[dist,dims]`	pseudorandom array with dimensionality *dims*, and elements from the specified distribution

Functions of statistical distributions.

The *cumulative distribution function* (cdf) at x is given by the integral of the *probability density function* (pdf) up to x. The pdf can therefore be obtained by differentiating the cdf (perhaps in a generalized sense). In this package the distributions are represented in symbolic form. `PDF[dist, x]` evaluates the density at x if x is a numerical value, and otherwise leaves the function in symbolic form. Similarly, `CDF[dist, x]` gives the cumulative distribution.

The inverse cdf `InverseCDF[dist, q]` gives the value of x at which `CDF[dist, x]` reaches q. The median is given by `InverseCDF[dist, 1 / 2]`. Quartiles, deciles and percentiles are particular values of the inverse cdf. Quartile skewness is equivalent to $(q_1 - 2\,q_2 + q_3)/(q_3 - q_1)$, where q_1, q_2 and q_3 are the first, second, and third quartiles, respectively. Inverse cdfs are used in constructing confidence intervals for statistical parameters. `InverseCDF[dist, q]` and `Quantile[dist, q]` are equivalent for continuous distributions.

The mean `Mean[dist]` is the expectation of the random variable distributed according to *dist* and is usually denoted by μ. The mean is given by $y \int x\, f(x)\, dx$, where $f(x)$ is the pdf of the distribution. The variance `Variance[dist]` is given by $\int (x-\mu)^2\, f(x)\, dx$. The square root of the variance is called the standard deviation, and is usually denoted by σ.

The `Skewness[dist]` and `Kurtosis[dist]` functions give shape statistics summarizing the asymmetry and the peakedness of a distribution, respectively. Skewness is given by $\frac{1}{\sigma^3}\int(x-\mu)^3 f(x)\,dx$ and kurtosis is given by $\frac{1}{\sigma^4}\int(x-\mu)^4 f(x)\,dx$.

The characteristic function `CharacteristicFunction[dist, t]` is given by $\phi(t)=\int f(x)\exp(i\,t\,x)\,dx$. In the discrete case, $\phi(t)=\sum f(x)\exp(i\,t\,x)$. Each distribution has a unique characteristic function, which is sometimes used instead of the pdf to define a distribution.

The expected value `ExpectedValue[g, dist]` of a function g is given by $\int f(x)\,g(x)\,dx$. In the discrete case, the expected value of g is given by $\sum f(x)\,g(x)$. `ExpectedValue[g[x], dist, x]` is equivalent to `ExpectedValue[g, dist]`.

`RandomReal[dist]` gives pseudorandom numbers from the specified distribution.

This gives a symbolic representation of the gamma distribution with $\alpha=3$ and $\beta=1$.

```
In[1]:= gdist = GammaDistribution[3, 1]
Out[1]= GammaDistribution[3, 1]
```

Here is the cumulative distribution function evaluated at 10.

```
In[2]:= CDF[gdist, 10]
Out[2]= GammaRegularized[3, 0, 10]
```

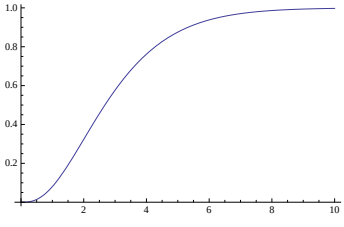
This is the cumulative distribution function. It is given in terms of the built-in function `GammaRegularized`.

```
In[3]:= cdfunction = CDF[gdist, x]
Out[3]= GammaRegularized[3, 0, x]
```

Here is a plot of the cumulative distribution function.

```
In[4]:= Plot[cdfunction, {x, 0, 10}]
```

Out[4]=



This is a pseudorandom array with elements distributed according to the gamma distribution.

```
In[5]:= RandomReal[gdist, 5]
Out[5]= {1.46446, 8.56359, 2.70647, 1.97748, 2.97108}
```

Partitioning Data into Clusters

Cluster analysis is an unsupervised learning technique used for classification of data. Data elements are partitioned into groups called clusters that represent proximate collections of data elements based on a distance or dissimilarity function. Identical element pairs have zero distance or dissimilarity, and all others have positive distance or dissimilarity.

`FindClusters[`*data*`]`	partition *data* into lists of similar elements
`FindClusters[`*data*`,`*n*`]`	partition *data* into exactly *n* lists of similar elements

General clustering function.

The data argument of `FindClusters` can be a list of data elements or rules indexing elements and labels.

$\{e_1, e_2, \ldots\}$	data specified as a list of data elements e_i
$\{e_1 \to v_1, e_2 \to v_2, \ldots\}$	data specified as a list of rules between data elements e_i and labels v_i
$\{e_1, e_2, \ldots\} \to \{v_1, v_2, \ldots\}$	data specified as a rule mapping data elements e_i to labels v_i

Ways of specifying data in `FindClusters`.

The data elements e_i can be numeric lists, matrices, tensors, lists of `True` and `False` elements, or lists of strings. All data elements e_i must have the same dimensions.

Here is a list of numbers.

```
In[1]:= data = {1.2, 9.1, 2.3, 15.4, 71.8};
```

`FindClusters` clusters the numbers based on their proximity.

```
In[2]:= FindClusters[data]
Out[2]= {{1.2, 2.3}, {9.1, 15.4}, {71.8}}
```

The rule-based data syntax allows for clustering data elements and returning labels for those elements.

Here two-dimensional points are clustered and labeled with their positions in the data list.

```
In[3]:= data1 = {{1, 2}, {3, 7}, {0, 3}, {3, 1}};
        FindClusters[data1 -> Range[Length[data1]]]
Out[3]= {{1, 3, 4}, {2}}
```

The rule-based data syntax can also be used to cluster data based on parts of each data entry. For instance, you might want to cluster data in a data table while ignoring particular columns in the table.

Here is a list of data entries.

```
In[4]:= datarecords = {{"Joe", "Smith", 158, 64.4}, {"Mary", "Davis", 137, 64.4},
          {"Bob", "Lewis", 141, 62.8}, {"John", "Thompson", 235, 71.1},
          {"Lewis", "Black", 225, 71.4}, {"Sally", "Jones", 168, 62.},
          {"Tom", "Smith", 243, 70.9}, {"Jane", "Doe", 225, 71.4}};
```

This clusters the data while ignoring the first two elements in each data entry.

```
In[5]:= FindClusters[Drop[datarecords, None, {1, 2}] → datarecords]
Out[5]= {{{Joe, Smith, 158, 64.4}, {Sally, Jones, 168, 62.}},
          {{Mary, Davis, 137, 64.4}, {Bob, Lewis, 141, 62.8}}, {{John, Thompson, 235, 71.1},
           {Lewis, Black, 225, 71.4}, {Tom, Smith, 243, 70.9}, {Jane, Doe, 225, 71.4}}}
```

In principle, it is possible to cluster points given in an arbitrary number of dimensions. However, it is difficult at best to visualize the clusters above two or three dimensions. To compare optional methods in this documentation, an easily visualizable set of two-dimensional data will be used.

The following commands define a set of 300 two-dimensional data points chosen to group into four somewhat nebulous clusters.

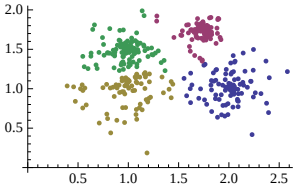
```
In[6]:= GaussianRandomData[n_Integer, p_, sigma_] := Table[p + {Re[#], Im[#]} &[
            RandomReal[NormalDistribution[0, sigma]] e^(i RandomReal[{0, 2 π}])], {n}];
        datapairs = BlockRandom[
          SeedRandom[1234];
          Join[GaussianRandomData[100, {2, 1}, .3],
           GaussianRandomData[100, {1, 1.5}, .2],
           GaussianRandomData[100, {1, 1.1}, .4],
           GaussianRandomData[100, {1.75, 1.75}, 0.1]]];
```

This clusters the data based on the proximity of points.

```
In[7]:= cl = FindClusters[datapairs];
```

Here is a plot of the clusters.

```
In[8]:= ListPlot[cl]
```

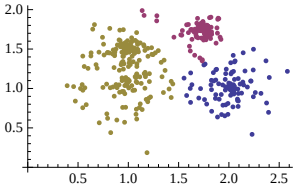


With the default settings, `FindClusters` has found the four clusters of points.

You can also direct `FindClusters` to find a specific number of clusters.

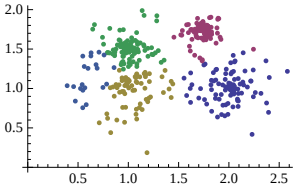
This shows the effect of choosing 3 clusters.

```
In[9]:= ListPlot[FindClusters[datapairs, 3]]
```



This shows the effect of choosing 5 clusters.

```
In[10]:= ListPlot[FindClusters[datapairs, 5]]
```



option name	*default value*	
`DistanceFunction`	`Automatic`	the distance or dissimilarity measure to use
`Method`	`Automatic`	the clustering method to use

Options for `FindClusters`.

Randomness is used in clustering in two different ways. Some of the methods use a random assignment of some points to a specific number of clusters as a starting point. Randomness may also be used to help determine what seems to be the best number of clusters to use. Changing the random seed for generating the randomness by using `FindClusters[{`e_1`, `e_2`, ...}, Method → {Automatic, "RandomSeed" -> `s`}]` may lead to different results for some cases.

In principle, clustering techniques can be applied to any set of data. All that is needed is a measure of how far apart each element in the set is from other elements, that is, a function giving the distance between elements.

`FindClusters[{`e_1`, `e_2`, ...}, DistanceFunction -> `f`]` treats pairs of elements as being less similar when their distances $f[e_i, e_j]$ are larger. The function f can be any appropriate distance or dissimilarity function. A dissimilarity function *f* satisfies the following:

$$f(e_i, e_i) = 0$$
$$f(e_i, e_j) \geq 0$$
$$f(e_i, e_j) = f(e_j, e_i)$$

If the e_i are vectors of numbers, `FindClusters` by default uses a squared Euclidean distance. If the e_i are lists of Boolean `True` and `False` (or 0 and 1) elements, `FindClusters` by default uses a dissimilarity based on the normalized fraction of elements that disagree. If the e_i are strings, `FindClusters` by default uses a distance function based on the number of point changes needed to get from one string to another.

`EuclideanDistance[`u`,`v`]`	the Euclidean norm $\sqrt{\sum(u-v)^2}$
`SquaredEuclideanDistance[`u`,`v`]`	squared Euclidean norm $\sum(u-v)^2$
`ManhattanDistance[`u`,`v`]`	the Manhattan distance $\sum\|u-v\|$
`ChessboardDistance[`u`,`v`]`	the chessboard or Chebyshev distance $\max(\|u-v\|)$
`CanberraDistance[`u`,`v`]`	the Canberra distance $\sum\|u-v\|/(\|u\|+\|v\|)$
`CosineDistance[`u`,`v`]`	the cosine distance $1 - u.v/(\|\|u\|\| \|\|v\|\|)$

`CorrelationDistance[`u`,`v`]`	the correlation distance
	`1 - (`u` - Mean[`u`]) . (`v` - Mean[`v`]) /`
	`(||`u` - Mean[`u`]|| ||`v` - Mean[`v`]||)`

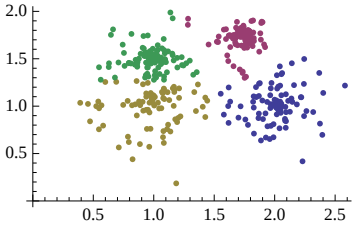
`BrayCurtisDistance[`u`,`v`]`	the Bray-Curtis distance $\sum\|u-v\|/\sum\|u+v\|$

Distance functions for numerical data.

This shows the clusters in *datapairs* found using a Manhattan distance.

```
In[11]:= ListPlot[FindClusters[datapairs, DistanceFunction → ManhattanDistance]]
```

Out[11]=



Dissimilarities for Boolean vectors are typically calculated by comparing the elements of two Boolean vectors u and v pairwise. It is convenient to summarize each dissimilarity function in terms of n_{ij}, where n_{ij} is the number of corresponding pairs of elements in u and v, respectively, equal to i and j. The number n_{ij} counts the pairs $\{i, j\}$ in $\{u_1, v_1\}, \{u_2, v_2\} \ldots$, with i and j being either 0 or 1. If the Boolean values are `True` and `False`, `True` is equivalent to 1 and `False` is equivalent to 0.

`MatchingDissimilarity[u,v]`	simple matching $(n_{10} + n_{01})$ / `Length[u]`
`JaccardDissimilarity[u,v]`	the Jaccard dissimilarity $(n_{10} + n_{01})/(n_{11} + n_{10} + n_{01})$
`RussellRaoDissimilarity[u,v]`	the Russell-Rao dissimilarity $(n_{10} + n_{01} + n_{00})$ / `Length[u]`
`SokalSneathDissimilarity[u,v]`	the Sokal-Sneath dissimilarity $2(n_{10} + n_{01})/(n_{11} + 2(n_{10} + n_{01}))$
`RogersTanimotoDissimilarity[u,v]`	the Rogers-Tanimoto dissimilarity $2(n_{10} + n_{01})/(n_{11} + 2(n_{10} + n_{01}) + n_{00})$
`DiceDissimilarity[u,v]`	the dice dissimilarity $(n_{10} + n_{01})/(2n_{11} + n_{10} + n_{01})$
`YuleDissimilarity[u,v]`	the Yule dissimilarity $2n_{10}n_{01}/(n_{11}n_{00} + n_{10}n_{01})$

Dissimilarity functions for Boolean data.

Here is some Boolean data.

```
In[12]:= bdata = {{False, False, False, False, False, True, False, False, True, True},
    {True, False, False, False, False, False, False, False, False, True},
    {True, False, False, True, False, False, True, False, True, True},
    {True, True, False, False, True, False, False, False, True, True},
    {True, True, False, False, True, True, True, True, True, True}};
```

These are the clusters found using the default dissimilarity for Boolean data.

```
In[13]:= FindClusters[bdata]
```

```
Out[13]= {{{False, False, False, False, False, True, False, False, True, True}},
  {{True, False, False, False, False, False, False, False, False, True},
   {True, False, False, True, False, False, True, False, True, True},
   {True, True, False, False, True, False, False, False, True, True},
   {True, True, False, False, True, True, True, True, True, True}}}
```

EditDistance[u,v]	the number of edits to transform u into string v
DamerauLevenshteinDistance[u,v]	Damerau-Levenshtein distance between u and v
HammingDistance[u,v]	the number of elements whose values disagree in u and v

Dissimilarity functions for string data.

The edit distance is determined by counting the number of deletions, insertions, and substitutions required to transform one string into another while preserving the ordering of characters. In contrast, the Damerau-Levenshtein distance counts the number of deletions, insertions, substitutions, and transpositions, while the Hamming distance counts only the number of substitutions.

Here is some string data.

```
In[14]:= sdata = {"The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"};
```

This clusters the string data using the edit distance.

```
In[15]:= FindClusters[sdata]
Out[15]= {{The, fox, over, the, lazy, dog}, {quick, brown, jumps}}
```

The Method option can be used to specify different methods of clustering.

"Agglomerate"	find clustering hierarchically
"Optimize"	find clustering by local optimization

Explicit settings for the Method option.

The methods "Agglomerate" and "Optimize" determine how to cluster the data for a particular number of clusters k. "Agglomerate" uses an agglomerative hierarchical method starting with each member of the set in a cluster of its own and fusing nearest clusters until there are k remaining. "Optimize" starts by building a set of k representative objects and clustering around those, iterating until a (locally) optimal clustering is found. The default "Optimize" method is based on partitioning around medoids.

Additional Method suboptions are available to allow for more control over the clustering. Available suboptions depend on the Method chosen.

"SignificanceTest"	test for identifying the best number of clusters

Suboption for all methods.

For a given set of data and distance function, the choice of the best number of clusters k may be unclear. With `Method -> {`*methodname*`, "SignificanceTest" -> "`*stest*`"}`, "*stest*" is used to determine statistically significant clusters to help choose an appropriate number. Possible values of "*stest*" are `"Silhouette"` and `"Gap"`. The `"Silhouette"` test uses the silhouette statistic to test how well the data is clustered. The `"Gap"` test uses the gap statistic to determine how well the data is clustered.

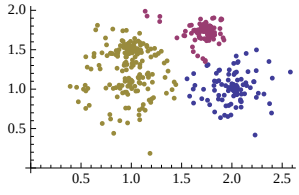
The `"Silhouette"` test subdivides the data into successively more clusters looking for the first minimum of the silhouette statistic.

The `"Gap"` test compares the dispersion of clusters generated from the data to that derived from a sample of null hypothesis sets. The null hypothesis sets are uniformly randomly distributed data in the box defined by the principal components of the input data. The `"Gap"` method takes two suboptions: `"NullSets"` and `"Tolerance"`. The suboption `"NullSets"` sets the number of null hypothesis sets to compare with the input data. The option `"Tolerance"` sets the sensitivity. Typically larger values of `"Tolerance"` will favor fewer clusters being chosen. The default settings are `"NullSets" -> 5` and `"Tolerance" -> 1`.

This shows the result of clustering *datapairs* using the `"Silhouette"` test.

```
In[16]:= ListPlot[FindClusters[datapairs,
           Method → {Automatic, "SignificanceTest" -> "Silhouette"}]]
```

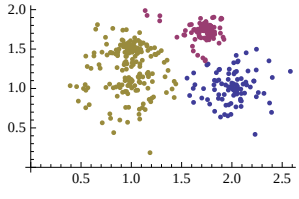
Out[16]=



Here are the clusters found using the `"Gap"` test with the tolerance parameter set to 3. The larger value leads to fewer clusters being selected.

```
In[17]:= ListPlot[FindClusters[datapairs,
           Method → {Automatic, "SignificanceTest" → {"Gap", "Tolerance" → 3}}]]
```

Out[17]=



Note that the clusters found in these two examples are identical. The only difference is how the number of clusters is chosen.

"Linkage"	the clustering linkage to use

Suboption for the "Agglomerate" method.

With Method -> {"Agglomerate", "Linkage" -> f}, the specified linkage function f is used for agglomerative clustering.

"Single"	smallest intercluster dissimilarity
"Average"	average intercluster dissimilarity
"Complete"	largest intercluster dissimilarity
"WeightedAverage"	weighted average intercluster dissimilarity
"Centroid"	distance from cluster centroids
"Median"	distance from cluster medians
"Ward"	Ward's minimum variance dissimilarity
f	a pure function

Possible values for the "Linkage" suboption.

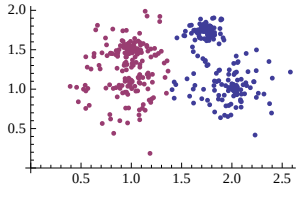
Linkage methods determine this intercluster dissimilarity, or fusion level, given the dissimilarities between member elements.

With Linkage -> f, f is a pure function that defines the linkage algorithm. Distances or dissimilarities between clusters are determined recursively using information about the distances or dissimilarities between unmerged clusters to determine the distances or dissimilarities for the newly merged cluster. The function f defines a distance from a cluster k to the new cluster formed by fusing clusters i and j. The arguments supplied to f are d_{ik}, d_{jk}, d_{ij}, n_i, n_j, and n_k, where d is the distance between clusters and n is the number of elements in a cluster.

These are the clusters found using complete linkage hierarchical clustering.

In[18]:=
```
ListPlot[FindClusters[datapairs, Method → {"Agglomerate", "Linkage" → "Complete"}]]
```

Out[18]=



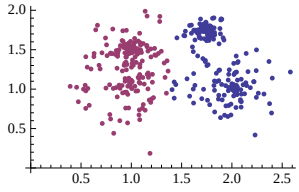
`"Iterations"`	the maximum number of iterations to use

Suboption for the `"Optimize"` method.

Here are the clusters determined from a single iteration of the `"Optimize"` method.

```
In[19]:= ListPlot[FindClusters[datapairs, Method → {"Optimize", "Iterations" → 1}]]
```

Out[19]=



Using Nearest

`Nearest` is used to find elements in a list that are closest to a given data point.

`Nearest[{`$elem_1$`,`$elem_2$`,…},`x`]`	give the list of $elem_i$ to which x is nearest
`Nearest[{`$elem_1$`->`v_1`,`$elem_2$`->`v_2`,…},`x`]`	give the v_i corresponding to the $elem_i$ to which x is nearest
`Nearest[{`$elem_1$`,`$elem_2$`,…}->{`v_1`,`v_2`,…},`x`]`	give the same result
`Nearest[{`$elem_1$`,`$elem_2$`,…}->Automatic,`x`]`	take the v_i to be the integers 1, 2, 3, …
`Nearest[`$data$`,`x`,`n`]`	give the n nearest elements to x
`Nearest[`$data$`,`x`,{`n`,`r`}]`	give up to the n nearest elements to x within a radius r
`Nearest[`$data$`]`	generate a `NearestFunction[…]` which can be applied repeatedly to different x

`Nearest` function.

`Nearest` works with numeric lists, tensors, or a list of strings.

This finds the elements nearest to 4.5.

```
In[1]:= Nearest[{1, 2, 3, 4, 5, 6, 7, 8}, 4.5]
Out[1]= {4, 5}
```

This finds 3 elements nearest to 4.5.

```
In[2]:= Nearest[{1, 2, 3, 4, 5, 6, 7, 8}, 4.5, 3]
Out[2]= {4, 5, 3}
```

This finds all elements nearest to 4.5 within a radius of 2.

```
In[3]:= Nearest[{1, 2, 3, 4, 5, 6, 7, 8}, 4.5, {Infinity, 2}]
Out[3]= {4, 5, 3, 6}
```

This finds the points nearest to `{1, 2}` in 2D.

```
In[4]:= Nearest[{{1, 1}, {2, 2}, {3, 3}}, {1, 2}]
Out[4]= {{1, 1}, {2, 2}}
```

This finds the nearest string to "cat".

```
In[5]:= Nearest[{"bat", "sad", "cake"}, "cat"]
Out[5]= {bat}
```

The rule-based data syntax lets you use nearest elements to return their labels.

Here two-dimensional points are labeled.

```
In[6]:= Nearest[{{1, 1} → a, {2, 2} → b, {3, 3} → c}, {1, 2}]
Out[6]= {a, b}
```

```
In[7]:= Nearest[{{1, 1}, {2, 2}, {3, 3}} → {a, b, c}, {1, 2}]
Out[7]= {a, b}
```

This labels the elements using successive integers.

```
In[8]:= Nearest[{{1, 1}, {2, 2}, {3, 3}, {4, 5}, {7, 7}} → Automatic, {4, 4}]
Out[8]= {4}
```

If `Nearest` is to be applied repeatedly to the same numerical data, you can get significant performance gains by first generating a `NearestFunction`.

This generates a set of 10,000 points in 2D and a `NearestFunction`.

```
In[9]:= pts = RandomReal[1, {10 000, 2}];
        nf = Nearest[pts]
Out[9]= NearestFunction[{10 000, 2}, <>]
```

This finds points in the set that are closest to the 10 target points.

```
In[10]:= target = RandomReal[1, {10, 2}];
         res = Map[nf, target]; // Timing
Out[10]= {4.85723×10^-16, Null}
```

It takes much longer if NearestFunction is not used.

```
In[11]:= res2 = Map[Nearest[pts, #] &, target]; // Timing
Out[11]= {0.504032, Null}

In[12]:= res == res2
Out[12]= True
```

option name	*default value*	
DistanceFunction	Automatic	the distance metric to use

Option for Nearest.

For numerical data, by default Nearest uses the EuclideanDistance. For strings, EditDistance is used.

Manipulating Numerical Data

When you have numerical data, it is often convenient to find a simple formula that approximates it. For example, you can try to "fit" a line or curve through the points in your data.

Fit[{y_1, y_2, ...}, {f_1, f_2, ...}, x]	fit the values y_n to a linear combination of functions f_i
Fit[{{x_1, y_1}, {x_2, y_2}, ...}, {f_1, f_2, ...}, x]	fit the points (x_n, y_n) to a linear combination of the f_i

Fitting curves to linear combinations of functions.

This generates a table of the numerical values of the exponential function. Table is discussed in "Making Tables of Values".

```
In[1]:= data = Table[Exp[x / 5.], {x, 7}]
Out[1]= {1.2214, 1.49182, 1.82212, 2.22554, 2.71828, 3.32012, 4.0552}
```

This finds a least-squares fit to `data` of the form $c_1 + c_2\, x + c_3\, x^2$. The elements of `data` are assumed to correspond to values 1, 2, … of x.

```
In[2]:= Fit[data, {1, x, x^2}, x]
Out[2]= 1.09428 + 0.0986337 x + 0.0459482 x^2
```

This finds a fit of the form $c_1 + c_2\, x + c_3\, x^3 + c_4\, x^5$.

```
In[3]:= Fit[data, {1, x, x^3, x^5}, x]
Out[3]= 0.96806 + 0.246829 x + 0.00428281 x^3 - 6.57948×10^-6 x^5
```

This gives a table of x, y pairs.

```
In[4]:= data = Table[{x, Exp[Sin[x]]}, {x, 0., 1., 0.2}]
Out[4]= {{0., 1.}, {0.2, 1.21978}, {0.4, 1.47612}, {0.6, 1.75882}, {0.8, 2.04901}, {1., 2.31978}}
```

This finds a fit to the new data, of the form $c_1 + c_2 \sin(x) + c_3 \sin(2\,x)$.

```
In[5]:= Fit[%, {1, Sin[x], Sin[2 x]}, x]
Out[5]= 0.989559 + 2.04199 Sin[x] - 0.418176 Sin[2 x]
```

`FindFit[`*data*`,`*form*`,{`p_1`,`p_2`,…},`*x*`]`	find a fit to *form* with parameters p_i

Fitting data to general forms.

This finds the best parameters for a linear fit.

```
In[6]:= FindFit[data, a + b x + c x^2, {a, b, c}, x]
Out[6]= {a → 0.991251, b → 1.16421, c → 0.174256}
```

This does a nonlinear fit.

```
In[7]:= FindFit[data, a + b^(c + d x), {a, b, c, d}, x]
Out[7]= {a → -3.65199, b → 1.65838, c → 3.03496, d → 0.50107}
```

One common way of picking out "signals" in numerical data is to find the *Fourier transform*, or frequency spectrum, of the data.

`Fourier[`*data*`]`	numerical Fourier transform
`InverseFourier[`*data*`]`	inverse Fourier transform

Fourier transforms.

Here is a simple square pulse.

```
In[8]:= data = {1, 1, 1, 1, -1, -1, -1, -1}
Out[8]= {1, 1, 1, 1, -1, -1, -1, -1}
```

This takes the Fourier transform of the pulse.

```
In[9]:= Fourier[data]
Out[9]= {0. + 0. i, 0.707107 + 1.70711 i, 0. + 0. i, 0.707107 + 0.292893 i,
 0. + 0. i, 0.707107 - 0.292893 i, 0. + 0. i, 0.707107 - 1.70711 i}
```

Note that the `Fourier` function in *Mathematica* is defined with the sign convention typically used in the physical sciences—opposite to the one often used in electrical engineering. "Fourier Transforms" gives more details.

Curve Fitting

There are many situations where one wants to find a formula that best fits a given set of data. One way to do this in *Mathematica* is to use `Fit`.

`Fit[{`f_1`,`f_2`,...},{`fun_1`,`fun_2`,...},`x`]`	find a linear combination of the fun_i that best fits the values f_i

Basic linear fitting.

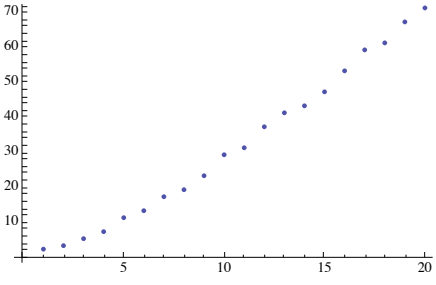
Here is a table of the first 20 primes.

```
In[1]:= fp = Table[Prime[x], {x, 20}]
Out[1]= {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71}
```

Here is a plot of this "data".

```
In[2]:= gp = ListPlot[fp]
```

Out[2]=



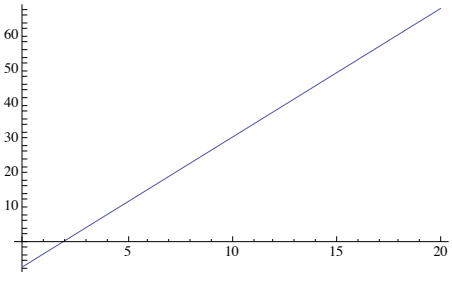
This gives a linear fit to the list of primes. The result is the best linear combination of the functions 1 and x.

```
In[3]:= Fit[fp, {1, x}, x]
Out[3]= -7.67368 + 3.77368 x
```

Here is a plot of the fit.

```
In[4]:= Plot[%, {x, 0, 20}]
```

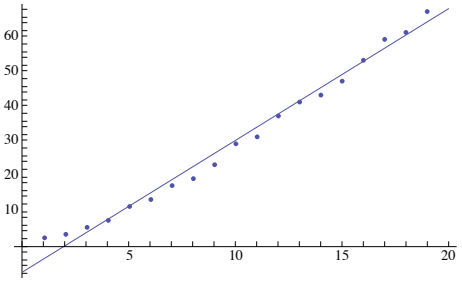
Out[4]=



Here is the fit superimposed on the original data.

```
In[5]:= Show[%, gp]
```

Out[5]=



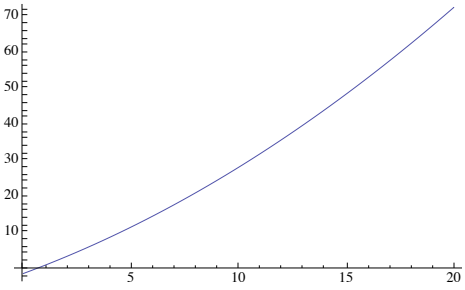
This gives a quadratic fit to the data.

```
In[6]:= Fit[fp, {1, x, x^2}, x]
Out[6]= -1.92368 + 2.2055 x + 0.0746753 x^2
```

Here is a plot of the quadratic fit.

```
In[7]:= Plot[%, {x, 0, 20}]
```

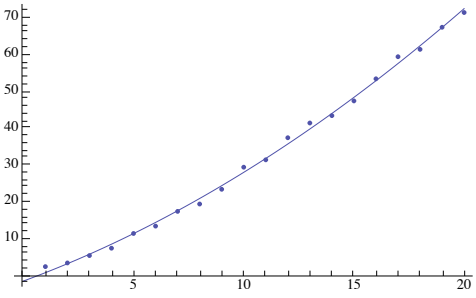
Out[7]=



This shows the fit superimposed on the original data. The quadratic fit is better than the linear one.

```
In[8]:= Show[%, gp]
```

Out[8]=



$\{f_1, f_2, \ldots\}$	data points obtained when a single coordinate takes on values $1, 2, \ldots$
$\{\{x_1, f_1\}, \{x_2, f_2\}, \ldots\}$	data points obtained when a single coordinate takes on values $x_1, x_2, \ldots$
$\{\{x_1, y_1, \ldots, f_1\}, \{x_2, y_2, \ldots, f_2\}, \ldots\}$	data points obtained with values $x_i, y_i, \ldots$ of a sequence of coordinates

Ways of specifying data.

If you give data in the form $\{f_1, f_2, \ldots\}$ then `Fit` will assume that the successive f_i correspond to values of a function at successive integer points $\{1, 2, \ldots\}$. But you can also give `Fit` data that corresponds to the values of a function at arbitrary points, in one or more dimensions.

`Fit[`*data*`, {`*fun*$_1$`, `*fun*$_2$`, ...}, {`*x*`, `*y*`, ...}]`	fit to a function of several variables

Multivariate fitting.

This gives a table of the values of x, y and $1 + 5\,x - x\,y$. You need to use `Flatten` to get it in the right form for `Fit`.

```
In[9]:= Flatten[Table[{x, y, 1 + 5 x - x y}, {x, 0, 1, 0.4}, {y, 0, 1, 0.4}], 1]
Out[9]= {{0., 0., 1.}, {0., 0.4, 1.}, {0., 0.8, 1.}, {0.4, 0., 3.}, {0.4, 0.4, 2.84},
  {0.4, 0.8, 2.68}, {0.8, 0., 5.}, {0.8, 0.4, 4.68}, {0.8, 0.8, 4.36}}
```

This produces a fit to a function of two variables.

```
In[10]:= Fit[%, {1, x, y, x y}, {x, y}]
Out[10]= 1. + 5. x + 5.21108×10^-15 y - 1. x y
```

`Fit` takes a list of functions, and uses a definite and efficient procedure to find what linear combination of these functions gives the best least-squares fit to your data. Sometimes, however, you may want to find a *nonlinear fit* that does not just consist of a linear combination of specified functions. You can do this using `FindFit`, which takes a function of any form, and then searches for values of parameters that yield the best fit to your data.

`FindFit[`*data*`,`*form*`, {`par_1`,`par_2`,…}`,*x*`]`	search for values of the par_i that make *form* best fit *data*
`FindFit[`*data*`,`*form*`,`*pars*`,{`*x*`,`*y*`,…}]`	fit multivariate data

Searching for general fits to data.

This fits the list of primes to a simple linear combination of terms.

```
In[11]:= FindFit[fp, a + b x + c Exp[x], {a, b, c}, x]
Out[11]= {a → -6.78932, b → 3.64309, c → 1.26883×10^-8}
```

The result is the same as from `Fit`.

```
In[12]:= Fit[fp, {1, x, Exp[x]}, x]
Out[12]= -6.78932 + 1.26883×10^-8 ℯ^x + 3.64309 x
```

This fits to a nonlinear form, which cannot be handled by `Fit`.

```
In[13]:= FindFit[fp, a x Log[b + c x], {a, b, c}, x]
Out[13]= {a → 1.42076, b → 1.65558, c → 0.534645}
```

By default, both `Fit` and `FindFit` produce *least-squares* fits, which are defined to minimize the quantity $\chi^2 = \sum_i |r_i|^2$, where the r_i are residuals giving the difference between each original data

point and its fitted value. One can, however, also consider fits based on other norms. If you set the option `NormFunction -> u`, then `FindFit` will attempt to find the fit that minimizes the quantity `u[r]`, where `r` is the list of residuals. The default is `NormFunction -> Norm`, corresponding to a least-squares fit.

This uses the ∞-norm, which minimizes the maximum distance between the fit and the data. The result is slightly different from least-squares.

```
In[14]:= FindFit[fp, a x Log[b + c x], {a, b, c}, x, NormFunction -> (Norm[#, Infinity] &)]
Out[14]= {a → 1.15077, b → 1.0023, c → 1.04686}
```

`FindFit` works by searching for values of parameters that yield the best fit. Sometimes you may have to tell it where to start in doing this search. You can do this by giving parameters in the form $\{\{a, a_0\}, \{b, b_0\}, \ldots\}$. `FindFit` also has various options that you can set to control how it does its search.

`FindFit[`*data*`,` `{`*form*`,`*cons*`},`*pars*`,`*vars*`]`	finds a best fit subject to the parameter constraints *cons*

Searching for general fits to data.

This gives a best fit subject to constraints on the parameters.

```
In[15]:= FindFit[fp, {a x Log[b + c x], {0 <= a <= 1, 0 <= b ≤ 1, c ≥ 1}}, {a, b, c}, x]
Out[15]= {a → 1., b → 1.34569×10^-9, c → 1.69145}
```

option name	*default value*	
`NormFunction`	`Norm`	the norm to use
`AccuracyGoal`	`Automatic`	number of digits of accuracy to try to get
`PrecisionGoal`	`Automatic`	number of digits of precision to try to get
`WorkingPrecision`	`Automatic`	precision to use in internal computations
`MaxIterations`	`Automatic`	maximum number of iterations to use
`StepMonitor`	`None`	expression to evaluate whenever a step is taken
`EvaluationMonitor`	`None`	expression to evaluate whenever *form* is evaluated
`Method`	`Automatic`	method to use

Options for `FindFit`.

Statistical Model Analysis

When fitting models of data, it is often useful to analyze how well the model fits the data and how well the fitting meets assumptions of the fitting. For a number of common statistical models, this is accomplished in *Mathematica* by way of fitting functions that construct `FittedModel` objects.

`FittedModel`	represents a symbolic fitted model

Object for fitted model information.

`FittedModel` objects can be evaluated at a point or queried for results and diagnostic information. Diagnostics vary somewhat across model types. Available model fitting functions fit linear, generalized linear, and nonlinear models.

`LinearModelFit`	constructs a linear model
`GeneralizedLinearModelFit`	constructs a generalized linear model
`LogitModelFit`	constructs a binomial logistic regression model
`ProbitModelFit`	constructs a binomial probit regression model
`NonlinearModelFit`	constructs a nonlinear least-squares model

Functions that generate `FittedModel` objects.

This fits a linear model assuming x values 1, 2, ...

```
In[1]:= lm = LinearModelFit[{1.5, 3.4, 7.1, 8.3, 10.4}, x, x]
Out[1]= FittedModel[ -0.67 + 2.27 x ]
```

Here is the functional form of the fitted model.

```
In[2]:= Normal[lm]
Out[2]= -0.67 + 2.27 x
```

This evaluates the model for $x = 2.5$.

```
In[3]:= lm[2.5]
Out[3]= 5.005
```

Here is a shortened list of available results for the linear fitted model.

```
In[4]:= lm["Properties"] // Short
Out[4]//Short= {AdjustedRSquared, AIC, <<58>>, StudentizedResiduals, VarianceInflationFactors}
```

The major difference between model fitting functions such as `LinearModelFit` and functions such as `Fit` and `FindFit` is the ability to easily obtain diagnostic information from the `FittedModel` objects. The results are accessible without refitting the model.

This gives the residuals for the fitting.

```
In[5]:= lm["FitResiduals"]
Out[5]= {-0.1, -0.47, 0.96, -0.11, -0.28}
```

Here multiple results are obtained at once.

```
In[6]:= lm[{"BestFitParameters", "ANOVATable"}]
```

Out[6]=

{{-0.67, 2.27},

	DF	SS	MS	F Statistic	P-Value
x	1	51.529	51.529	124.366	0.00154521
Error	3	1.243	0.414333		
Total	4	52.772			

}

Fitting options relevant to property computations can be passed to `FittedModel` objects to override defaults.

This gives default 95% confidence intervals.

```
In[7]:= lm["ParameterConfidenceIntervals"]
Out[7]= {{-2.81849, 1.47849}, {1.62221, 2.91779}}
```

Here 90% intervals are obtained.

```
In[8]:= lm["ParameterConfidenceIntervals", ConfidenceLevel → .9]
Out[8]= {{-2.25877, 0.918767}, {1.79097, 2.74903}}
```

Typical data for these model fitting functions takes the same form as data in other fitting functions such as `Fit` and `FindFit`.

$\{y_1, y_2, \ldots\}$	data points with a single predictor variable taking values 1, 2, …
$\{\{x_{11}, x_{12}, \ldots, y_1\}, \{x_{21}, x_{22}, \ldots, y_2\}, \ldots\}$	data points with explicit coordinates

Data specifications.

Linear Models

Linear models with assumed independent normally distributed errors are among the most common models for data. Models of this type can be fitted using the `LinearModelFit` function.

`LinearModelFit[{y1,y2,…},{f1,f2,…},x]`	obtain a linear model with basis functions f_i and a single predictor variable x
`LinearModelFit[{{x11,x12,…,y1},{x21,x22,…,y2}}, {f1,f2,…},{x1,x2,…}]`	obtain a linear model of multiple predictor variables x_i
`LinearModelFit[{m,v}]`	obtain a linear model based on a design matrix m and response vector v

Linear model fitting.

Linear models have the form $\hat{y} = \beta_0 + \beta_1 f_1 + \beta_2 f_2 + \cdots$ where $\hat{y}$ is the fitted or predicted value, the β_i are parameters to be fitted, and the f_i are functions of the predictor variables x_i. The models are linear in the parameters β_i. The f_i can be any functions of the predictor variables. Quite often the f_i are simply the predictor variables x_i.

This fits a linear model to the first 20 primes.

```
In[9]:= lm = LinearModelFit[Array[Prime, 20], x, x]
Out[9]= FittedModel[ -7.67368 + 3.77368 x ]
```

Options for model specification and for model analysis are available.

option name	*default value*	
`ConfidenceLevel`	`95/100`	confidence level to use for parameters and predictions
`IncludeConstantBasis`	`True`	whether to include a constant basis function
`LinearOffsetFunction`	`None`	known offset in the linear predictor
`NominalVariables`	`None`	variables considered as nominal or categorical
`VarianceEstimatorFunction`	`Automatic`	function for estimating the error variance
`Weights`	`Automatic`	weights for data elements
`WorkingPrecision`	`Automatic`	precision used in internal computations

Options for `LinearModelFit`.

The `Weights` option specifies weight values for weighted linear regression. The `NominalVariables` option specifies which predictor variables should be treated as nominal or categorical. With `NominalVariables -> All`, the model is an analysis of variance (ANOVA) model. With `NominalVariables -> {`x_1`, ..., `x_{i-1}`, `x_{i+1}`, ..., `x_n`}` the model is an analysis of covariance (ANCOVA) model with all but the i^{th} predictor treated as nominal. Nominal variables are represented by a collection of binary variables indicating equality and inequality to the observed nominal categorical values for the variable.

`ConfidenceLevel`, `VarianceEstimatorFunction`, and `WorkingPrecision` are relevant to the computation of results after the initial fitting. These options can be set within `LinearModelFit` to specify the default settings for results obtained from the `FittedModel` object. These options can also be set within an already constructed `FittedModel` object to override the option values originally given to `LinearModelFit`.

Here are the default and mean squared error variance estimates.

```
In[10]:= {lm["EstimatedVariance"],
          lm["EstimatedVariance", VarianceEstimatorFunction → (Mean[#^2] &)]}
Out[10]= {6.71608, 6.04447}
```

`IncludeConstantBasis`, `LinearOffsetFunction`, `NominalVariables`, and `Weights` are relevant only to the fitting. Setting these options within an already constructed `FittedModel` object will have no further impact on the result.

A major feature of the model fitting framework is the ability to obtain results after the fitting. The full list of available results can be obtained from the `"Properties"` value.

This is the number of properties available for linear models.

```
In[11]:= lm["Properties"] // Length
Out[11]= 62
```

The properties include basic information about the data, fitted model, and numerous results and diagnostics.

`"BestFit"`	fitted function
`"BestFitParameters"`	parameter estimates
`"Data"`	the input data or design matrix and response vector

"DesignMatrix"	design matrix for the model
"Function"	best-fit pure function
"Response"	response values in the input data

Properties related to data and the fitted function.

The "BestFitParameters" property gives the fitted parameter values $\{\beta_0, \beta_1, \ldots\}$. "BestFit" is the fitted function $\beta_0 + \beta_1 f_1 + \beta_2 f_2 + \cdots$ and "Function" gives the fitted function as a pure function. The "DesignMatrix" is the design or model matrix for the data. "Response" gives the list of the response or y values from the original data.

"FitResiduals"	difference between actual and predicted responses
"StandardizedResiduals"	fit residuals divided by the standard error for each residual
"StudentizedResiduals"	fit residuals divided by single deletion error estimates

Types of residuals.

Residuals give a measure of the point-wise difference between the fitted values and the original responses. "FitResiduals" gives the differences between the observed and fitted values $\{y_1 - \hat{y}_1, y_2 - \hat{y}_2, \ldots\}$. "StandardizedResiduals" and "StudentizedResiduals" are scaled forms of the residuals. The i^{th} standardized residual is $(y_i - \hat{y}_i) / \sqrt{\hat{\sigma}^2 (1 - h_{ii}) / w_i}$ where $\hat{\sigma}^2$ is the estimated error variance, h_{ii} is the i^{th} diagonal element of the hat matrix, and w_i is the weight for the i^{th} data point. The i^{th} studentized residual uses the same formula with $\hat{\sigma}^2$ replaced by $\hat{\sigma}_{(i)}^2$, the variance estimate omitting the i^{th} data point.

"ANOVATable"	analysis of variance table
"ANOVATableDegreesOfFreedom"	degrees of freedom from the ANOVA table
"ANOVATableEntries"	unformatted array of values from the table
"ANOVATableFStatistics"	F statistics from the table
"ANOVATableMeanSquares"	mean square errors from the table
"ANOVATablePValues"	p-values from the table
"ANOVATableSumsOfSquares"	sums of squares from the table
"CoefficientOfVariation"	response mean divided by the estimated standard deviation

"EstimatedVariance"	estimate of the error variance
"PartialSumOfSquares"	changes in model sum of squares as nonconstant basis functions are removed
"SequentialSumOfSquares"	the model sum of squares partitioned componentwise

Properties related to the sum of squared errors.

"ANOVATable" gives a formatted analysis of variance table for the model. "ANOVATableEntries" gives the numeric entries in the table and the remaining ANOVATable properties give the elements of columns in the table so individual parts of the table can easily be used in further computations.

This gives a formatted ANOVA table for the fitted model.

```
In[12]:= lm["ANOVATable"]
```

Out[12]=

	DF	SS	MS	F Statistic	P-Value
x	1	9470.06	9470.06	1410.06	1.49794×10^{-18}
Error	18	120.889	6.71608		
Total	19	9590.95			

Here are the elements of the MS column of the table.

```
In[13]:= lm["ANOVATableMeanSquares"]
Out[13]= {9470.06, 6.71608}
```

"CorrelationMatrix"	parameter correlation matrix
"CovarianceMatrix"	parameter covariance matrix
"EigenstructureTable"	eigenstructure of the parameter correlation matrix
"EigenstructureTableEigenvalues"	eigenvalues from the table
"EigenstructureTableEntries"	unformatted array of values from the table
"EigenstructureTableIndexes"	index values from the table
"EigenstructureTablePartitions"	partitioning from the table
"ParameterConfidenceIntervals"	parameter confidence intervals
"ParameterConfidenceIntervalTable"	table of confidence interval information for the fitted parameters
"ParameterConfidenceIntervalTableEntries"	unformatted array of values from the table
"ParameterConfidenceRegion"	ellipsoidal parameter confidence region

"ParameterErrors"	standard errors for parameter estimates
"ParameterPValues"	p-values for parameter t statistics
"ParameterTable"	table of fitted parameter information
"ParameterTableEntries"	unformatted array of values from the table
"ParameterTStatistics"	t statistics for parameter estimates
"VarianceInflationFactors"	list of inflation factors for the estimated parameters

Properties and diagnostics for parameter estimates.

"CovarianceMatrix" gives the covariance between fitted parameters. The matrix is $\hat{\sigma}^2 \left(X^{\mathsf{T}} W X\right)^{-1}$ where $\hat{\sigma}^2$ is the variance estimate, X is the design matrix, and W is the diagonal matrix of weights. "CorrelationMatrix" is the associated correlation matrix for the parameter estimates. "ParameterErrors" is equivalent to the square root of the diagonal elements of the covariance matrix.

"ParameterTable" and "ParameterConfidenceIntervalTable" contain information about the individual parameter estimates, tests of parameter significance, and confidence intervals.

Here is some data.

In[14]:=
```
data = {{8.71, 6.92, 18.89}, {6.05, 5.97, 15.08}, {6.24, 0.99, 5.92},
    {8.25, 3.37, 11.39}, {6.58, 8.22, 20.77}, {4.14, 9., 21.09}, {4.35, 9.94, 24.32},
    {8.99, 4.47, 13.79}, {2.82, 3.91, 10.68}, {5.14, 0.4, 3.82}};
```

This fits a model using both predictor variables.

In[15]:=
```
lm2 = LinearModelFit[data, {x, y}, {x, y}]
```

Out[15]=
```
FittedModel[ 1.40308 + 0.340391 x + 2.08429 y ]
```

These are the formatted parameter and parameter confidence interval tables.

In[16]:=
```
lm2[{"ParameterTable", "ParameterConfidenceIntervalTable"}]
```

Out[16]=

	Estimate	Standard Error	t Statistic	P-Value
1	1.40308	0.595477	2.35622	0.0506221
x	0.340391	0.0782093	4.35231	0.00334539
y	2.08429	0.0496681	41.9643	1.13829×10^{-9}

	Estimate	Standard Error	Confidence Interval
1	1.40308	0.595477	{−0.00500488, 2.81116}
x	0.340391	0.0782093	{0.155456, 0.525327}
y	2.08429	0.0496681	{1.96684, 2.20174}

Here 99% confidence intervals are used in the table.

In[17]:=
```
lm2["ParameterConfidenceIntervalTable", ConfidenceLevel → .99]
```

Out[17]=

	Estimate	Standard Error	Confidence Interval
1	1.40308	0.595477	{−0.680788, 3.48694}
x	0.340391	0.0782093	{0.0666993, 0.614084}
y	2.08429	0.0496681	{1.91048, 2.2581}

The Estimate column of these tables is equivalent to "BestFitParameters". The t statistics are the estimates divided by the standard errors. Each p-value is the two-sided p-value for the t statistic and can be used to assess whether the parameter estimate is statistically significantly different from 0. Each confidence interval gives the upper and lower bounds for the parameter confidence interval at the level prescribed by the ConfidenceLevel option. The various ParameterTable and ParameterConfidenceIntervalTable properties can be used to get the columns or the unformatted array of values from the table.

"VarianceInflationFactors" is used to measure the multicollinearity between basis functions. The i^{th} inflation factor is equal to $1/(1-R_i^2)$ where R_i^2 is the coefficient of variation from fitting the i^{th} basis function to a linear function of the other basis functions. With IncludeConstantBasis -> True, the first inflation factor is for the constant term.

"EigenstructureTable" gives the eigenvalues, condition indices, and variance partitions for the nonconstant basis functions. The Index column gives the square root of the ratios of the eigenvalues to the largest eigenvalue. The column for each basis function gives the proportion of variation in that basis function explained by the associated eigenvector. "EigenstructureTablePartitions" gives the values in the variance partitioning for all basis functions in the table.

"BetaDifferences"	DFBETAS measures of influence on parameter values
"CatcherMatrix"	catcher matrix
"CookDistances"	list of Cook distances
"CovarianceRatios"	COVRATIO measures of observation influence
"DurbinWatsonD"	Durbin-Watson d statistic for autocorrelation
"FitDifferences"	DFFITS measures of influence on predicted values
"FVarianceRatios"	FVARATIO measures of observation influence
"HatDiagonal"	diagonal elements of the hat matrix
"SingleDeletionVariances"	list of variance estimates with the i^{th} data point omitted

Properties related to influence measures.

Point-wise measures of influence are often employed to assess whether individual data points have a large impact on the fitting. The hat matrix and catcher matrix play important roles in such diagnostics. The hat matrix is the matrix H such that $\hat{y} = H\,y$ where y is the observed

response vector and $\hat{y}$ is the predicted response vector. `"HatDiagonal"` gives the diagonal elements of the hat matrix. `"CatcherMatrix"` is the matrix C such that $\beta = C\,y$ where β is the fitted parameter vector.

`"FitDifferences"` gives the DFFITS values that provide a measure of influence of each data point on the fitted or predicted values. The i^{th} DFFITS value is given by $\sqrt{h_{ii}/(1-h_{ii})}\ r_{ti}$ where h_{ii} is the i^{th} hat diagonal and r_{ti} is the i^{th} studentized residual.

`"BetaDifferences"` gives the DFBETAS values that provide measures of influence of each data point on the parameters in the model. For a model with p parameters, the i^{th} element of `"BetaDifferences"` is a list of length p with the j^{th} value giving the measure the of the influence of data point i on the j^{th} parameter in the model. The i^{th} `"BetaDifferences"` vector can be written as $\{c_{i1}, \ldots, c_{ip}\}\sqrt{r_{ti}/(1-h_{ii})}\ \Big/\left(\sum_{j=1}^{n}\sum_{k=1}^{p} c_{jk}^2\right)$ where c_{jk} is the j,k^{th} element of the catcher matrix.

`"CookDistances"` gives the Cook distance measures of leverage given. The i^{th} Cook distance is given by $(h_{ii}/(1-h_{ii}))\ r_{si}/p$ where r_{si} is the i^{th} standardized residual.

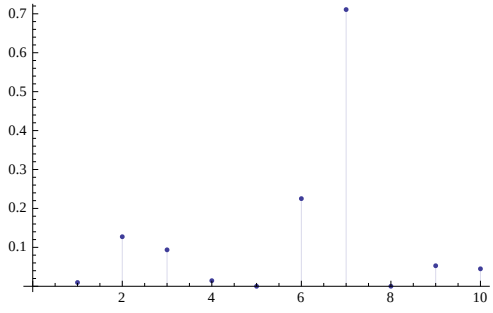
The i^{th} element of `"CovarianceRatios"` is given by $(n-p)^p \Big/ \left((1-h_{ii})\left(r_{ti}^2 + n - p - 1\right)^p\right)$ and the i^{th} `"FVarianceRatios"` value is equal to $\hat{\sigma}_{(i)}{}^2 \Big/ \left(\hat{\sigma}^2(1-h_{ii})\right)$ where $\hat{\sigma}_{(i)}{}^2$ is the i^{th} single deletion variance.

The Durbin-Watson d statistic `"DurbinWatsonD"` is used for testing the existence of a first-order autoregressive process. The d statistic is equivalent to $\sum_{i=1}^{n-1}(r_{i+1}-r_i)^2 \Big/ \sum_{i=1}^{n} r_i^2$ where r_i is the i^{th} residual.

This plots the Cook distances for the bivariate model.

```
In[18]:= ListPlot[lm2["CookDistances"], Filling → 0]
```

Out[18]=



"MeanPredictionBands"	confidence bands for mean predictions
"MeanPredictionConfidenceIntervals"	confidence intervals for the mean predictions
"MeanPredictionConfidenceIntervalTable"	table of confidence intervals for the mean predictions
"MeanPredictionConfidenceIntervalTableEntries"	unformatted array of values from the table
"MeanPredictionErrors"	standard errors for mean predictions
"PredictedResponse"	fitted values for the data
"SinglePredictionBands"	confidence bands based on single observations
"SinglePredictionConfidenceIntervals"	confidence intervals for the predicted response of single observations
"SinglePredictionConfidenceIntervalTable"	table of confidence intervals for the predicted response of single observations
"SinglePredictionConfidenceIntervalTableEntries"	unformatted array of values from the table
"SinglePredictionErrors"	standard errors for the predicted response of single observations

Properties of predicted values.

Tabular results for confidence intervals are given by "MeanPredictionConfidenceIntervalTable" and "SinglePredictionConfidenceIntervalTable". These include the observed and predicted responses, standard error estimates, and confidence intervals for each point. Mean prediction confidence intervals are often referred to simply as confidence intervals and single prediction confidence intervals are often referred to as prediction intervals.

"MeanPredictionBands" and "SinglePredictionBands" give functions of the predictor variables.

Here is the mean prediction table.

```
In[19]:= lm2["MeanPredictionConfidenceIntervalTable"]
```

Out[19]=

Observed	Predicted	Standard Error	Confidence Interval
18.89	18.7912	0.272818	{18.1461, 19.4363}
15.08	15.9057	0.155811	{15.5372, 16.2741}
5.92	5.59057	0.262819	{4.9691, 6.21204}
11.39	11.2354	0.236917	{10.6751, 11.7956}
20.77	20.7757	0.215751	{20.2656, 21.2859}
21.09	21.5709	0.271392	{20.9292, 22.2127}
24.32	23.6016	0.295281	{22.9034, 24.2999}
13.79	13.78	0.269774	{13.1421, 14.4179}
10.68	10.5126	0.315597	{9.76629, 11.2588}
3.82	3.9864	0.306026	{3.26277, 4.71004}

This gives the 90% mean prediction intervals.

```
In[20]:= lm2["MeanPredictionConfidenceIntervals", ConfidenceLevel → .9]
```

```
Out[20]= {{18.2743, 19.3081}, {15.6105, 16.2009}, {5.09263, 6.0885},
  {10.7865, 11.6842}, {20.367, 21.1845}, {21.0567, 22.0851}, {23.0422, 24.1611},
  {13.2689, 14.2911}, {9.91464, 11.1105}, {3.40661, 4.56619}}
```

`"AdjustedRSquared"`	R^2 adjusted for the number of model parameters
`"AIC"`	Akaike Information Criterion
`"BIC"`	Bayesian Information Criterion
`"RSquared"`	coefficient of determination R^2

Goodness of fit measures.

Goodness of fit measures are used to assess how well a model fits or to compare models. The coefficient of determination `"RSquared"` is the ratio of the model sum of squares to the total sum of squares. `"AdjustedRSquared"` penalizes for the number of parameters in the model and is given by $1 - (\frac{n-1}{n-p})\ (1 - R^2)$.

`"AIC"` and `"BIC"` are likelihood-based goodness of fit measures. Both are equal to -2 times the log-likelihood for the model plus $k\,p$ where p is the number of parameters to be estimated including the estimated variance. For `"AIC"` k is 2, and for `"BIC"` k is $\log(n)$.

Generalized Linear Models

The linear model can be seen as a model where each response value y is an observation from a normal distribution with mean value $\hat{y} = \beta_0 + \beta_1 f_1 + \beta_2 f_2 + \cdots$. The generalized linear model extends to models of the form $\hat{y} = g^{-1}(\beta_0 + \beta_1 f_1 + \beta_2 f_2 + \ldots)$ with each y assumed to be an observation from a distribution of known exponential family form with mean $\hat{y}$ and g being an invertible function over the support of the exponential family. Models of this sort can be obtained via `GeneralizedLinearModelFit`.

`GeneralizedLinearModelFit[{y1,y2,…},{f1,f2,…},x]`	obtain a generalized linear model with basis functions f_i and a single predictor variable x
`GeneralizedLinearModelFit[{{x11,x12,…,y1},{x21,x22,…,y2}},{f1,f2,…},{x1,x2,…}]`	obtain a generalized linear model of multiple predictor variables x_i
`GeneralizedLinearModelFit[{m,v}]`	obtain a generalized linear model based on a design matrix m and response vector v

Generalized linear model fitting.

The invertible function g is called the link function and the linear combination $\beta_0 + \beta_1 f_1 + \beta_2 f_2 + \cdots$ is referred to as the linear predictor. Common special cases include the linear regression model with the identity link function and Gaussian or normal exponential family distribution, logit and probit models for probabilities, Poisson models for count data, and gamma and inverse Gaussian models.

The error variance is a function of the prediction $\hat{y}$ and is defined by the distribution up to a constant ϕ, which is referred to as the dispersion parameter. The error variance for a fitted value $\hat{y}$ can be written as $\hat{\phi}\, v(\hat{y})$, where $\hat{\phi}$ is an estimate of the dispersion parameter obtained from the observed and predicted response values, and $v(\hat{y})$ is the variance function associated with the exponential family evaluated at the value $\hat{y}$.

This fits a linear regression model.

```
In[21]:= glm1 = GeneralizedLinearModelFit[Sqrt[Range[10]], x, x]

Out[21]= FittedModel[ 0.973709 + 0.231476 x ]
```

This fits a canonical gamma regression model to the same data.

```
In[22]:= glm2 = GeneralizedLinearModelFit[
           Sqrt[Range[10]], x, x, ExponentialFamily → "Gamma"]

Out[22]= FittedModel[ 1 / (0.742193 - ≪20≫ x) ]
```

Here are the functional forms of the models.

```
In[23]:= Map[Normal, {glm1, glm2}]

Out[23]= {0.973709 + 0.231476 x, 1 / (0.742193 - 0.0467911 x)}
```

Logit and probit models are common binomial models for probabilities. The link function for the logit model is $\log\left(\frac{y}{1-y}\right)$ and the link for the probit model is the inverse CDF for a standard normal distribution $\sqrt{2}\ \text{erf}^{-1}(2y-1)$. Models of this type can be fitted via `GeneralizedLinearModelFit` with `ExponentialFamily -> "Binomial"` and the appropriate `LinkFunction` or via `LogitModelFit` and `ProbitModelFit`.

`LogitModelFit[`*data*`,`*funs*`,`*vars*`]`	obtain a logit model with basis functions *funs* and predictor variables *vars*
`LogitModelFit[{`*m*`,`*v*`}]`	obtain a logit model based on a design matrix *m* and response vector *v*
`ProbitModelFit[`*data*`,`*funs*`,`*vars*`]`	obtain a probit model fit to *data*
`ProbitModelFit[{`*m*`,`*v*`}]`	obtain a probit model fit to a design matrix *m* and response vector *v*

Logit and probit model fitting.

Parameter estimates are obtained via iteratively reweighted least squares with weights obtained from the variance function of the assumed distribution. Options for `GeneralizedLinearModelFit` include options for iteration fitting such as `PrecisionGoal`, options for model specification such as `LinkFunction`, and options for further analysis such as `ConfidenceLevel`.

option name	default value	
AccuracyGoal	Automatic	the accuracy sought
ConfidenceLevel	95/100	confidence level to use for parameters and predictions
CovarianceEstimatorFunction	"ExpectedInformation"	estimation method for the parameter covariance matrix
DispersionEstimatorFunction	Automatic	function for estimating the dispersion parameter
ExponentialFamily	Automatic	exponential family distribution for y
IncludeConstantBasis	True	whether to include a constant basis function
LinearOffsetFunction	None	known offset in the linear predictor
LinkFunction	Automatic	link function for the model
MaxIterations	Automatic	maximum number of iterations to use
NominalVariables	None	variables considered as nominal or categorical
PrecisionGoal	Automatic	the precision sought
Weights	Automatic	weights for data elements
WorkingPrecision	Automatic	precision used in internal computations

Options for GeneralizedLinearModelFit.

The options for LogitModelFit and ProbitModelFit are the same as for GeneralizedLinearModelFit except that ExponentialFamily and LinkFunction are defined by the logit or probit model and so are not options to LogitModelFit and ProbitModelFit.

ExponentialFamily can be "Binomial", "Gamma", "Gaussian", "InverseGaussian", "Poisson", or "QuasiLikelihood". Binomial models are valid for responses from 0 to 1. Poisson models are valid for non-negative integer responses. Gaussian or normal models are valid for real responses. Gamma and inverse Gaussian models are valid for positive responses. Quasi-likelihood models define the distributional structure in terms of a variance function $v(\mu)$ such that the log of the quasi-likelihood function for the i^{th} data point is given by $\int_{y_i}^{\hat{y}_i} \frac{y_i-\mu}{\phi\, v(\mu)}\, d\mu$. The variance function for a "QuasiLikelihood" model can be optionally set via ExponentialFamily -> {"QuasiLikelihood", "VarianceFunction" → *fun*} where *fun* is a pure function to be applied to fitted values.

DispersionEstimatorFunction defines a function for estimating the dispersion parameter ϕ. The estimate $\hat{\phi}$ is analogous to $\hat{\sigma}^2$ in linear and nonlinear regression models.

ExponentialFamily, IncludeConstantBasis, LinearOffsetFunction, LinkFunction, NominalVariables, and Weights all define some aspect of the model structure and optimization criterion and can only be set within GeneralizedLinearModelFit. All other options can be set either within GeneralizedLinearModelFit or passed to the FittedModel object when obtaining results and diagnostics. Options set in evaluations of FittedModel objects take precedence over settings given to GeneralizedLinearModelFit at the time of the fitting.

This gives 95% and 99% confidence intervals for the parameters in the gamma model.

```
In[24]:= {glm2["ParameterConfidenceIntervals"],
          glm2["ParameterConfidenceIntervals", ConfidenceLevel → .99]}
Out[24]= {{{0.62891, 0.855475}, {-0.0616093, -0.0319729}},
          {{0.593314, 0.891071}, {-0.0662656, -0.0273166}}}
```

"BestFit"	fitted function
"BestFitParameters"	parameter estimates
"Data"	the input data or design matrix and response vector
"DesignMatrix"	design matrix for the model
"Function"	best fit pure function
"LinearPredictor"	fitted linear combination
"Response"	response values in the input data

Properties related to data and the fitted function.

"BestFitParameters" gives the parameter estimates for the basis functions. "BestFit" gives the fitted function $g^{-1}\,(\hat{\beta}_0 + \hat{\beta}_1\, f_1 + \hat{\beta}_2\, f_2 + \ldots)$, and "LinearPredictor" gives the linear combination $\hat{\beta}_0 + \hat{\beta}_1\, f_1 + \hat{\beta}_2\, f_2 + \ldots$. "DesignMatrix" is the design or model matrix for the basis functions.

"Deviances"	deviances
"DevianceTable"	deviance table
"DevianceTableDegreesOfFreedom"	degrees of freedom differences from the table
"DevianceTableDeviances"	deviance differences from the table
"DevianceTableEntries"	unformatted array of values from the table
"DevianceTableResidualDegreesOfFreedom"	residual degrees of freedom from the table
"DevianceTableResidualDeviances"	residual deviances from the table

"EstimatedDispersion"	estimated dispersion parameter
"NullDeviance"	deviance for the null model
"NullDegreesOfFreedom"	degrees of freedom for the null model
"ResidualDeviance"	difference between the model deviance and null deviance
"ResidualDegreesOfFreedom"	difference between the model degrees of freedom and null degrees of freedom

Properties related to dispersion and model deviances.

Deviances and deviance tables generalize the model decomposition given by analysis of variance in linear models. The deviance for a single data point is $2\hat{\phi}\left(\ell_m(y) - \ell_m(\hat{y})\right)$ where ℓ_m is the log-likelihood function for the fitted model. "Deviances" gives a list of the deviance values for all data points. The sum of all deviances gives the model deviance. The model deviance can be decomposed as sums of squares are in an ANOVA table for linear models.

Here is some data with two predictor variables.

```
In[31]:= glmdata = {{3.58, 1.83, 0.21}, {3.58, 4.47, 0.17}, {3.58, 3.11, 0.19},
    {3.58, 4.2, 0.18}, {3.58, 2.83, 0.2}, {4.01, 2.53, 0.19}, {4.01, 3.63, 0.18},
    {4.01, 3.93, 0.17}, {4.01, 4.05, 0.17}, {4.01, 1.33, 0.22},
    {2.01, 3.46, 0.22}, {2.01, 1.77, 0.25}, {2.01, 2.44, 0.23}, {2.01, 2.39, 0.25},
    {2.01, 3.78, 0.2}, {3.59, 3.49, 0.19}, {3.59, 3.82, 0.19}, {3.59, 3.05, 0.2},
    {3.59, 4.51, 0.17}, {3.59, 3.37, 0.18}, {2.62, 1.42, 0.25}, {2.62, 3.9, 0.19},
    {2.62, 2.51, 0.21}, {2.62, 4.59, 0.18}, {2.62, 4.28, 0.19}};
```

This fits the data to an inverse Gaussian model.

```
In[32]:= glm3 = GeneralizedLinearModelFit[glmdata,
    {x, y}, {x, y}, ExponentialFamily → "InverseGaussian"]
```

```
Out[32]= FittedModel[ 1 / Sqrt[-0.852313 + <<18>> x + <<18>> y] ]
```

Here is the deviance table for the model.

```
In[33]:= glm3["DevianceTable"]
```

```
Out[33]=
   | DF | Deviance | Residual DF | Residual Deviance
   |    |          | 24          | 1.79112
x  | 1  | 0.782767 | 23          | 1.00835
y  | 1  | 0.913425 | 22          | 0.0949267
```

As with sums of squares, deviances are additive. The Deviance column of the table gives the increase in the model deviance when the given basis function is added. The Residual Deviance column gives the difference between the model deviance and the deviance for the submodel containing all previous terms in the table. For large samples, the increase in deviance is approximately χ^2 distributed with degrees of freedom equal to that for the basis function in the table.

"NullDeviance" is the deviance for the null model, the constant model equal to the mean of all observed responses for models including a constant or $g^{-1}(0)$ if a constant term is not included.

As with "ANOVATable", a number of properties are included to extract the columns or unformatted array of entries from "DevianceTable".

"AnscombeResiduals"	Anscombe residuals
"DevianceResiduals"	deviance residuals
"FitResiduals"	difference between actual and predicted responses
"LikelihoodResiduals"	likelihood residuals
"PearsonResiduals"	Pearson residuals
"StandardizedDevianceResiduals"	standardized deviance residuals
"StandardizedPearsonResiduals"	standardized Pearson residuals
"WorkingResiduals"	working residuals

Types of residuals.

"FitResiduals" is the list of residuals, differences between the observed and predicted responses. Given the distributional assumptions, the magnitude of the residuals is expected to change as a function of the predicted response value. Various types of scaled residuals are employed in the analysis of generalized linear models.

If d_i and $r_i = y_i - \hat{y}_i$ are the deviance and residual for the i^{th} data point, the i^{th} deviance residual is given by $r_{di} = \sqrt{d_i}\ sgn(r_i)$. The i^{th} Pearson residual is defined as $r_{pi} = r_i \Big/ \sqrt{v(\hat{y}_i)}$ where v is the variance function for the exponential family distribution. Standardized deviance residuals and standardized Pearson residuals include division by $\sqrt{\hat{\phi}(1-h_{ii})}$ where h_{ii} is the i^{th}diagonal of the hat matrix. "LikelihoodResiduals" values combine deviance and Pearson residuals. The i^{th} likelihood residual is given by $sgn(r_i)\sqrt{\left(r_{di}^2 + h_{ii}\, r_{pi}^2 \big/ (1-h_{ii})\right)\Big/\hat{\phi}}$.

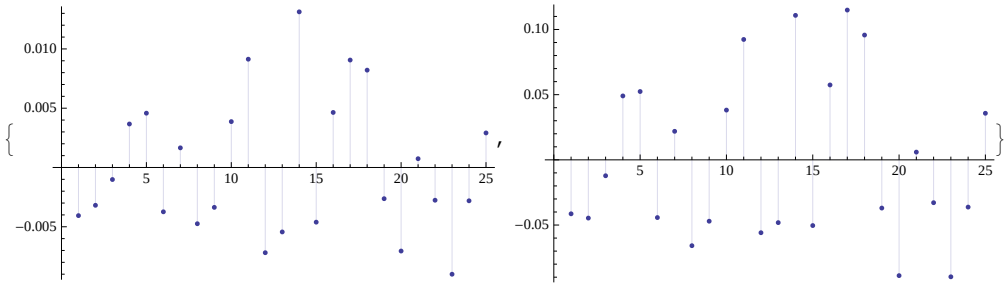
"AnscombeResiduals" provide a transformation of the residuals toward normality, so a plot of these residuals should be expected to look roughly like white noise. The i^{th} Anscombe residual can be written as $\sqrt[3]{v(\hat{y}_i)} \Big/ \sqrt{v(\hat{y}_i)} \int_{\hat{y}_i}^{y_i} \left(\sqrt[3]{v(\mu)}\right)^{-1} d\mu$.

"WorkingResiduals" gives the residuals from the last step of the iterative fitting. The i^{th} working residual can be obtained as $r_i \frac{\partial g(\mu)}{\partial \mu}$ evaluated at $\mu = \hat{y}_i$.

This plots the residuals and Anscombe residuals for the inverse Gaussian model.

```
In[41]:= Map[ListPlot[#, Filling → 0] &, glm3[{"FitResiduals", "AnscombeResiduals"}]]
```

Out[41]=



"CorrelationMatrix"	asymptotic parameter correlation matrix
"CovarianceMatrix"	asymptotic parameter covariance matrix
"ParameterConfidenceIntervals"	parameter confidence intervals
"ParameterConfidenceIntervalTable"	table of confidence interval information for the fitted parameters
"ParameterConfidenceIntervalTableEntries"	unformatted array of values from the table
"ParameterConfidenceRegion"	ellipsoidal parameter confidence region
"ParameterTableEntries"	unformatted array of values from the table
"ParameterErrors"	standard errors for parameter estimates
"ParameterPValues"	p-values for parameter z-statistics
"ParameterTable"	table of fitted parameter information
"ParameterZStatistics"	z-statistics for parameter estimates

Properties and diagnostics for parameter estimates.

"CovarianceMatrix" gives the covariance between fitted parameters and is very similar to the definition for linear models. With CovarianceEstimatorFunction -> "ExpectedInformation" the expected information matrix obtained from the iterative fitting is used. The matrix is $\hat{\phi}\left(X^{\mathsf{T}} W X\right)^{-1}$ where X is the design matrix, and W is the diagonal matrix of weights from the final stage of the fitting. The weights include both weights specified via the Weights option and the weights associated with the distribution's variance function. With

`CovarianceEstimatorFunction -> "ObservedInformation"` the matrix is given by $-\phi\, I^{-1}$ where I is the observed Fisher information matrix, which is the Hessian of the log-likelihood function with respect to parameters of the model.

`"CorrelationMatrix"` is the associated correlation matrix for the parameter estimates. `"ParameterErrors"` is equivalent to the square root of the diagonal elements of the covariance matrix. `"ParameterTable"` and `"ParameterConfidenceIntervalTable"` contain information about the individual parameter estimates, tests of parameter significance, and confidence intervals. The test statistics for generalized linear models asymptotically follow normal distributions.

`"CookDistances"`	list of Cook distances
`"HatDiagonal"`	diagonal elements of the hat matrix

Properties related to influence measures.

`"CookDistances"` and `"HatDiagonal"` extend the leverage measures from linear regression to generalized linear models. The hat matrix from which the diagonal elements are extracted is defined using the final weights of the iterative fitting.

The Cook distance measures of leverage are defined as in linear regression with standardized residuals replaced by standardized Pearson residuals. The i^{th} Cook distance is given by $(h_{ii} / (1-h_{ii})\, r_{spi} / p$ where r_{spi} is the i^{th} standardized Pearson residual.

`"PredictedResponse"`	fitted values for the data

Properties of predicted values.

`"AdjustedLikelihoodRatioIndex"`	Ben-Akiva and Lerman's adjusted likelihood ratio index
`"AIC"`	Akaike Information Criterion
`"BIC"`	Bayesian Information Criterion
`"CoxSnellPseudoRSquared"`	Cox and Snell's pseudo R^2
`"CraggUhlerPseudoRSquared"`	Cragg and Uhler's pseudo R^2
`"EfronPseudoRSquared"`	Efron's pseudo R^2
`"LikelihoodRatioIndex"`	McFadden's likelihood ratio index

"LikelihoodRatioStatistic"	likelihood ratio
"LogLikelihood"	log-likelihood for the fitted model
"PearsonChiSquare"	Pearson's χ^2 statistic

Goodness of fit measures.

"LogLikelihood" is the log-likelihood for the fitted model. "AIC" and "BIC" are penalized log-likelihood measures $2\ell + k\,p$ where ℓ is the log-likelihood for the fitted model, p is the number of parameters estimated including the dispersion parameter, and k is 2 for "AIC" and $\log(n)$ for "BIC" for a model of n data points. "LikelihoodRatioStatistic" is given by $2(\ell - \ell_0)$ where ℓ_0 is the log-likelihood for the null model.

A number of the goodness of fit measures generalize R^2 from linear regression as either a measure of explained variation or as a likelihood-based measure. "CoxSnellPseudoRSquared" is given by $1 - \left(e^{\ell_0 - \ell}\right)^{2/n}$. "CraggUhlerPseudoRSquared" is a scaled version of Cox and Snell's measure $\left(1 - \left(e^{\ell_0 - \ell}\right)^{2/n}\right) \Big/ \left(1 - \left(e^{\ell_0}\right)^{2/n}\right)$. "LikelihoodRatioIndex" involves the ratio of log-likelihoods $1 - \ell/\ell_0$, and "AdjustedLikelihoodRatioIndex" adjusts by penalizing for the number of parameters $1 - (\ell - p)/\ell_0$. "EfronPseudoRSquared" uses the sum of squares interpretation of R^2 and is given as $1 - \sum_{i=1}^{n} r_i^2 \Big/ \sum_{i=1}^{n} (y_i - \bar{y})^2$ where r_i is the i^{th} residual and $\bar{y}$ is the mean of the responses y_i.

"PearsonChiSquare" is equal to $\sum_{i=1}^{n} r_{pi}^2$ where the r_{pi} are Pearson residuals.

Nonlinear Models

A nonlinear least-squares model is an extension of the linear model where the model need not be a linear combination of basis function. The errors are still assumed to be independent and normally distributed. Models of this type can be fitted using the NonlinearModelFit function.

NonlinearModelFit[{y_1, y_2, ...}, *form*, {β_1, ...}, x]	obtain a nonlinear model of the function *form* with parameters β_i and a single parameter predictor variable x
NonlinearModelFit[{{x_{11}, ..., y_1}, {x_{21}, ..., y_2}}, *form*, {β_1, ...}, {x_1, ...}]	obtain a nonlinear model as a function of multiple predictor variables x_i
NonlinearModelFit[*data*, {*form*, *cons*}, {β_1, ...}, {x_1, ...}]	obtain a nonlinear model subject to the constraints *cons*

Nonlinear model fitting.

Nonlinear models have the form $\hat{y} = f(x_1, \ldots, x_i, \beta_1, \ldots, \beta_j)$ where $\hat{y}$ is the fitted or predicted value, the β_i are parameters to be fitted, and the x_i are predictor variables. As with any nonlinear optimization problem, a good choice of starting values for the parameters may be necessary. Starting values can be given using the same parameter specifications as for `FindFit`.

This fits a nonlinear model to a sequence of square roots.

```
In[25]:= nlm = NonlinearModelFit[Array[Sqrt, 20], Log[a + b x], {a, b}, x]
Out[25]= FittedModel[ Log[-0.748315 + 2.76912 x] ]
```

Options for model fitting and for model analysis are available.

option name	*default value*	
`AccuracyGoal`	`Automatic`	the accuracy sought
`ConfidenceLevel`	`95/100`	confidence level to use for parameters and predictions
`EvaluationMonitor`	`None`	expression to evaluate whenever *expr* is evaluated
`MaxIterations`	`Automatic`	maximum number of iterations to use
`Method`	`Automatic`	method to use
`PrecisionGoal`	`Automatic`	the precision sought
`StepMonitor`	`None`	the expression to evaluate whenever a step is taken
`VarianceEstimatorFunction`	`Automatic`	function for estimating the error variance
`Weights`	`Automatic`	weights for data elements
`WorkingPrecision`	`Automatic`	precision used in internal computations

Options for `NonlinearModelFit`.

General numeric options such as `AccuracyGoal`, `Method`, and `WorkingPrecision` are the same as for `FindFit`.

The `Weights` option specifies weight values for weighted nonlinear regression. The optimal fit is for a weighted sum of squared errors.

All other options can be relevant to computation of results after the initial fitting. They can be set within `NonlinearModelFit` for use in the fitting and to specify the default settings for results obtained from the `FittedModel` object. These options can also be set within an already constructed `FittedModel` object to override the option values originally given to `NonlinearModelFit`.

"BestFit"	fitted function
"BestFitParameters"	parameter estimates
"Data"	the input data
"Function"	best fit pure function
"Response"	response values in the input data

Properties related to data and the fitted function.

Basic properties of the data and fitted function for nonlinear models behave like the same properties for linear and generalized linear models with the exception that "BestFitParameters" returns a rule as is done for the result of FindFit.

This gives the fitted function and rules for the parameter estimates.

```
In[26]:= nlm[{"BestFit", "BestFitParameters"}]
Out[26]= {Log[-0.748315 + 2.76912 x], {a → -0.748315, b → 2.76912}}
```

Many diagnostics for nonlinear models extend or generalize concepts from linear regression. These extensions often rely on linear approximations or large sample approximations.

"FitResiduals"	difference between actual and predicted responses
"StandardizedResiduals"	fit residuals divided by the standard error for each residual
"StudentizedResiduals"	fit residuals divided by single deletion error estimates

Types of residuals.

As in linear regression, "FitResiduals" gives the differences between the observed and fitted values $\{y_1 - \hat{y}_1, y_2 - \hat{y}_2, \ldots\}$, and "StandardizedResiduals" and "StudentizedResiduals" are scaled forms of these differences.

The i^{th} standardized residual is $(y_i - \hat{y}_i) / \sqrt{\hat{\sigma}^2 (1 - h_{ii}) / w_i}$ where $\hat{\sigma}^2$ is the estimated error variance, h_{ii} is the i^{th} diagonal element of the hat matrix, and w_i is the weight for the i^{th} data point, and the i^{th} studentized residual is obtained by $\hat{\sigma}^2$ replacing with the i^{th} single deletion variance $\hat{\sigma}_{(i)}^2$. For nonlinear models a first-order approximation is used for the design matrix, which is needed to compute the hat matrix.

"ANOVATable"	analysis of variance table
"ANOVATableDegreesOfFreedom"	degrees of freedom from the ANOVA table
"ANOVATableEntries"	unformatted array of values from the table
"ANOVATableMeanSquares"	mean square errors from the table
"ANOVATableSumsOfSquares"	sums of squares from the table
"EstimatedVariance"	estimate of the error variance

Properties related to the sum of squared errors.

"ANOVATable" provides a decomposition of the variation in the data attributable to the fitted function and to the errors or residuals.

This gives the ANOVA table for the nonlinear model.

```
In[27]:= nlm["ANOVATable"]
```

Out[27]=

	DF	SS	MS
Model	2	208.604	104.302
Error	18	1.39635	0.0775748
Uncorrected Total	20	210.	
Corrected Total	19	19.8654	

The uncorrected total sums of squares gives the sum of squared responses, while the corrected total gives the sum of squared differences between the responses and their mean value.

"CorrelationMatrix"	asymptotic parameter correlation matrix
"CovarianceMatrix"	asymptotic parameter covariance matrix
"ParameterBias"	estimated bias in the parameter estimates
"ParameterConfidenceIntervals"	parameter confidence intervals
"ParameterConfidenceIntervalTable"	table of confidence interval information for the fitted parameters
"ParameterConfidenceIntervalTableEntries"	unformatted array of values from the table
"ParameterConfidenceRegion"	ellipsoidal parameter confidence region
"ParameterErrors"	standard errors for parameter estimates
"ParameterPValues"	p-values for parameter t statistics
"ParameterTable"	table of fitted parameter information
"ParameterTableEntries"	unformatted array of values from the table
"ParameterTStatistics"	t statistics for parameter estimates

Properties and diagnostics for parameter estimates.

"CovarianceMatrix" gives the approximate covariance between fitted parameters. The matrix is $\hat{\sigma}^2 (X^T W X)^{-1}$ where $\hat{\sigma}^2$ is the variance estimate, X is the design matrix for the linear approximation to the model, and W is the diagonal matrix of weights. "CorrelationMatrix" is the associated correlation matrix for the parameter estimates. "ParameterErrors" is equivalent to the square root of the diagonal elements of the covariance matrix.

"ParameterTable" and "ParameterConfidenceIntervalTable" contain information about the individual parameter estimates, tests of parameter significance, and confidence intervals obtained using the error estimates.

"CurvatureConfidenceRegion"	confidence region for curvature diagnostics
"FitCurvatureTable"	table of curvature diagnostics
"FitCurvatureTableEntries"	unformatted array of values from the table
"MaxIntrinsicCurvature"	measure of maximum intrinsic curvature
"MaxParameterEffectsCurvature"	measure of maximum parameter effects curvature

Curvature diagnostics.

The first-order approximation used for many diagnostics is equivalent to the model being linear in the parameters. If the parameter space near the parameter estimates is sufficiently flat, the linear approximations and any results that rely on first-order approximations can be deemed reasonable. Curvature diagnostics are used to assess whether the approximate linearity is reasonable. "FitCurvatureTable" is a table of curvature diagnostics.

"MaxIntrinsicCurvature" and "MaxParameterEffectsCurvature" are scaled measures of the normal and tangential curvatures of the parameter spaces at the best-fit parameter values. "CurvatureConfidenceRegion" is a scaled measure of the radius of curvature of the parameter space at the best-fit parameter values. If the normal and tangential curvatures are small relative to the value of the "CurvatureConfidenceRegion", the linear approximation is considered reasonable. Some rules of thumb suggest comparing the values directly, while others suggest comparing with half the "CurvatureConfidenceRegion".

Here is the curvature table for the nonlinear model.

In[28]:=
```
nlm["FitCurvatureTable"]
```

Out[28]=

	Curvature
Max Intrinsic	0.109997
Max Parameter Effects	0.311792
95. % Confidence Region	0.530405

"HatDiagonal"	diagonal elements of the hat matrix
"SingleDeletionVariances"	list of variance estimates with the i^{th} data point omitted

Properties related to influence measures.

The hat matrix is the matrix H such that $\hat{y} = H\,y$ where y is the observed response vector and $\hat{y}$ is the predicted response vector. "HatDiagonal" gives the diagonal elements of the hat matrix. As with other properties, H uses the design matrix for the linear approximation to the model.

The i^{th} element of "SingleDeletionVariances" is equivalent to $\left((n-p)\,\hat{\sigma} - r_i^2/(1-h_{ii})\right)/(n-p-1)$ where n is the number of data points, p is the number of parameters, h_{ii} is the i^{th} hat diagonal, $\hat{\sigma}$ is the variance estimate for the full dataset, and r_i is the i^{th} residual.

"MeanPredictionBands"	confidence bands for mean predictions
"MeanPredictionConfidenceIntervals"	confidence intervals for the mean predictions
"MeanPredictionConfidenceIntervalTable"	table of confidence intervals for the mean predictions
"MeanPredictionConfidenceIntervalTableEntries"	unformatted array of values from the table
"MeanPredictionErrors"	standard errors for mean predictions
"PredictedResponse"	fitted values for the data
"SinglePredictionBands"	confidence bands based on single observations
"SinglePredictionConfidenceIntervals"	confidence intervals for the predicted response of single observations
"SinglePredictionConfidenceIntervalTable"	table of confidence intervals for the predicted response of single observations
"SinglePredictionConfidenceIntervalTableEntries"	unformatted array of values from the table
"SinglePredictionErrors"	standard errors for the predicted response of single observations

Properties of predicted values.

Tabular results for confidence intervals are given by "MeanPredictionConfidenceIntervalTable" and "SinglePredictionConfidenceIntervalTable". These results are analogous to those for linear models obtained via LinearModelFit, again with first-order approximations used for the design matrix.

"MeanPredictionBands" and "SinglePredictionBands" give functions of the predictor variables.

Here the fitted function and mean prediction bands are obtained.

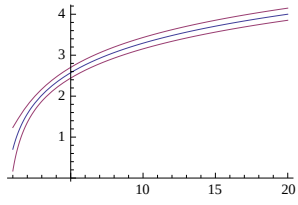
```
In[29]:= {fit[x_], mp[x_]} = nlm[{"BestFit", "MeanPredictionBands"}]
```

$$\text{Out[29]=}\ \left\{\text{Log}[-0.748315 + 2.76912\,x],\ \left\{-2.10092\sqrt{\frac{0.382399 - 0.165343\,x + 0.0443921\,x^2}{(0.748315 - 2.76912\,x)^2}} + \text{Log}[-0.748315 + 2.76912\,x],\right.\right.$$
$$\left.\left.2.10092\sqrt{\frac{0.382399 - 0.165343\,x + 0.0443921\,x^2}{(0.748315 - 2.76912\,x)^2}} + \text{Log}[-0.748315 + 2.76912\,x]\right\}\right\}$$

This plots the fitted curve and confidence bands.

```
In[30]:= Plot[{fit[x], mp[x]}, {x, 1, 20}]
```

Out[30]=



"AdjustedRSquared"	R^2 adjusted for the number of model parameters
"AIC"	Akaike Information Criterion
"BIC"	Bayesian Information Criterion
"RSquared"	coefficient of determination R^2

Goodness of fit measures.

"AdjustedRSquared", "AIC", "BIC", and "RSquared" are all direct extensions of the measures as defined for linear models. The coefficient of determination "RSquared" is the ratio of the model sum of squares to the total sum of squares. "AdjustedRSquared" penalizes for the number of parameters in the model and is given by $1 - (\frac{n-1}{n-p})(1 - R^2)$.

"AIC" and "BIC" are equal to -2 times the log-likelihood for the model plus $k\,p$ where p is the number of parameters to be estimated including the estimated variance. For "AIC" k is 2, and for "BIC" k is $\log(n)$.

Approximate Functions and Interpolation

In many kinds of numerical computations, it is convenient to introduce *approximate functions*. Approximate functions can be thought of as generalizations of ordinary approximate real numbers. While an approximate real number gives the value to a certain precision of a single numerical quantity, an approximate function gives the value to a certain precision of a quantity which depends on one or more parameters. *Mathematica* uses approximate functions, for example, to represent numerical solutions to differential equations obtained with `NDSolve`, as discussed in "Numerical Differential Equations".

Approximate functions in *Mathematica* are represented by `InterpolatingFunction` objects. These objects work like the pure functions discussed in "Pure Functions". The basic idea is that when given a particular argument, an `InterpolatingFunction` object finds the approximate function value that corresponds to that argument.

The `InterpolatingFunction` object contains a representation of the approximate function based on interpolation. Typically it contains values and possibly derivatives at a sequence of points. It effectively assumes that the function varies smoothly between these points. As a result, when you ask for the value of the function with a particular argument, the `InterpolatingFunction` object can interpolate to find an approximation to the value you want.

`Interpolation[{`f_1`,`f_2`,...}]`	construct an approximate function with values f_i at successive integers
`Interpolation[{{`x_1`,`f_1`},{`x_2`,`f_2`},...}]`	construct an approximate function with values f_i at points x_i

Constructing approximate functions.

Here is a table of the values of the sine function.

```
In[1]:= Table[{x, Sin[x]}, {x, 0, 2, 0.25}]
Out[1]= {{0., 0.}, {0.25, 0.247404}, {0.5, 0.479426}, {0.75, 0.681639},
  {1., 0.841471}, {1.25, 0.948985}, {1.5, 0.997495}, {1.75, 0.983986}, {2., 0.909297}}
```

This constructs an approximate function which represents these values.

```
In[2]:= sin = Interpolation[%]
Out[2]= InterpolatingFunction[{{0., 2.}}, <>]
```

The approximate function reproduces each of the values in the original table.

```
In[3]:= sin[0.25]
Out[3]= 0.247404
```

It also allows you to get approximate values at other points.

```
In[4]:= sin[0.3]
Out[4]= 0.2955
```

In this case the interpolation is a fairly good approximation to the true sine function.

```
In[5]:= Sin[0.3]
Out[5]= 0.29552
```

You can work with approximate functions much as you would with any other *Mathematica* functions. You can plot approximate functions, or perform numerical operations such as integration or root finding.

If you give a non-numerical argument, the approximate function is left in symbolic form.

```
In[6]:= sin[x]
Out[6]= InterpolatingFunction[{{0., 2.}}, <>][x]
```

Here is a numerical integral of the approximate function.

```
In[7]:= NIntegrate[sin[x]^2, {x, 0, Pi / 2}]
Out[7]= 0.78531
```

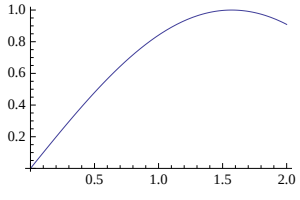
Here is the same numerical integral for the true sine function.

```
In[8]:= NIntegrate[Sin[x]^2, {x, 0, Pi / 2}]
Out[8]= 0.785398
```

A plot of the approximate function is essentially indistinguishable from the true sine function.

```
In[9]:= Plot[sin[x], {x, 0, 2}]
```

Out[9]=



If you differentiate an approximate function, *Mathematica* will return another approximate function that represents the derivative.

This finds the derivative of the approximate sine function, and evaluates it at $\pi/6$.

```
In[10]:= sin'[Pi / 6]
Out[10]= 0.865372
```

The result is close to the exact one.

```
In[11]:= N[Cos[Pi / 6]]
Out[11]= 0.866025
```

`InterpolatingFunction` objects contain all the information *Mathematica* needs about approximate functions. In standard *Mathematica* output format, however, only the part that gives the domain of the `InterpolatingFunction` object is printed explicitly. The lists of actual parameters used in the `InterpolatingFunction` object are shown only in iconic form.

In standard output format, the only part of an `InterpolatingFunction` object printed explicitly is its domain.

```
In[12]:= sin
Out[12]= InterpolatingFunction[{{0., 2.}}, <>]
```

If you ask for a value outside of the domain, *Mathematica* prints a warning, then uses extrapolation to find a result.

```
In[13]:= sin[3]

InterpolatingFunction::dmval:
  Input value {3} lies outside the range of data in the interpolating function. Extrapolation will be used. >>
Out[13]= 0.0155471
```

The more information you give about the function you are trying to approximate, the better the approximation *Mathematica* constructs can be. You can, for example, specify not only values of the function at a sequence of points, but also derivatives.

`Interpolation[{{{`x_1`}, `f_1`, `df_1`, `ddf_1`, ...}, ...}]`	construct an approximate function with specified derivatives at points x_i

Constructing approximate functions with specified derivatives.

This interpolates through the values of the sine function and its first derivative.

```
In[14]:= sind = Interpolation[Table[{{x}, Sin[x], Cos[x]}, {x, 0, 2, 0.25}]]
Out[14]= InterpolatingFunction[{{0., 2.}}, <>]
```

This finds a better approximation to the derivative than the previous interpolation.

```
In[15]:= sind'[Pi / 6]
Out[15]= 0.865974
```

`Interpolation` works by fitting polynomial curves between the points you specify. You can use the option `InterpolationOrder` to specify the degree of these polynomial curves. The default setting is `InterpolationOrder -> 3`, yielding cubic curves.

This makes a table of values of the cosine function.

```
In[16]:= tab = Table[{x, Cos[x]}, {x, 0, 6}];
```

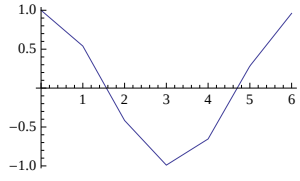
This creates an approximate function using linear interpolation between the values in the table.

```
In[17]:= Interpolation[tab, InterpolationOrder -> 1]
Out[17]= InterpolatingFunction[{{0, 6}}, <>]
```

The approximate function consists of a collection of straight-line segments.

```
In[18]:= Plot[%[x], {x, 0, 6}]
```

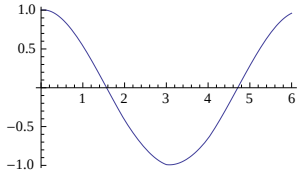
Out[18]=



With the default setting `InterpolationOrder -> 3`, cubic curves are used, and the function looks smooth.

```
In[19]:= Plot[Evaluate[Interpolation[tab]][x], {x, 0, 6}]
```

Out[19]=



Increasing the setting for `InterpolationOrder` typically leads to smoother approximate functions. However, if you increase the setting too much, spurious wiggles may develop.

`ListInterpolation[{{`f_{11}`,`f_{12}`,…},{`f_{21}`,…},…}]`	construct an approximate function from a two-dimensional grid of values at integer points
`ListInterpolation[`*list*`,{{`x_{min}`,`x_{max}`},{`y_{min}`,`y_{max}`}}]`	assume the values are from an evenly spaced grid with the specified domain
`ListInterpolation[`*list*`,{{`x_1`,`x_2`,…},{`y_1`,`y_2`,…}}]`	assume the values are from a grid with the specified grid lines

Interpolating multidimensional arrays of data.

This interpolates an array of values from integer grid points.

```
In[20]:= ListInterpolation[Table[1.5 / (x^2 + y^3), {x, 10}, {y, 15}]]
Out[20]= InterpolatingFunction[{{1., 10.}, {1., 15.}}, <>]
```

Here is the value at a particular position.

```
In[21]:= %[6.5, 7.2]
Out[21]= 0.00360759
```

Here is another array of values.

```
In[22]:= tab = Table[1.5 / (x^2 + y^3), {x, 5.5, 7.2, .2}, {y, 2.3, 8.9, .1}];
```

To interpolate this array you explicitly have to tell *Mathematica* the domain it covers.

```
In[23]:= ListInterpolation[tab, {{5.5, 7.2}, {2.3, 8.9}}]
Out[23]= InterpolatingFunction[{{5.5, 7.2}, {2.3, 8.9}}, <>]
```

`ListInterpolation` works for arrays of any dimension, and in each case it produces an `InterpolatingFunction` object which takes the appropriate number of arguments.

This interpolates a three-dimensional array.

```
In[24]:= ListInterpolation[Array[#1^2 + #2^2 - #3^2 &, {10, 10, 10}]];
```

The resulting `InterpolatingFunction` object takes three arguments.

```
In[25]:= %[3.4, 7.8, 2.6]
Out[25]= 65.64
```

Mathematica can handle not only purely numerical approximate functions, but also ones which involve symbolic parameters.

This generates an InterpolatingFunction that depends on the parameters a and b.

```
In[26]:= sfun = ListInterpolation[{1 + a, 2, 3, 4 + b, 5}]
Out[26]= InterpolatingFunction[{{1, 5}}, <>]
```

This shows how the interpolated value at 2.2 depends on the parameters.

```
In[27]:= sfun[2.2] // Simplify
Out[27]= 2.2 - 0.048 a - 0.032 b
```

With the default setting for InterpolationOrder used, the value at this point no longer depends on a.

```
In[28]:= sfun[3.8] // Simplify
Out[28]= 3.8 + 0.864 b
```

In working with approximate functions, you can quite often end up with complicated combinations of InterpolatingFunction objects. You can always tell *Mathematica* to produce a single InterpolatingFunction object valid over a particular domain by using FunctionInterpolation.

This generates a new InterpolatingFunction object valid in the domain 0 to 1.

```
In[29]:= FunctionInterpolation[x + sin[x^2], {x, 0, 1}]
Out[29]= InterpolatingFunction[{{0., 1.}}, <>]
```

This generates a nested InterpolatingFunction object.

```
In[30]:= ListInterpolation[{3, 4, 5, sin[a], 6}]
Out[30]= InterpolatingFunction[{{1, 5}}, <>]
```

This produces a pure two-dimensional InterpolatingFunction object.

```
In[31]:= FunctionInterpolation[a^2 + %[x], {x, 1, 3}, {a, 0, 1.5}]
Out[31]= InterpolatingFunction[{{1., 3.}, {0., 1.5}}, <>]
```

FunctionInterpolation[*expr*, {*x*, x_{min}, x_{max}}]	construct an approximate function by evaluating *expr* with *x* ranging from x_{min} to x_{max}
FunctionInterpolation[*expr*, {*x*, x_{min}, x_{max}}, {*y*, y_{min}, y_{max}}, ...]	construct a higher-dimensional approximate function

Constructing approximate functions by evaluating expressions.

Discrete Fourier Transforms

A common operation in analyzing various kinds of data is to find the discrete Fourier transform (or spectrum) of a list of values. The idea is typically to pick out components of the data with particular frequencies or ranges of frequencies.

Fourier[{u_1, u_2, ..., u_n}]	discrete Fourier transform
InverseFourier[{v_1, v_2, ..., v_n}]	inverse discrete Fourier transform

Discrete Fourier transforms.

Here is some data, corresponding to a square pulse.

```
In[1]:= {-1, -1, -1, -1, 1, 1, 1, 1}
Out[1]= {-1, -1, -1, -1, 1, 1, 1, 1}
```

Here is the discrete Fourier transform of the data. It involves complex numbers.

```
In[2]:= Fourier[%]
Out[2]= {0. + 0. i, -0.707107 - 1.70711 i, 0. + 0. i, -0.707107 - 0.292893 i,
          0. + 0. i, -0.707107 + 0.292893 i, 0. + 0. i, -0.707107 + 1.70711 i}
```

Here is the inverse discrete Fourier transform.

```
In[3]:= InverseFourier[%]
Out[3]= {-1., -1., -1., -1., 1., 1., 1., 1.}
```

Fourier works whether or not your list of data has a length which is a power of two.

```
In[4]:= Fourier[{1, -1, 1}]
Out[4]= {0.57735 + 0. i, 0.57735 - 1. i, 0.57735 + 1. i}
```

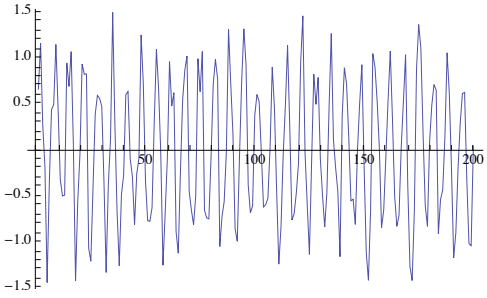
This generates a list of 200 elements containing a periodic signal with random noise added.

```
In[5]:= data = Table[N[Sin[30 × 2 Pi n / 200] + (RandomReal[] - 1 / 2)], {n, 200}];
```

The data looks fairly random if you plot it directly.

```
In[6]:= ListLinePlot[data]
```

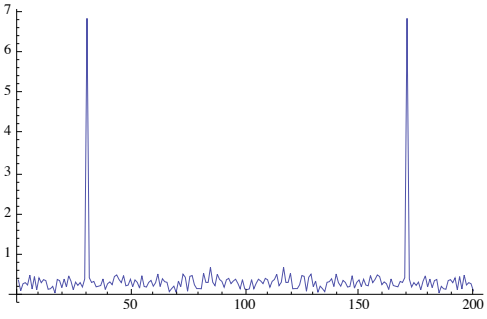
Out[6]=



The discrete Fourier transform, however, shows a strong peak at $30 + 1$, and a symmetric peak at $201 - 30$, reflecting the frequency component of the original signal near $30/200$.

```
In[7]:= ListLinePlot[Abs[Fourier[data]], PlotRange -> All]
```

Out[7]=



In *Mathematica*, the discrete Fourier transform v_s of a list u_r of length n is by default defined to be $\frac{1}{\sqrt{n}} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$. Notice that the zero frequency term appears at position 1 in the resulting list.

The inverse discrete Fourier transform u_r of a list v_s of length n is by default defined to be $\frac{1}{\sqrt{n}} \sum_{s=1}^{n} v_s \, e^{-2\pi i (r-1)(s-1)/n}$.

In different scientific and technical fields different conventions are often used for defining discrete Fourier transforms. The option `FourierParameters` allows you to choose any of these conventions you want.

common convention	*setting*	*discrete Fourier transform*	*inverse discrete Fourier transform*
Mathematica default	{0, 1}	$\frac{1}{n^{1/2}} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$	$\frac{1}{n^{1/2}} \sum_{s=1}^{n} v_s \, e^{-2\pi i (r-1)(s-1)/n}$
data analysis	{-1, 1}	$\frac{1}{n} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$	$\sum_{s=1}^{n} v_s \, e^{-2\pi i (r-1)(s-1)/n}$
signal processing	{1, -1}	$\sum_{r=1}^{n} u_r \, e^{-2\pi i (r-1)(s-1)/n}$	$\frac{1}{n} \sum_{s=1}^{n} v_s \, e^{2\pi i (r-1)(s-1)/n}$
general case	{*a*, *b*}	$\frac{1}{n^{(1-a)/2}} \sum_{r=1}^{n} u_r \, e^{2\pi i b (r-1)(s-1)/n}$	$\frac{1}{n^{(1+a)/2}} \sum_{s=1}^{n} v_s \, e^{-2\pi i b (r-1)(s-1)/n}$

Typical settings for `FourierParameters` with various conventions.

`Fourier[{{`u_{11}`,`u_{12}`,...},{`u_{21}`,`u_{22}`,...},...}]`	two-dimensional discrete Fourier transform

Two-dimensional discrete Fourier transform.

Mathematica can find discrete Fourier transforms for data in any number of dimensions. In n dimensions, the data is specified by a list nested n levels deep. Two-dimensional discrete Fourier transforms are often used in image processing.

One issue with the usual discrete Fourier transform for real data is that the result is complex-valued. There are variants of real discrete Fourier transforms that have real results. *Mathematica* has commands for computing the discrete cosine transform and the discrete sine transform.

`FourierDCT[`*list*`]`	Fourier discrete cosine transform of a list of real numbers
`FourierDST[`*list*`]`	Fourier discrete sine transform of a list of real numbers

Discrete real Fourier transforms.

Here is some data, corresponding to a square pulse.

```
In[8]:= pulse = {-1, -1, -1, -1, -1, 1, 1, 1, 1, 1}
Out[8]= {-1, -1, -1, -1, -1, 1, 1, 1, 1, 1}
```

Here is the Fourier discrete cosine transform of the data.

```
In[9]:= FourierDCT[pulse]
Out[9]= {0., -2.02147, 3.29753×10^-17, 0.696552, -5.68065×10^-17,
 -0.447214, -4.12723×10^-17, 0.354911, 4.76294×10^-17, -0.32017}
```

Here is the Fourier discrete sine transform of the data.

```
In[10]:= FourierDST[pulse]
Out[10]= {8.03365×10^-17, -2.04667, -3.85062×10^-17, -9.37835×10^-18,
          0., -0.781758, 9.84259×10^-17, 4.4886×10^-17, 5.83679×10^-17, -0.632456}
```

There are four types each of Fourier discrete sine and cosine transforms typically in use, denoted by number or sometimes roman numeral as in "DCTII" for the discrete cosine transform of type 2.

`FourierDCT[`*list*`,`*m*`]`	Fourier discrete cosine transform of type *m*
`FourierDST[`*list*`,`*m*`]`	Fourier discrete sine transform of type *m*

Discrete real Fourier transforms of different types.

The default is type 2 for both `FourierDCT` and `FourierDST`.

Mathematica does not need `InverseFourierDCT` or `InverseFourierDST` functions because `FourierDCT` and `FourierDST` are their own inverses when used with the appropriate type. The inverse transforms for types 1, 2, 3, 4 are types 1, 3, 2, 4, respectively.

Check that the type 3 transform is the inverse of the type 2 transform.

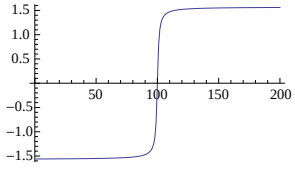
```
In[11]:= FourierDCT[FourierDCT[pulse, 2], 3]
Out[11]= {-1., -1., -1., -1., -1., 1., 1., 1., 1., 1.}
```

The discrete real transforms are convenient to use for data or image compression.

Here is data that might be like a front or an edge.

```
In[12]:= data = Table[ArcTan[x - 100], {x, 1., 200.}];
         ListLinePlot[data]
```

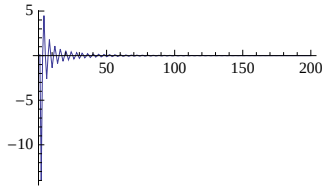
Out[12]=



The discrete cosine transform has most of the information in the first few modes.

```
In[13]:= dct = FourierDCT[data];
         ListLinePlot[dct, PlotRange → All]
```

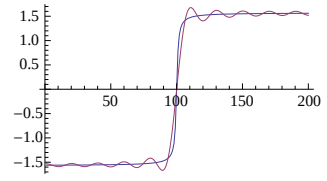
Out[13]=



Reconstruct the front from only the first 20 modes (1/10 of the original data size). The oscillations are a consequence of the truncation and are known to show up in image processing applications as well.

```
In[14]:= tdata = FourierDCT[PadRight[Take[dct, 20], 200, 0], 3];
         ListLinePlot[{data, tdata}]
```

Out[14]=



Convolutions and Correlations

Convolution and correlation are central to many kinds of operations on lists of data. They are used in such areas as signal and image processing, statistical data analysis, approximations to partial differential equations, as well as operations on digit sequences and power series.

In both convolution and correlation the basic idea is to combine a kernel list with successive sublists of a list of data. The *convolution* of a kernel K_r with a list u_s has the general form $\sum_r K_r u_{s-r}$, while the *correlation* has the general form $\sum_r K_r u_{s+r}$.

`ListConvolve[`*kernel*`,`*list*`]`	form the convolution of *kernel* with *list*
`ListCorrelate[`*kernel*`,`*list*`]`	form the correlation of *kernel* with *list*

Convolution and correlation of lists.

This forms the convolution of the kernel $\{x, y\}$ with a list of data.

```
In[1]:= ListConvolve[{x, y}, {a, b, c, d, e}]
Out[1]= {b x + a y, c x + b y, d x + c y, e x + d y}
```

This forms the correlation.

```
In[2]:= ListCorrelate[{x, y}, {a, b, c, d, e}]
Out[2]= {a x + b y, b x + c y, c x + d y, d x + e y}
```

In this case reversing the kernel gives exactly the same result as `ListConvolve`.

```
In[3]:= ListCorrelate[{y, x}, {a, b, c, d, e}]
Out[3]= {b x + a y, c x + b y, d x + c y, e x + d y}
```

This forms successive differences of the data.

```
In[4]:= ListCorrelate[{-1, 1}, {a, b, c, d, e}]
Out[4]= {-a + b, -b + c, -c + d, -d + e}
```

In forming sublists to combine with a kernel, there is always an issue of what to do at the ends of the list of data. By default, `ListConvolve` and `ListCorrelate` never form sublists which would "overhang" the ends of the list of data. This means that the output you get is normally shorter than the original list of data.

With an input list of length 6, the output is in this case of length 4.

```
In[5]:= ListCorrelate[{1, 1, 1}, Range[6]]
Out[5]= {6, 9, 12, 15}
```

In practice one often wants to get output that is as long as the original list of data. To do this requires including sublists that overhang one or both ends of the list of data. The additional elements needed to form these sublists must be filled in with some kind of "padding". By default, *Mathematica* takes copies of the original list to provide the padding, thus effectively treating the list as being cyclic.

`ListCorrelate[`*kernel*`,`*list*`]`	do not allow overhangs on either side (result shorter than *list*)
`ListCorrelate[`*kernel*`,`*list*`,1]`	allow an overhang on the right (result same length as *list*)
`ListCorrelate[`*kernel*`,`*list*`,-1]`	allow an overhang on the left (result same length as *list*)
`ListCorrelate[`*kernel*`,`*list*`,{-1,1}]`	allow overhangs on both sides (result longer than *list*)
`ListCorrelate[`*kernel*`,`*list*`,{`k_L`,`k_R`}]`	allow particular overhangs on left and right

Controlling how the ends of the list of data are treated.

The default involves no overhangs.

```
In[6]:= ListCorrelate[{x, y}, {a, b, c, d}]
Out[6]= {a x + b y, b x + c y, c x + d y}
```

The last term in the last element now comes from the beginning of the list.

```
In[7]:= ListCorrelate[{x, y}, {a, b, c, d}, 1]
Out[7]= {a x + b y, b x + c y, c x + d y, d x + a y}
```

Now the first term of the first element and the last term of the last element both involve wraparound.

```
In[8]:= ListCorrelate[{x, y}, {a, b, c, d}, {-1, 1}]
Out[8]= {d x + a y, a x + b y, b x + c y, c x + d y, d x + a y}
```

In the general case `ListCorrelate[`*kernel*`, `*list*`, {`k_L`, `k_R`}]` is set up so that in the first element of the result, the first element of *list* appears multiplied by the element at position k_L in *kernel*, and in the last element of the result, the last element of *list* appears multiplied by the element at position k_R in *kernel*. The default case in which no overhang is allowed on either side thus corresponds to `ListCorrelate[`*kernel*`, `*list*`, {1, -1}]`.

With a kernel of length 3, alignments `{-1, 2}` always make the first and last elements of the result the same.

```
In[9]:= ListCorrelate[{x, y, z}, {a, b, c, d}, {-1, 2}]
Out[9]= {c x + d y + a z, d x + a y + b z, a x + b y + c z, b x + c y + d z, c x + d y + a z}
```

For many kinds of data, it is convenient to assume not that the data is cyclic, but rather that it is padded at either end by some fixed element, often 0, or by some sequence of elements.

`ListCorrelate[`*kernel*`,`*list*`,`*klist*`,`*p*`]`	pad with element *p*
`ListCorrelate[`*kernel*`,`*list*`,`*klist*`,{`p_1`,`p_2`,...}]`	pad with cyclic repetitions of the p_i
`ListCorrelate[`*kernel*`,`*list*`,`*klist*`,`*list*`]`	pad with cyclic repetitions of the original data

Controlling the padding for a list of data.

This pads with element *p*.

```
In[10]:= ListCorrelate[{x, y}, {a, b, c, d}, {-1, 1}, p]
Out[10]= {p x + a y, a x + b y, b x + c y, c x + d y, d x + p y}
```

A common case is to pad with zero.

```
In[11]:= ListCorrelate[{x, y}, {a, b, c, d}, {-1, 1}, 0]
Out[11]= {a y, a x + b y, b x + c y, c x + d y, d x}
```

When the padding is indicated by $\{p, q\}$, the list $\{a, b, c\}$ overlays $\{\ldots, p, q, p, q, \ldots\}$ with a p aligned under the a.

```
In[12]:= ListCorrelate[{x, y, z}, {a, b, c}, {-1, 1}, {p, q}]
Out[12]= {p x + q y + a z, q x + a y + b z, a x + b y + c z, b x + c y + q z, c x + q y + p z}
```

Different choices of kernel allow `ListConvolve` and `ListCorrelate` to be used for different kinds of computations.

This finds a moving average of data.

```
In[13]:= ListCorrelate[{1, 1, 1} / 3, {a, b, c, d, e}]
```

$$\text{Out[13]}= \left\{\frac{a}{3}+\frac{b}{3}+\frac{c}{3}, \frac{b}{3}+\frac{c}{3}+\frac{d}{3}, \frac{c}{3}+\frac{d}{3}+\frac{e}{3}\right\}$$

Here is a Gaussian kernel.

```
In[14]:= kern = Table[Exp[-n^2 / 100] / Sqrt[2. Pi], {n, -10, 10}];
```

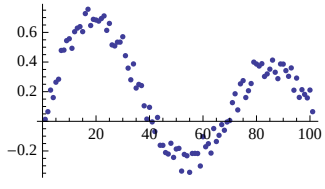
This generates some "data".

```
In[15]:= data = Table[BesselJ[1, x] + 0.2 RandomReal[], {x, 0, 10, .1}];
```

Here is a plot of the data.

```
In[16]:= ListPlot[data]
```

Out[16]=



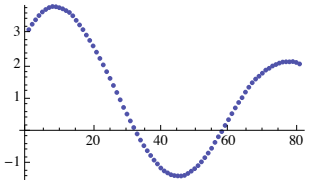
This convolves the kernel with the data.

```
In[17]:= ListConvolve[kern, data];
```

The result is a smoothed version of the data.

```
In[18]:= ListPlot[%]
```

Out[18]=



You can use `ListConvolve` and `ListCorrelate` to handle symbolic as well as numerical data.

This forms the convolution of two symbolic lists.

```
In[19]:= ListConvolve[{a, b, c}, {u, v, w}, {1, -1}, 0]
Out[19]= {a u, b u + a v, c u + b v + a w, c v + b w, c w}
```

The result corresponds exactly with the coefficients in the expanded form of this product of polynomials.

```
In[20]:= Expand[(a + b x + c x^2) (u + v x + w x^2)]
Out[20]= a u + b u x + a v x + c u x^2 + b v x^2 + a w x^2 + c v x^3 + b w x^3 + c w x^4
```

`ListConvolve` and `ListCorrelate` work on data in any number of dimensions.

This imports image data from a file.

```
In[21]:= g = ReadList["ExampleData/fish.data", Number, RecordLists -> True];
```

Here is the image.

```
In[22]:= Graphics[Raster[g]]
```

Out[22]=

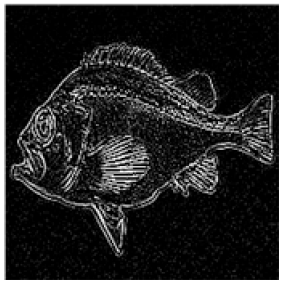
This convolves the data with a two-dimensional kernel.

```
In[23]:= ListConvolve[{{1, 1, 1}, {1, -8, 1}, {1, 1, 1}}, g];
```

This shows the image corresponding to the data.

```
In[24]:= Graphics[Raster[%]]
```

Out[24]=



Cellular Automata

Cellular automata provide a convenient way to represent many kinds of systems in which the values of cells in an array are updated in discrete steps according to a local rule.

`CellularAutomaton[`*rnum*`,`*init*`,`*t*`]`	evolve rule *rnum* from *init* for *t* steps

Generating a cellular automaton evolution.

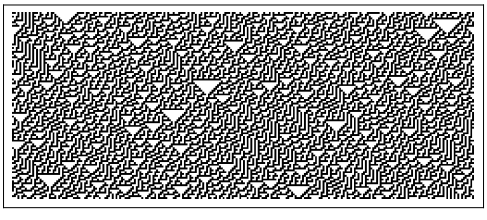
This starts with the list given, then evolves rule 30 for 4 steps.

```
In[1]:= CellularAutomaton[30, {0, 0, 0, 1, 0, 0, 0}, 4]
Out[1]= {{0, 0, 0, 1, 0, 0, 0}, {0, 0, 1, 1, 1, 0, 0},
         {0, 1, 1, 0, 0, 1, 0}, {1, 1, 0, 1, 1, 1, 1}, {0, 0, 0, 1, 0, 0, 0}}
```

This shows 100 steps of rule 30 evolution from random initial conditions.

```
In[2]:= ArrayPlot[CellularAutomaton[30, RandomInteger[1, 250], 100]]
```

Out[2]=



$\{a_1, a_2, \ldots\}$	explicit list of values a_i
$\{\{a_1, a_2, \ldots\}, b\}$	values a_i superimposed on a b background
$\{\{a_1, a_2, \ldots\}, blist\}$	values a_i superimposed on a background of repetitions of $blist$
$\{\{\{\{a_{11}, a_{12}, \ldots\}, \{d_1\}\}, \ldots\}, blist\}$	values a_{ij} at offsets d_i

Ways of specifying initial conditions for one-dimensional cellular automata.

If you give an explicit list of initial values, `CellularAutomaton` will take the elements in this list to correspond to all the cells in the system, arranged cyclically.

The right neighbor of the cell at the end is the cell at the beginning.

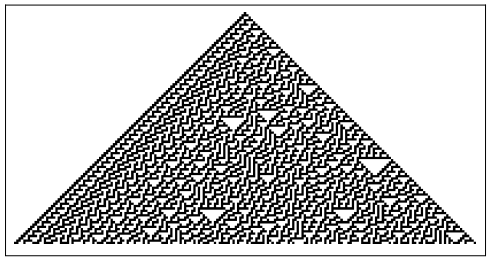
```
In[4]:= CellularAutomaton[30, {1, 0, 0, 0, 0}, 1]
Out[4]= {{1, 0, 0, 0, 0}, {1, 1, 0, 0, 1}}
```

It is often convenient to set up initial conditions in which there is a small "seed" region, superimposed on a constant "background". By default, `CellularAutomaton` automatically fills in enough background to cover the size of the pattern that can be produced in the number of steps of evolution you specify.

This shows rule 30 evolving from an initial condition containing a single black cell.

```
In[5]:= ArrayPlot[CellularAutomaton[30, {{1}, 0}, 100]]
```

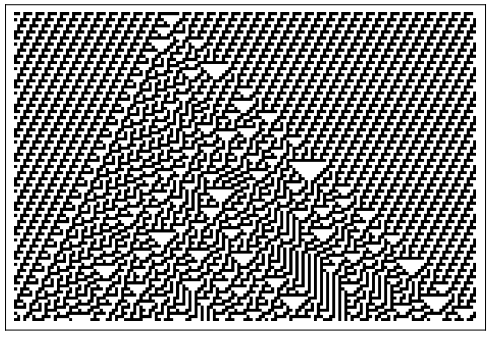
Out[5]=



This shows rule 30 evolving from an initial condition consisting of a {1, 1} seed on a background of repeated {1, 0, 1, 1} blocks.

```
In[6]:= ArrayPlot[CellularAutomaton[30, {{1, 1}, {1, 0, 1, 1}}, 100]]
```

Out[6]=

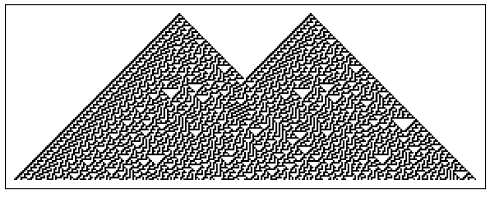


Particularly in studying interactions between structures, you may sometimes want to specify initial conditions for cellular automata in which certain blocks are placed at particular offsets.

This sets up an initial condition with black cells at offsets ± 40.

```
In[7]:= ArrayPlot[CellularAutomaton[30, {{{{1}, {-40}}, {{1}, {40}}}, 0}, 100]]
```

Out[7]=



n	$k = 2$, $r = 1$, elementary rule
$\{n, k\}$	general nearest-neighbor rule with k colors
$\{n, k, r\}$	general rule with k colors and range r
$\{n, \{k, 1\}\}$	k-color nearest-neighbor totalistic rule
$\{n, \{k, 1\}, r\}$	k-color range r totalistic rule
$\{n, \{k, \{wt_1, wt_2, \ldots\}\}, r\}$	rule in which neighbor i is assigned weight wt_i
$\{n, kspec, \{\{off_1\}, \{off_2\}, \ldots, \{off_s\}\}\}$	rule with neighbors at specified offsets
$\{lhs_1 \text{->} rhs_1, lhs_2 \text{->} rhs_2, \ldots\}$	explicit replacements for lists of neighbors
$\{fun, \{\}, rspec\}$	rule obtained by applying function fun to each neighbor list

Specifying rules for one-dimensional cellular automata.

In the simplest cases, a cellular automaton allows k possible values or "colors" for each cell, and has rules that involve up to r neighbors on each side. The digits of the "rule number" n then specify what the color of a new cell should be for each possible configuration of the neighborhood.

This evolves a single neighborhood for 1 step.

```
In[8]:= CellularAutomaton[30, {1, 1, 0}, 1]
Out[8]= {{1, 1, 0}, {1, 0, 0}}
```

Here are the 8 possible neighborhoods for a $k = 2$, $r = 1$ cellular automaton.

```
In[9]:= Tuples[{1, 0}, 3]
Out[9]= {{1, 1, 1}, {1, 1, 0}, {1, 0, 1}, {1, 0, 0}, {0, 1, 1}, {0, 1, 0}, {0, 0, 1}, {0, 0, 0}}
```

This shows the new color of the center cell for each of the 8 neighborhoods.

```
In[10]:= Map[CellularAutomaton[30, #, 1][[2, 2]] &, %]
Out[10]= {0, 0, 0, 1, 1, 1, 1, 0}
```

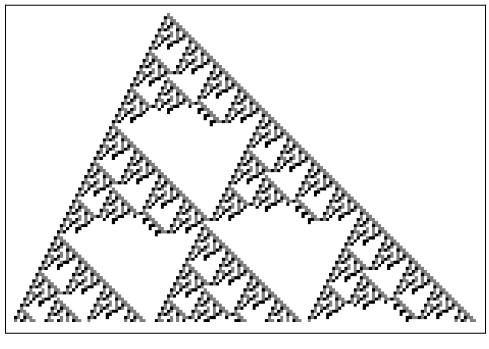
For rule 30, this sequence corresponds to the base-2 digits of the number 30.

```
In[11]:= FromDigits[%, 2]
Out[11]= 30
```

This runs the general $k = 3$, $r = 1$ rule with rule number 921408.

```
In[12]:= ArrayPlot[CellularAutomaton[{921408, 3, 1}, {{1}, 0}, 100]]
```

Out[12]=



For a general cellular automaton rule, each digit of the rule number specifies what color a different possible neighborhood of $2r + 1$ cells should yield. To find out which digit corresponds

to which neighborhood, one effectively treats the cells in a neighborhood as digits in a number. For an $r = 1$ cellular automaton, the number is obtained from the list of elements *neig* in the neighborhood by *neig*`.{`k`^2, `k`, 1}`.

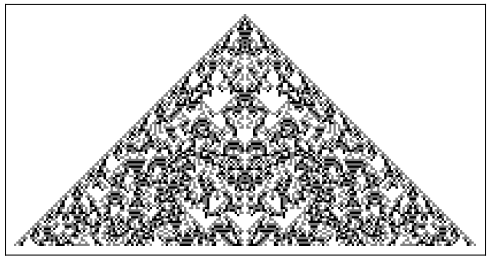
It is sometimes convenient to consider *totalistic* cellular automata, in which the new value of a cell depends only on the total of the values in its neighborhood. One can specify totalistic cellular automata by rule numbers or "codes" in which each digit refers to neighborhoods with a given total value, obtained for example from *neig*`.{1, 1, 1}`.

In general, `CellularAutomaton` allows one to specify rules using any sequence of weights. Another choice sometimes convenient is `{`k`, 1, `k`}`, which yields outer totalistic rules.

This runs the $k = 3$, $r = 1$ totalistic rule with code number 867.

```
In[13]:= ArrayPlot[CellularAutomaton[{867, {3, 1}, 1}, {{1}, 0}, 100]]
```

Out[13]=



Rules with range r involve all cells with offsets $-r$ through $+r$. Sometimes it is convenient to think about rules that involve only cells with specific offsets. You can do this by replacing a single r with a list of offsets.

Any $k = 2$ cellular automaton rule can be thought of as corresponding to a Boolean function. In the simplest case, basic Boolean functions like `And` or `Nor` take two arguments. These are conveniently specified in a cellular automaton rule as being at offsets `{{0}, {1}}`. Note that for compatibility with handling higher-dimensional cellular automata, offsets must always be given in lists, even for one-dimensional cellular automata.

This generates the truth table for 2-cell-neighborhood rule number 7, which turns out to be the Boolean function Nand.

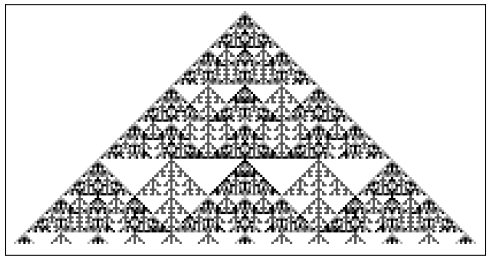
```
In[14]:= Map[CellularAutomaton[{7, 2, {{0}, {1}}}, #, 1][[2, 2]] &,
          {{1, 1}, {1, 0}, {0, 1}, {0, 0}}]
Out[14]= {0, 1, 1, 1}
```

Rule numbers provide a highly compact way to specify cellular automaton rules. But sometimes it is more convenient to specify rules by giving an explicit function that should be applied to each possible neighborhood.

This runs an additive cellular automaton whose rule adds all values in each neighborhood modulo 4.

```
In[15]:= ArrayPlot[CellularAutomaton[{Mod[Apply[Plus, #], 4] &, {}, 1}, {{1}, 0}, 100]]
```

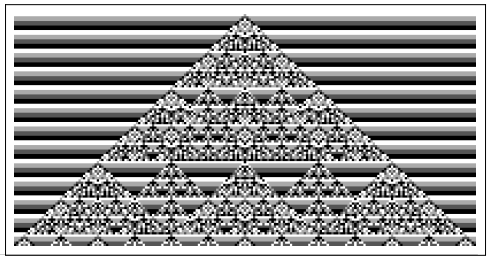
Out[15]=



The function is given the step number as a second argument.

```
In[16]:= ArrayPlot[CellularAutomaton[{Mod[Total[#] + #2, 4] &, {}, 1}, {{1}, 0}, 100]]
```

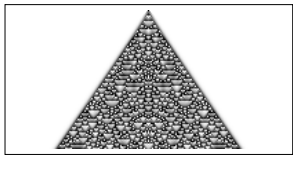
Out[16]=



When you specify rules by functions, the values of cells need not be integers.

```
In[17]:= ArrayPlot[CellularAutomaton[{Mod[1 / 2 Apply[Plus, #], 1] &, {}, 1}, {{1}, 0}, 100]]
```

Out[17]=



They can even be symbolic.

```
In[18]:= Simplify[
          CellularAutomaton[{Mod[Apply[Plus, #], 2] &, {}, 1}, {{a}, 0}, 2], a ∈ Integers]
Out[18]= {{0, 0, a, 0, 0}, {0, Mod[a, 2], Mod[a, 2], Mod[a, 2], 0}, {Mod[a, 2], 0, Mod[a, 2], 0, Mod[a, 2]}}
```

CellularAutomaton[*rnum*, *init*, *t*]	evolve for *t* steps, keeping all steps
CellularAutomaton[*rnum*, *init*, {{*t*}}]	evolve for *t* steps, keeping only the last step
CellularAutomaton[*rnum*, *init*, {*spec*$_t$}]	keep only steps specified by *spec*$_t$
CellularAutomaton[*rnum*, *init*]	evolve rule for one step, giving only the last step

Selecting which steps to keep.

This runs rule 30 for 5 steps, keeping only the last step.

```
In[19]:= CellularAutomaton[30, {{1}, 0}, {{5}}]
Out[19]= {{1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1}}
```

This keeps the last 2 steps.

```
In[20]:= CellularAutomaton[30, {{1}, 0}, {{4, 5}}]
Out[20]= {{0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0}, {1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1}}
```

This gives 1 step.

```
In[21]:= CellularAutomaton[30, {0, 0, 1, 0, 0}]
Out[21]= {0, 1, 1, 1, 0}
```

The step specification *spec*$_t$ works very much like taking elements from a list with `Take`. One difference, though, is that the initial condition for the cellular automaton is considered to be step `0`. Note that any step specification of the form {…} must be enclosed in an additional list.

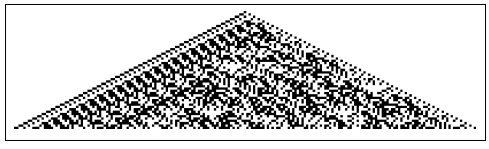
u	steps 0 through *u*
{*u*}	step *u*
{*u*$_1$, *u*$_2$}	steps *u*$_1$ through *u*$_2$
{*u*$_1$, *u*$_2$, *du*}	steps *u*$_1$, *u*$_1$ + *du*, …

Cellular automaton step specifications.

This evolves for 100 steps, but keeps only every other step.

```
In[22]:= ArrayPlot[CellularAutomaton[30, {{1}, 0}, {{0, 100, 2}}]]
```

Out[22]=



`CellularAutomaton[`*rnum*`,`*init*`,`*t*`]`	keep all steps, and all relevant cells
`CellularAutomaton[`*rnum*`,`*init*`,{`$spec_t$`,`$spec_x$`}]`	keep only specified steps and cells

Selecting steps and cells to keep.

Much as you can specify which steps to keep in a cellular automaton evolution, so also you can specify which cells to keep. If you give an initial condition such as $\{a_1, a_2, \ldots\}$, *blist*, then a_i is taken to have offset 0 for the purpose of specifying which cells to keep.

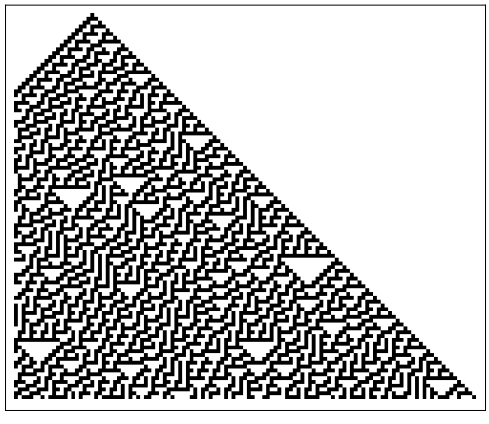
`All`	all cells that can be affected by the specified initial condition
`Automatic`	all cells in the region that differs from the background (default)
0	cell aligned with beginning of *aspec*
x	cells at offsets up to x on the right
$-x$	cells at offsets up to x on the left
$\{x\}$	cell at offset x to the right
$\{-x\}$	cell at offset x to the left
$\{x_1, x_2\}$	cells at offsets x_1 through x_2
$\{x_1, x_2, dx\}$	cells x_1, $x_1 + dx$, …

Cellular automaton cell specifications.

This keeps all steps, but drops cells at offsets more than 20 on the left.

In[23]:=
```
ArrayPlot[CellularAutomaton[30, {{1}, 0}, {100, {-20, 100}}]]
```

Out[23]=



This keeps just the center column of cells.

```
In[24]:= CellularAutomaton[30, {{1}, 0}, {20, {0}}]
Out[24]= {{1}, {1}, {0}, {1}, {1}, {1}, {0}, {0}, {1},
          {1}, {0}, {0}, {0}, {1}, {0}, {1}, {1}, {0}, {0}, {1}, {0}}
```

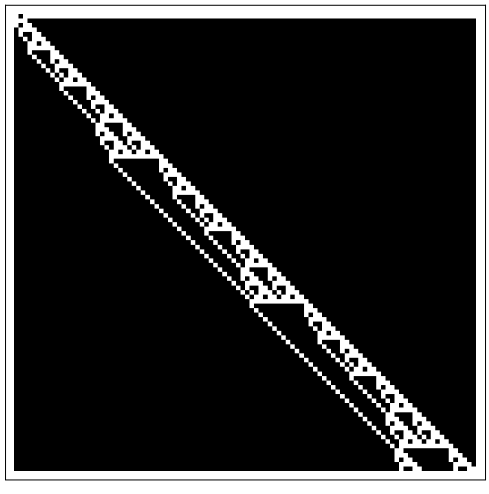
If you give an initial condition such as $\{\{a_1, a_2, \ldots\}, blist\}$, then `CellularAutomaton` will always effectively do the cellular automaton as if there were an infinite number of cells. By using a $spec_x$ such as $\{x_1, x_2\}$ you can tell `CellularAutomaton` to include only cells at specific offsets x_1 through x_2 in its output. `CellularAutomaton` by default includes cells out just far enough that their values never simply stay the same as in the background $blist$.

In general, given a cellular automaton rule with range r, cells out to distance rt on each side could in principle be affected in the evolution of the system. With $spec_x$ being `All`, all these cells are included; with the default setting of `Automatic`, cells whose values effectively stay the same as in $blist$ are trimmed off.

By default, only the parts that are not constant black are kept.

```
In[25]:= ArrayPlot[CellularAutomaton[225, {{1}, 0}, 100]]
```

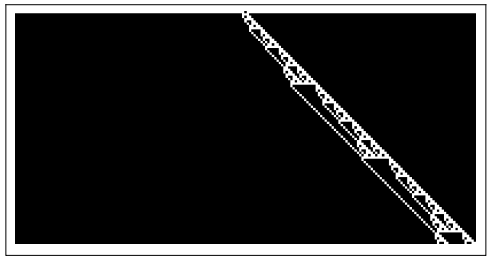
Out[25]=



Using `All` for $spec_x$ includes all cells that could be affected by a cellular automaton with this range.

```
In[26]:= ArrayPlot[CellularAutomaton[225, {{1}, 0}, {100, All}]]
```

Out[26]=



`CellularAutomaton` generalizes quite directly to any number of dimensions. Above two dimensions, however, totalistic and other special types of rules tend to be more useful, since the number of entries in the rule table for a general rule rapidly becomes astronomical.

$\{n, k, \{r_1, r_2, \ldots, r_d\}\}$	d-dimensional rule with $(2r_1+1)\times(2r_2+1)\times\ldots\times(2r_d+1)$ neighborhood
$\{n, \{k, 1\}, \{1, 1\}\}$	two-dimensional 9-neighbor totalistic rule
$\{n, \{k, \{\{0,1,0\}, \{1,1,1\}, \{0,1,0\}\}\}, \{1,1\}\}$	two-dimensional 5-neighbor totalistic rule
$\{n, \{k, \{\{0,k,0\}, \{k,1,k\}, \{0,k,0\}\}\}, \{1,1\}\}$	two-dimensional 5-neighbor outer totalistic rule

Higher-dimensional rule specifications.

This is the rule specification for the two-dimensional 9-neighbor totalistic cellular automaton with code 797.

```
In[27]:= code797 = {797, {2, 1}, {1, 1}};
```

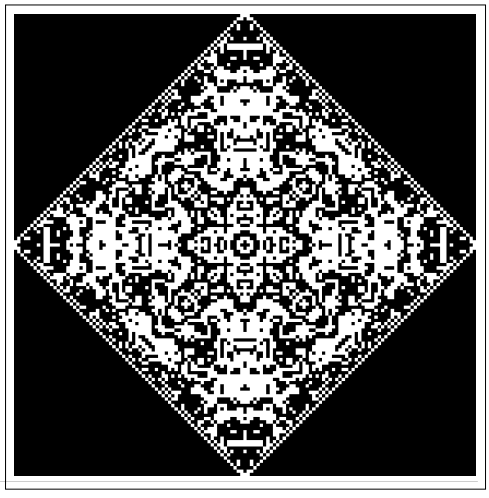
This gives steps 0 and 1 in its evolution.

```
In[28]:= CellularAutomaton[code797, {{{1}}, 0}, 1]
Out[28]= {{{0, 0, 0}, {0, 1, 0}, {0, 0, 0}}, {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}}}
```

This shows step 70 in the evolution.

In[29]:=
```
ArrayPlot[First[CellularAutomaton[code797, {{{1}}, 0},
                        {{70}}]]]
```

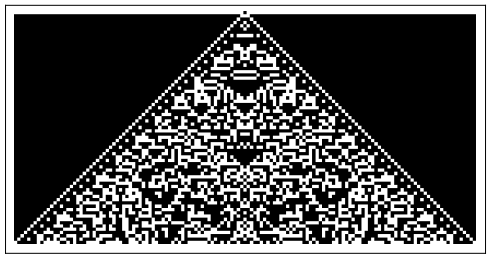
Out[29]=



This shows all steps in a slice along the x axis.

In[30]:=
```
ArrayPlot[Map[First,
  CellularAutomaton[code797, {{{1}}, 0},
                {70, {0}, All}]]]
```

Out[30]=



Mathematical Functions

Naming Conventions

Mathematical functions in *Mathematica* are given names according to definite rules. As with most *Mathematica* functions, the names are usually complete English words, fully spelled out. For a few very common functions, *Mathematica* uses the traditional abbreviations. Thus the modulo function, for example, is `Mod`, not `Modulo`.

Mathematical functions that are usually referred to by a person's name have names in *Mathematica* of the form *PersonSymbol*. Thus, for example, the Legendre polynomials $P_n(x)$ are denoted `LegendreP[`n`, `x`]`. Although this convention does lead to longer function names, it avoids any ambiguity or confusion.

When the standard notation for a mathematical function involves both subscripts and superscripts, the subscripts are given *before* the superscripts in the *Mathematica* form. Thus, for example, the associated Legendre polynomials $P_n^m(x)$ are denoted `LegendreP[`n`, `m`, `x`]`.

Generic and Non-Generic Cases

This gives a result for the integral of x^n that is valid for almost all values of n.

```
In[1]:= Integrate[x^n, x]
```

$$\text{Out[1]}= \frac{x^{1+n}}{1+n}$$

For the special case of x^{-1}, however, the correct result is different.

```
In[2]:= Integrate[x^-1, x]
Out[2]= Log[x]
```

The overall goal of symbolic computation is typically to get formulas that are valid for many possible values of the variables that appear in them. It is however often not practical to try to get formulas that are valid for absolutely every possible value of each variable.

Mathematica always replaces $0/x$ by 0.

```
In[3]:= 0 / x
Out[3]= 0
```

If x is equal to 0, however, then the true result is not 0.

```
In[4]:= 0 / 0
```

Power::infy: Infinite expression $\frac{1}{0}$ encountered. ≫

∞::indet: Indeterminate expression 0 ComplexInfinity encountered. ≫

```
Out[4]= Indeterminate
```

This construct treats both cases, but would be quite unwieldy to use.

```
In[5]:= If[x != 0, 0, Indeterminate]
Out[5]= If[x ≠ 0, 0, Indeterminate]
```

If *Mathematica* did not automatically replace $0/x$ by 0, then few symbolic computations would get very far. But you should realize that the practical necessity of making such replacements can cause misleading results to be obtained when exceptional values of parameters are used.

The basic operations of *Mathematica* are nevertheless carefully set up so that whenever possible the results obtained will be valid for almost all values of each variable.

$\sqrt{x^2}$ is not automatically replaced by x.

```
In[6]:= Sqrt[x^2]
Out[6]= √x²
```

If it were, then the result here would be -2, which is incorrect.

```
In[7]:= % /. x -> -2
Out[7]= 2
```

This makes the assumption that x is a positive real variable, and does the replacement.

```
In[8]:= Simplify[Sqrt[x^2], x > 0]
Out[8]= x
```

Numerical Functions

`IntegerPart[x]`	integer part of x
`FractionalPart[x]`	fractional part of x
`Round[x]`	integer $\langle x \rangle$ closest to x
`Floor[x]`	greatest integer $\lfloor x \rfloor$ not larger than x
`Ceiling[x]`	least integer $\lceil x \rceil$ not smaller than x
`Rationalize[x]`	rational number approximation to x
`Rationalize[x,dx]`	rational approximation within tolerance dx

Functions relating real numbers and integers.

x	`IntegerPart[x]`	`FractionalPart[x]`	`Round[x]`	`Floor[x]`	`Ceiling[x]`
2.4	2	0.4	2	2	3
2.5	2	0.5	2	2	3
2.6	2	0.6	3	2	3
-2.4	-2	-0.4	-2	-3	-2
-2.5	-2	-0.5	-2	-3	-2
-2.6	-2	-0.6	-3	-3	-2

Extracting integer and fractional parts.

`IntegerPart[x]` and `FractionalPart[x]` can be thought of as extracting digits to the left and right of the decimal point. `Round[x]` is often used for forcing numbers that are close to integers to be exactly integers. `Floor[x]` and `Ceiling[x]` often arise in working out how many elements there will be in sequences of numbers with non-integer spacings.

`Sign[x]`	1 for $x > 0$, -1 for $x < 0$
`UnitStep[x]`	1 for $x \geq 0$, 0 for $x < 0$
`Abs[x]`	absolute value $\lvert x \rvert$ of x
`Clip[x]`	x clipped to be between -1 and +1
`Rescale[x,{xmin,xmax}]`	x rescaled to run from 0 to 1
`Max[x1,x2,...]` or `Max[{x1,x2,...},...]`	the maximum of x_1, x_2, ...
`Min[x1,x2,...]` or `Min[{x1,x2,...},...]`	the minimum of x_1, x_2, ...

Numerical functions of real variables.

x+I y	the complex number $x + i\,y$
Re[z]	the real part Re z
Im[z]	the imaginary part Im z
Conjugate[z]	the complex conjugate z^* or $\overline{z}$
Abs[z]	the absolute value $\lvert z \rvert$
Arg[z]	the argument ϕ such that $z = \lvert z \rvert\, e^{i\phi}$

Numerical functions of complex variables.

Piecewise Functions

Boole[*expr*]	give 1 if *expr* is True, and 0 if it is False

Turning conditions into numbers.

Boole[expr] is a basic function that turns True and False into 1 and 0. It is sometimes known as the *characteristic function* or *indicator function*.

This gives the area of a unit disk.

```
In[1]:= Integrate[Boole[x^2 + y^2 <= 1], {x, -1, 1}, {y, -1, 1}]
Out[1]= π
```

Piecewise[{{val_1, $cond_1$}, {val_2, $cond_2$}, …}]	give the first val_i for which $cond_i$ is True
Piecewise[{{val_1, $cond_1$}, …}, val]	give val if all $cond_i$ are False

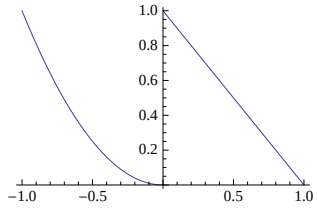
Piecewise functions.

It is often convenient to have functions with different forms in different regions. You can do this using Piecewise.

This plots a piecewise function.

```
In[2]:= Plot[Piecewise[{{x^2, x < 0}, {1 - x, x > 0}}], {x, -1, 1}]
```

Out[2]=

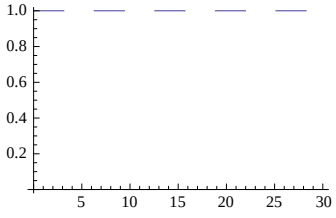


Piecewise functions appear in systems where there is discrete switching between different domains. They are also at the core of many computational methods, including splines and finite elements. Special cases include such functions as `Abs`, `UnitStep`, `Clip`, `Sign`, `Floor` and `Max`. *Mathematica* handles piecewise functions in both symbolic and numerical situations.

This generates a square wave.

```
In[3]:= Plot[UnitStep[Sin[x]], {x, 0, 30}]
```

Out[3]=



Here is the integral of the square wave.

```
In[4]:= Integrate[UnitStep[Sin[x]], {x, 0, 30}]
```

Out[4]= $5\,\pi$

Pseudorandom Numbers

Mathematica has three functions for generating pseudorandom numbers that are distributed uniformly over a range of values.

`RandomInteger[]`	0 or 1 with probability $\frac{1}{2}$
`RandomInteger[{`i_{min}`,`i_{max}`}]`	an integer between i_{min} and i_{max}, inclusive
`RandomInteger[`i_{max}`]`	an integer between 0 and i_{max}, inclusive

`RandomReal[ ]`	a real number between 0 and 1
`RandomReal[{`x_{min}`, `x_{max}`}]`	a real number between x_{min} and x_{max}
`RandomReal[`x_{max}`]`	a real number between 0 and x_{max}
`RandomComplex[]`	a complex number in the unit square
`RandomComplex[{`z_{min}`, `z_{max}`}]`	a complex number in the rectangle defined by z_{min} and z_{max}
`RandomComplex[`z_{max}`]`	a complex number in the rectangle defined by 0 and z_{max}

Pseudorandom number generation.

`RandomReal[`*range*`,`*n*`] , RandomComplex[`*range*`,`*n*`] , RandomInteger[`*range*`,`*n*`]`	a list of *n* pseudorandom numbers from the given range
`RandomReal[`*range*`,{`n_1`,`n_2`,…}] , RandomComplex[`*range*`,{`n_1`,`n_2`,…}] , RandomInteger[`*range*`,{`n_1`,`n_2`,…}]`	an $n_1 \times n_2 \times \ldots$ array of pseudorandom numbers

Generating tables of pseudorandom numbers.

This will give 0 or 1 with equal probability.

```
In[1]:= RandomInteger[]
Out[1]= 1
```

This gives a pseudorandom complex number.

```
In[2]:= RandomComplex[]
Out[2]= 0.2597 + 0.789124 i
```

This gives a list of 10 pseudorandom integers between 0 and 9 (inclusive).

```
In[3]:= RandomInteger[{0, 9}, 10]
Out[3]= {7, 8, 0, 7, 5, 5, 1, 4, 2, 3}
```

This gives a matrix of pseudorandom reals between 0 and 1.

```
In[4]:= RandomReal[1, {3, 3}]
Out[4]= {{0.798457, 0.90866, 0.0501389}, {0.254316, 0.428327, 0.734689}, {0.293283, 0.628561, 0.463396}}
```

`RandomReal` and `RandomComplex` allow you to obtain pseudorandom numbers with any precision.

option name	default value	
WorkingPrecision	MachinePrecision	precision to use for real or complex numbers

Changing the precision for pseudorandom numbers.

Here is a 30-digit pseudorandom real number in the range 0 to 1.

```
In[5]:= RandomReal[WorkingPrecision → 30]
Out[5]= 0.592425435593582531459708475075
```

Here is a list of four 20-digit pseudorandom complex numbers.

```
In[6]:= RandomComplex[{-1 - I, 1 + I}, 4, WorkingPrecision → 20]
Out[6]= {0.90904993322285593741 + 0.12394429399708576389 i,
         -0.32862367115658248044 - 0.99682159655818692192 i,
         0.28641328852741506413 + 0.19292256786947586951 i,
         0.59587180971009053802 + 0.83133753157739276902 i}
```

If you get arrays of pseudorandom numbers repeatedly, you should get a "typical" sequence of numbers, with no particular pattern. There are many ways to use such numbers.

One common way to use pseudorandom numbers is in making numerical tests of hypotheses. For example, if you believe that two symbolic expressions are mathematically equal, you can test this by plugging in "typical" numerical values for symbolic parameters, and then comparing the numerical results. (If you do this, you should be careful about numerical accuracy problems and about functions of complex variables that may not have unique values.)

Here is a symbolic equation.

```
In[7]:= Sqrt[x^2] == Abs[x]
Out[7]= Sqrt[x^2] == Abs[x]
```

Substituting in a random numerical value shows that the equation is not always `True`.

```
In[8]:= % /. x -> RandomComplex[]
Out[8]= False
```

Other common uses of pseudorandom numbers include simulating probabilistic processes, and sampling large spaces of possibilities. The pseudorandom numbers that *Mathematica* generates for a range of numbers are always uniformly distributed over the range you specify.

RandomInteger, RandomReal and RandomComplex are unlike almost any other *Mathematica* functions in that every time you call them, you potentially get a different result. If you use them in a calculation, therefore, you may get different answers on different occasions.

The sequences that you get from RandomInteger, RandomReal and RandomComplex are not in most senses "truly random", although they should be "random enough" for practical purposes. The sequences are in fact produced by applying a definite mathematical algorithm, starting from a particular "seed". If you give the same seed, then you get the same sequence.

When *Mathematica* starts up, it takes the time of day (measured in small fractions of a second) as the seed for the pseudorandom number generator. Two different *Mathematica* sessions will therefore almost always give different sequences of pseudorandom numbers.

If you want to make sure that you always get the same sequence of pseudorandom numbers, you can explicitly give a seed for the pseudorandom generator, using SeedRandom.

SeedRandom[]	reseed the pseudorandom generator, with the time of day
SeedRandom[*s*]	reseed with the integer *s*

Pseudorandom number generator seed.

This reseeds the pseudorandom generator.

```
In[9]:= SeedRandom[143]
```

Here are three pseudorandom numbers.

```
In[10]:= RandomReal[1, {3}]
Out[10]= {0.110762, 0.364563, 0.163681}
```

If you reseed the pseudorandom generator with the same seed, you get the same sequence of pseudorandom numbers.

```
In[11]:= SeedRandom[143]; RandomReal[1, {3}]
Out[11]= {0.110762, 0.364563, 0.163681}
```

Every single time RandomInteger, RandomReal or RandomComplex is called, the internal state of the pseudorandom generator that it uses is changed. This means that subsequent calls to these functions made in subsidiary calculations will have an effect on the numbers returned in your main calculation. To avoid any problems associated with this, you can localize this effect of their use by doing the calculation inside of BlockRandom.

`BlockRandom[`*expr*`]`	evaluates *expr* with the current state of the pseudorandom generators localized

Localizing the effects of using `RandomInteger`, `RandomReal` or `RandomComplex`.

By localizing the calculation inside `BlockRandom`, the internal state of the pseudorandom generator is restored after generating the first list.

```
In[12]:= {BlockRandom[{RandomReal[], RandomReal[]}], {RandomReal[], RandomReal[]}}
Out[12]= {{0.952312, 0.93591}, {0.952312, 0.93591}}
```

Many applications require random numbers from non-uniform distributions. *Mathematica* has many distributions built into the system. You can give a distribution with appropriate parameters instead of a range to `RandomInteger` or `RandomReal`.

`RandomInteger[`*dist*`]` , `RandomReal[`*dist*`]`	a pseudorandom number distributed by the random distribution *dist*
`RandomInteger[`*dist*`,`*n*`]` , `RandomReal[`*dist*`,`*n*`]`	a list of *n* pseudorandom numbers distributed by the random distribution *dist*
`RandomInteger[`*dist*`,{`n_1`,`n_2`,...}]` , `RandomReal[`*dist*`,{`n_1`,`n_2`,...}]`	an $n_1 \times n_2 \times \ldots$ array of pseudorandom numbers distributed by the random distribution *dist*

Generating pseudorandom numbers with non-uniform distributions.

This generates 12 integers distributed by the Poisson distribution with mean 3.

```
In[13]:= RandomInteger[PoissonDistribution[3], 12]
Out[13]= {4, 3, 1, 3, 2, 2, 5, 3, 5, 2, 1, 3}
```

This generates a 4×4 matrix of real numbers using the standard normal distribution.

```
In[14]:= RandomReal[NormalDistribution[], {4, 4}]
Out[14]= {{1.17681, -0.774733, -1.74139, 1.3577}, {-1.261, 0.0408214, 0.989022, 2.80942},
          {-1.27146, 1.63037, 1.98221, 0.403135}, {1.00722, -0.927379, 0.747369, -2.28065}}
```

This generates five high-precision real numbers distributed normally with mean 2 and standard deviation 4.

```
In[15]:= RandomReal[NormalDistribution[2, 4], 5, WorkingPrecision → 32]
Out[15]= {-3.7899344407106290701062146195097,
          -1.1607986070402381009885236231751, 12.042079595098604792470496688453,
          2.3508651879131153670572237267418, 5.028745244941346304530057781817 3}
```

An additional use of pseudorandom numbers is for selecting from a list. `RandomChoice` selects with replacement and `RandomSample` samples without replacement.

`RandomChoice[`*list*`, `*n*`]`	choose *n* items at random from *list*
`RandomChoice[`*list*`,{`n_1`,`n_2`,…}]`	an $n_1 \times n_2 \times \ldots$ array of values chosen randomly from *list*
`RandomSample[`*list*`, `*n*`]`	a sample of size *n* from *list*

Selecting at random.

Choose 10 items at random from the digits 0 through 9.

```
In[16]:= RandomChoice[Range[0, 9], 10]
Out[16]= {8, 8, 3, 5, 3, 5, 0, 8, 4, 1}
```

Chances are very high that at least one of the choices was repeated in the output. That is because when an element is chosen, it is immediately replaced. On the other hand, if you want to select from an actual set of elements, there should be no replacement.

Sample 10 items at random from the digits 0 through 9 without replacement. The result is a random permutation of the digits.

```
In[17]:= RandomSample[Range[0, 9], 10]
Out[17]= {7, 9, 2, 5, 3, 4, 1, 8, 0, 6}
```

Sample 10 items from a set having different frequencies for each digit.

```
In[18]:= RandomSample[{0, 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6,
           7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9}, 10]
Out[18]= {6, 6, 7, 8, 9, 7, 2, 2, 9, 9}
```

Integer and Number Theoretic Functions

`Mod[`k`,`n`]`	k modulo n (remainder from dividing k by n)
`Quotient[`m`,`n`]`	the quotient of m and n (truncation of m / n)
`QuotientRemainder[`m`,`n`]`	a list of the quotient and the remainder
`Divisible[`m`,`n`]`	test whether m is divisible by n
`CoprimeQ[`n_1`,`n_2`,…]`	test whether the n_i are pairwise relatively prime
`GCD[`n_1`,`n_2`,…]`	the greatest common divisor of n_1, n_2, …

LCM[n_1,n_2,...]	the least common multiple of n_1, n_2, ...
KroneckerDelta[n_1,n_2,...]	the Kronecker delta $\delta_{n_1 n_2 \ldots}$ equal to 1 if all the n_i are equal, and 0 otherwise
IntegerDigits[n,b]	the digits of n in base b
IntegerExponent[n,b]	the highest power of b that divides n

Some integer functions.

The remainder on dividing 17 by 3.

```
In[1]:= Mod[17, 3]
Out[1]= 2
```

The integer part of 17/3.

```
In[2]:= Quotient[17, 3]
Out[2]= 5
```

Mod also works with real numbers.

```
In[3]:= Mod[5.6, 1.2]
Out[3]= 0.8
```

The result from Mod always has the same sign as the second argument.

```
In[4]:= Mod[-5.6, 1.2]
Out[4]= 0.4
```

For any integers a and b, it is always true that $b * \texttt{Quotient}[a, b] + \texttt{Mod}[a, b]$ is equal to a.

Mod[k,n]	result in the range 0 to $n - 1$
Mod[k,n,1]	result in the range 1 to n
Mod[k,n,$-n/2$]	result in the range $\lceil -n/2 \rceil$ to $\lfloor +n/2 \rfloor$
Mod[k,n,d]	result in the range d to $d + n - 1$

Integer remainders with offsets.

Particularly when you are using Mod to get indices for parts of objects, you will often find it convenient to specify an offset.

This effectively extracts the 18th part of the list, with the list treated cyclically.

```
In[5]:= Part[{a, b, c}, Mod[18, 3, 1]]
Out[5]= c
```

The *greatest common divisor* function `GCD[`n_1`, `n_2`, ...]` gives the largest integer that divides all the n_i exactly. When you enter a ratio of two integers, *Mathematica* effectively uses `GCD` to cancel out common factors and give a rational number in lowest terms.

The *least common multiple* function `LCM[`n_1`, `n_2`, ...]` gives the smallest integer that contains all the factors of each of the n_i.

The largest integer that divides both 24 and 15 is 3.

```
In[6]:= GCD[24, 15]
Out[6]= 3
```

The *Kronecker delta* function `KroneckerDelta[`n_1`, `n_2`, ...]` is equal to 1 if all the n_i are equal, and is 0 otherwise. $\delta_{n_1 n_2 \ldots}$ can be thought of as a totally symmetric tensor.

This gives a totally symmetric tensor of rank 3.

```
In[7]:= Array[KroneckerDelta, {3, 3, 3}]
Out[7]= {{{1, 0, 0}, {0, 0, 0}, {0, 0, 0}},
         {{0, 0, 0}, {0, 1, 0}, {0, 0, 0}}, {{0, 0, 0}, {0, 0, 0}, {0, 0, 1}}}
```

`FactorInteger[`n`]`	a list of the prime factors of n, and their exponents
`Divisors[`n`]`	a list of the integers that divide n
`Prime[`k`]`	the k^{th} prime number
`PrimePi[`x`]`	the number of primes less than or equal to x
`PrimeQ[`n`]`	give `True` if n is a prime, and `False` otherwise
`PrimeNu[`n`]`	the number of distinct primes $\nu(n)$ in n
`PrimeOmega[`n`]`	the number of prime factors counting multiplicities $\Omega(n)$ in n.
`LiouvilleLambda[`n`]`	the Liouville function $\lambda(n)$
`MangoldtLambda[`n`]`	the von Mandgoldt function $\Lambda(n)$
`FactorInteger[`n`,GaussianIntegers->True]`	a list of the Gaussian prime factors of the Gaussian integer n, and their exponents

PrimeQ[*n*,GaussianIntegers->True]

give True if *n* is a Gaussian prime, and False otherwise

Integer factoring and related functions.

This gives the factors of 24 as 2^3, 3^1. The first element in each list is the factor; the second is its exponent.

```
In[8]:= FactorInteger[24]
Out[8]= {{2, 3}, {3, 1}}
```

Here are the factors of a larger integer.

```
In[9]:= FactorInteger[111 111 111 111 111 111]
Out[9]= {{3, 2}, {7, 1}, {11, 1}, {13, 1}, {19, 1}, {37, 1}, {52 579, 1}, {333 667, 1}}
```

You should realize that according to current mathematical thinking, integer factoring is a fundamentally difficult computational problem. As a result, you can easily type in an integer that *Mathematica* will not be able to factor in anything short of an astronomical length of time. But as long as the integers you give are less than about 50 digits long, FactorInteger should have no trouble. And in special cases it will be able to deal with much longer integers.

Here is a rather special long integer.

```
In[10]:= 30 !
Out[10]= 265 252 859 812 191 058 636 308 480 000 000
```

Mathematica can easily factor this special integer.

```
In[11]:= FactorInteger[%]
Out[11]= {{2, 26}, {3, 14}, {5, 7}, {7, 4}, {11, 2}, {13, 2}, {17, 1}, {19, 1}, {23, 1}, {29, 1}}
```

Although *Mathematica* may not be able to factor a large integer, it can often still test whether or not the integer is a prime. In addition, *Mathematica* has a fast way of finding the k^{th} prime number.

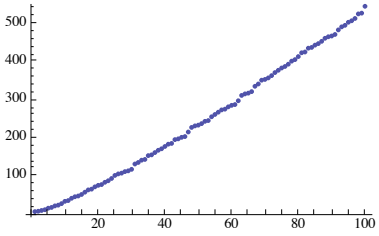
It is often much faster to test whether a number is prime than to factor it.

```
In[12]:= PrimeQ[234 242 423]
Out[12]= False
```

Here is a plot of the first 100 primes.

```
In[13]:= ListPlot[Table[Prime[n], {n, 100}]]
```

Out[13]=



This is the millionth prime.

```
In[14]:= Prime[1 000 000]
Out[14]= 15 485 863
```

Particularly in number theory, it is often more important to know the distribution of primes than their actual values. The function `PrimePi[x]` gives the number of primes $\pi(x)$ that are less than or equal to x.

This gives the number of primes less than a billion.

```
In[15]:= PrimePi[10^9]
Out[15]= 50 847 534
```

`PrimeNu` gives the number of distinct primes.

```
In[16]:= PrimeNu[10^9]
Out[16]= 2
```

`PrimeOmega` gives the number of prime factors counting multiplicities $\Omega(n)$ in n.

```
In[17]:= PrimeOmega[10^9]
Out[17]= 18
```

Liouville's function gives $(-1)^k$ where k is the number of prime factors counting multiplicity.

```
In[18]:= {LiouvilleLambda[3^5], LiouvilleLambda[2 × 3^5]}
Out[18]= {-1, 1}
```

The Mangoldt function returns the log of prime power base or zero when composite.

```
In[19]:= {MangoldtLambda[3^5], MangoldtLambda[2 × 3^5]}
Out[19]= {Log[3], 0}
```

By default, `FactorInteger` allows only real integers. But with the option setting `GaussianIntegers -> True`, it also handles *Gaussian integers*, which are complex numbers with integer real and imaginary parts. Just as it is possible to factor uniquely in terms of real primes, it is also possible to factor uniquely in terms of Gaussian primes. There is nevertheless some potential ambiguity in the choice of Gaussian primes. In *Mathematica*, they are always chosen to have positive real parts, and non-negative imaginary parts, except for a possible initial factor of -1 or $\pm i$.

Over the Gaussian integers, 2 can be factored as $(-i)(1+i)^2$.

```
In[20]:= FactorInteger[2, GaussianIntegers -> True]
Out[20]= {{-i, 1}, {1 + i, 2}}
```

Here are the factors of a Gaussian integer.

```
In[21]:= FactorInteger[111 + 78 I, GaussianIntegers -> True]
Out[21]= {{2 + i, 1}, {3, 1}, {20 + 3 i, 1}}
```

`PowerMod[a,b,n]`	the power a^b modulo n
`DirichletCharacter[k,j,n]`	the Dirichlet character $\chi_{\{k,j\}}(n)$
`EulerPhi[n]`	the Euler totient function $\phi(n)$
`MoebiusMu[n]`	the Möbius function $\mu(n)$
`DivisorSigma[k,n]`	the divisor function $\sigma_k(n)$
`DivisorSum[n,form]`	the sum of $form[i]$ for all i that divide n
`DivisorSum[n,form,cond]`	the sum for only those divisors for which $cond[i]$ gives `True`
`JacobiSymbol[n,m]`	the Jacobi symbol $\left(\frac{n}{m}\right)$
`ExtendedGCD[n1,n2,...]`	the extended GCD of n_1, n_2, ...
`MultiplicativeOrder[k,n]`	the multiplicative order of k modulo n
`MultiplicativeOrder[k,n,{r1,r2,...}]`	the generalized multiplicative order with residues r_i
`CarmichaelLambda[n]`	the Carmichael function $\lambda(n)$
`PrimitiveRoot[n]`	a primitive root of n

Some functions from number theory.

The *modular power function* PowerMod[a, b, n] gives exactly the same results as Mod[a^b, n] for $b > 0$. PowerMod is much more efficient, however, because it avoids generating the full form of a^b.

You can use PowerMod not only to find positive modular powers, but also to find *modular inverses*. For negative b, PowerMod[a, b, n] gives, if possible, an integer k such that $k\,a^{-b} \equiv 1 \bmod n$. (Whenever such an integer exists, it is guaranteed to be unique modulo n.) If no such integer k exists, *Mathematica* leaves PowerMod unevaluated.

PowerMod is equivalent to using Power, then Mod, but is much more efficient.

```
In[22]:= PowerMod[2, 13451, 3]
Out[22]= 2
```

This gives the modular inverse of 3 modulo 7.

```
In[23]:= PowerMod[3, -1, 7]
Out[23]= 5
```

Multiplying the inverse by 3 modulo 7 gives 1, as expected.

```
In[24]:= Mod[3 %, 7]
Out[24]= 1
```

This finds the smallest non-negative integer x so that x^2 is equal to 3 mod 11.

```
In[25]:= PowerMod[3, 1 / 2, 11]
Out[25]= 5
```

This verifies the result.

```
In[26]:= Mod[5^2, 11]
Out[26]= 3
```

This returns all integers less than 11 which satisfy the relation.

```
In[27]:= PowerModList[3, 1 / 2, 11]
Out[27]= {5, 6}
```

If d does not have a square root modulo n, `PowerMod[d, n]` will remain unevaluated and `PowerModList` will return an empty list.

```
In[28]:= PowerMod[3, 1 / 2, 5]

PowerMod::root: The equation x^2 = 3 (mod 5) has no integer solutions.

Out[28]= PowerMod[3, 1/2, 5]

In[29]:= PowerModList[3, 1 / 2, 5]
Out[29]= {}
```

This checks that 3 is not a square modulo 5.

```
In[30]:= Mod[{0, 1, 2, 3, 4}^2, 5]
Out[30]= {0, 1, 4, 4, 1}
```

Even for a large modulus, the square root can be computed fairly quickly.

```
In[31]:= PowerMod[2, 1 / 2, 10^64 + 57] // Timing
Out[31]= {0.01, 876 504 467 496 681 643 735 926 111 996 546 100 401 033 611 976 777 074 909 122 865}
```

`PowerMod[d, n]` also works for composite n.

```
In[32]:= PowerMod[3, 1 / 2, 11^3]
Out[32]= 578
```

There are $\phi(k)$ distinct Dirichlet characters for a given modulus k, as labeled by the index j. Different conventions can give different orderings for the possible characters.

`DirichletCharacter` works for very large numbers.

```
In[33]:= DirichletCharacter[20!, 300, 23]
Out[33]= e^(i π/9)
```

The *Euler totient function* $\phi(n)$ gives the number of integers less than n that are relatively prime to n. An important relation (Fermat's little theorem) is that $a^{\phi(n)} \equiv 1 \bmod n$ for all a relatively prime to n.

The *Möbius function* $\mu(n)$ is defined to be $(-1)^k$ if n is a product of k distinct primes, and 0 if n contains a squared factor (other than 1). An important relation is the Möbius inversion formula, which states that if $g(n) = \sum_{d|n} f(d)$ for all n, then $f(n) = \sum_{d|n} \mu(d)\, g(n/d)$, where the sums are over all positive integers d that divide n.

The *divisor function* $\sigma_k(n)$ is the sum of the k^{th} powers of the divisors of n. The function $\sigma_0(n)$ gives the total number of divisors of n, and is variously denoted $d(n)$, $\nu(n)$ and $\tau(n)$. The function $\sigma_1(n)$, equal to the sum of the divisors of n, is often denoted $\sigma(n)$.

For prime n, $\phi(n) = n - 1$.

```
In[34]:= EulerPhi[17]
Out[34]= 16
```

The result is 1, as guaranteed by Fermat's little theorem.

```
In[35]:= PowerMod[3, %, 17]
Out[35]= 1
```

This gives a list of all the divisors of 24.

```
In[36]:= Divisors[24]
Out[36]= {1, 2, 3, 4, 6, 8, 12, 24}
```

$\sigma_0(n)$ gives the total number of distinct divisors of 24.

```
In[37]:= DivisorSigma[0, 24]
Out[37]= 8
```

The function `DivisorSum[`n`, `*form*`]` represents the sum of *form*`[`i`]` for all i that divide n. `DivisorSum[`n`, `*form*`, `*cond*`]` includes only those divisors for which *cond*`[`i`]` gives `True`.

This gives a list of sums for the divisors of five positive integers.

```
In[38]:= Table[DivisorSum[n, # &], {n, 5}]
Out[38]= {1, 3, 4, 7, 6}
```

This imposes the condition that the value of each divisor i must be less than 6.

```
In[39]:= Table[DivisorSum[n, # &, # < 6 &], {n, 11, 15}]
Out[39]= {1, 10, 1, 3, 9}
```

The *Jacobi symbol* `JacobiSymbol[`n`, `m`]` reduces to the *Legendre symbol* $\left(\frac{n}{m}\right)$ when m is an odd prime. The Legendre symbol is equal to zero if n is divisible by m; otherwise it is equal to 1 if n is a quadratic residue modulo the prime m, and to -1 if it is not. An integer n relatively prime to m

is said to be a quadratic residue modulo m if there exists an integer k such that $k^2 \equiv n \bmod m$. The full Jacobi symbol is a product of the Legendre symbols $\left(\frac{n}{p_i}\right)$ for each of the prime factors p_i such that $m = \prod_i p_i$.

The *extended GCD* `ExtendedGCD[n1, n2, ...]` gives a list $\{g, \{r_1, r_2, \ldots\}\}$ where g is the greatest common divisor of the n_i, and the r_i are integers such that $g = r_1 n_1 + r_2 n_2 + \ldots$. The extended GCD is important in finding integer solutions to linear Diophantine equations.

The first number in the list is the GCD of 105 and 196.

```
In[40]:= ExtendedGCD[105, 196]
Out[40]= {7, {-13, 7}}
```

The second pair of numbers satisfies $g = r\,m + s\,n$.

```
In[41]:= -13 105 + 7 × 196
Out[41]= 7
```

The *multiplicative order function* `MultiplicativeOrder[k, n]` gives the smallest integer m such that $k^m \equiv 1 \bmod n$. Then m is known as the *order* of k modulo n. The notation $\mathrm{ord}_n(k)$ is occasionally used.

The *generalized multiplicative order function* `MultiplicativeOrder[k, n, {r1, r2, ...}]` gives the smallest integer m such that $k^m \equiv r_i \bmod n$ for some i. `MultiplicativeOrder[k, n, {-1, 1}]` is sometimes known as the *suborder function* of k modulo n, denoted $\mathrm{sord}_n(k)$. `MultiplicativeOrder[k, n, {a}]` is sometimes known as the discrete log of a with respect to the base k modulo n.

The *Carmichael function* or *least universal exponent* $\lambda(n)$ gives the smallest integer m such that $k^m \equiv 1 \bmod n$ for all integers k relatively prime to n.

`ContinuedFraction[x,n]`	generate the first n terms in the continued fraction representation of x
`FromContinuedFraction[list]`	reconstruct a number from its continued fraction representation
`Rationalize[x,dx]`	find a rational approximation to x with tolerance dx

Continued fractions.

This generates the first 10 terms in the continued fraction representation for π.

```
In[42]:= ContinuedFraction[Pi, 10]
Out[42]= {3, 7, 15, 1, 292, 1, 1, 1, 2, 1}
```

This reconstructs the number represented by the list of continued fraction terms.

```
In[43]:= FromContinuedFraction[%]
```

$$\text{Out[43]}= \frac{1\,146\,408}{364\,913}$$

The result is close to π.

```
In[44]:= N[%]
Out[44]= 3.14159
```

This gives directly a rational approximation to π.

```
In[45]:= Rationalize[Pi, 1 / 1000]
```

$$\text{Out[45]}= \frac{201}{64}$$

Continued fractions appear in many number theoretic settings. Rational numbers have terminating continued fraction representations. Quadratic irrational numbers have continued fraction representations that become repetitive.

`ContinuedFraction[`x`]`	the complete continued fraction representation for a rational or quadratic irrational number
`QuadraticIrrationalQ[`x`]`	test whether x is a quadratic irrational
`RealDigits[`x`]`	the complete digit sequence for a rational number
`RealDigits[`x`,`b`]`	the complete digit sequence in base b

Complete representations for numbers.

The continued fraction representation of $\sqrt{79}$ starts with the term 8, then involves a sequence of terms that repeat forever.

```
In[46]:= ContinuedFraction[Sqrt[79]]
Out[46]= {8, {1, 7, 1, 16}}
```

This reconstructs $\sqrt{79}$ from its continued fraction representation.

```
In[47]:= FromContinuedFraction[%]
```

$$\text{Out[47]=}\ \sqrt{79}$$

This number is a quadratic irrational.

```
In[48]:= QuadraticIrrationalQ[ Sqrt[79]]
Out[48]= True
```

This shows the recurring sequence of decimal digits in 3/7.

```
In[49]:= RealDigits[3 / 7]
Out[49]= {{{4, 2, 8, 5, 7, 1}}, 0}
```

`FromDigits` reconstructs the original number.

```
In[50]:= FromDigits[%]
```

$$\text{Out[50]=}\ \frac{3}{7}$$

Continued fraction convergents are often used to approximate irrational numbers by rational ones. Those approximations alternate from above and below, and converge exponentially in the number of terms. Furthermore, a convergent p/q of a simple continued fraction is better than any other rational approximation with denominator less than or equal to q.

`Convergents[`x`]`	give a list of rational approximations of x
`Convergents[`x`,`n`]`	give only the first n approximations

Continued fraction convergents.

This gives a list of rational approximations of 101/9801, derived from its continued fraction expansion.

```
In[51]:= Convergents[ 101 / 9801 ]
```

$$\text{Out[51]=}\ \left\{0,\ \frac{1}{97},\ \frac{25}{2426},\ \frac{101}{9801}\right\}$$

This lists only the first 10 convergents.

```
In[52]:= Convergents[ 10 201 / 96 059 601, 10 ]
```

$$\text{Out[52]=}\ \left\{0,\ \frac{1}{9416},\ \frac{1}{9417},\ \frac{3}{28\,250},\ \frac{16}{150\,667},\ \frac{19}{178\,917},\ \frac{92}{866\,335},\ \frac{203}{1\,911\,587},\ \frac{498}{4\,689\,509},\ \frac{701}{6\,601\,096}\right\}$$

This lists successive rational approximations to π, until the numerical precision is exhausted.

```
In[53]:= Convergents[N[Pi]]
```

$$\text{Out[53]= } \left\{3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103\,993}{33\,102}, \frac{104\,348}{33\,215}, \frac{208\,341}{66\,317}, \frac{312\,689}{99\,532}, \frac{833\,719}{265\,381}, \frac{1\,146\,408}{364\,913}, \frac{4\,272\,943}{1\,360\,120}, \frac{5\,419\,351}{1\,725\,033}, \frac{80\,143\,857}{25\,510\,582}\right\}$$

With an exact irrational number, you have to explicitly ask for a certain number of terms.

```
In[54]:= Convergents[Pi, 10]
```

$$\text{Out[54]= } \left\{3, \frac{22}{7}, \frac{333}{106}, \frac{355}{113}, \frac{103\,993}{33\,102}, \frac{104\,348}{33\,215}, \frac{208\,341}{66\,317}, \frac{312\,689}{99\,532}, \frac{833\,719}{265\,381}, \frac{1\,146\,408}{364\,913}\right\}$$

`LatticeReduce[{`v_1 v_2`,...}]`	a reduced lattice basis for the set of integer vectors v_i
`HermiteDecomposition[{`v_1`,`v_2`,...}]`	the echelon form for the set of integer vectors v_i

Functions for integer lattices.

The lattice reduction function `LatticeReduce[{`v_1`, `v_2`, ...}]` is used in several kinds of modern algorithms. The basic idea is to think of the vectors v_k of integers as defining a mathematical *lattice*. Any vector representing a point in the lattice can be written as a linear combination of the form $\sum c_k v_k$, where the c_k are integers. For a particular lattice, there are many possible choices of the "basis vectors" v_k. What `LatticeReduce` does is to find a reduced set of basis vectors $\overline{v}_k$ for the lattice, with certain special properties.

Three unit vectors along the three coordinate axes already form a reduced basis.

```
In[55]:= LatticeReduce[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}]
Out[55]= {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}
```

This gives the reduced basis for a lattice in four-dimensional space specified by three vectors.

```
In[56]:= l = LatticeReduce[{{1, 0, 0, 12 345}, {0, 1, 0, 12 435}, {0, 0, 1, 12 354}}]
Out[56]= {{-1, 0, 1, 9}, {9, 1, -10, 0}, {85, -143, 59, 6}}
```

Notice that in the last example, `LatticeReduce` replaces vectors that are nearly parallel by vectors that are more perpendicular. In the process, it finds some quite short basis vectors.

For a matrix m, HermiteDecomposition gives matrices u and r such that u is unimodular, $u.m = r$, and r is in reduced row echelon form. In contrast to RowReduce, pivots may be larger than 1 because there are no fractions in the ring of integers. Entries above a pivot are minimized by subtracting appropriate multiples of the pivot row.

In this case, the original matrix is recovered because it was in row echelon form.

```
In[57]:= {u, r} = HermiteDecomposition[l]
Out[57]= {{{1371, 143, 1}, {1381, 144, 1}, {1372, 143, 1}},
          {{1, 0, 0, 12 345}, {0, 1, 0, 12 435}, {0, 0, 1, 12 354}}}
```

This satisfies the required identities.

```
In[58]:= {Abs[Det[u]] == 1, u.l == r}
Out[58]= {True, True}
```

Here the second matrix has some pivots larger than 1, and nonzero entries over pivots.

```
In[59]:= HermiteDecomposition[{{-2, 1, 1}, {5, 9, 4}, {-4, 2, -11}}]
Out[59]= {{{2, 1, 0}, {3, 2, 1}, {2, 0, -1}}, {{1, 11, 6}, {0, 23, 0}, {0, 0, 13}}}
```

DigitCount[n,b,d] — the number of d digits in the base-b representation of n

Digit count function.

Here are the digits in the base-2 representation of the number 77.

```
In[60]:= IntegerDigits[77, 2]
Out[60]= {1, 0, 0, 1, 1, 0, 1}
```

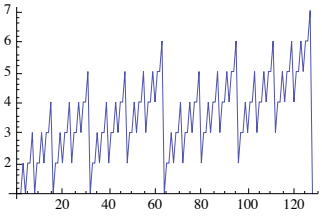
This directly computes the number of ones in the base-2 representation.

```
In[61]:= DigitCount[77, 2, 1]
Out[61]= 4
```

The plot of the digit count function is self-similar.

```
In[62]:= ListLinePlot[Table[DigitCount[n, 2, 1], {n, 128}]]
```

Out[62]=



`BitAnd[`n_1`,`n_2`,...]`	bitwise AND of the integers n_i
`BitOr[`n_1`,`n_2`,...]`	bitwise OR of the integers n_i
`BitXor[`n_1`,`n_2`,...]`	bitwise XOR of the integers n_i
`BitNot[`n`]`	bitwise NOT of the integer n
`BitLength[`n`]`	number of binary bits in the integer n
`BitSet[`n`,`k`]`	set bit k to 1 in the integer n
`BitGet[`n`,`k`]`	get bit k from the integer n
`BitClear[`n`,`k`]`	set bit k to 0 in the integer n
`BitShiftLeft[`n`,`k`]`	shift the integer n to the left by k bits, padding with zeros
`BitShiftRight[`n`,`k`]`	shift to the right, dropping the last k bits

Bitwise operations.

Bitwise operations act on integers represented as binary bits. `BitAnd[`n_1`, `n_2`, ...]` yields the integer whose binary bit representation has ones at positions where the binary bit representations of all of the n_i have ones. `BitOr[`n_1`, `n_2`, ...]` yields the integer with ones at positions where any of the n_i have ones. `BitXor[`n_1`, `n_2`]` yields the integer with ones at positions where n_1 or n_2 but not both have ones. `BitXor[`n_1`, `n_2`, ...]` has ones where an odd number of the n_i have ones.

This finds the bitwise AND of the numbers 23 and 29 entered in base 2.

```
In[63]:= BaseForm[BitAnd[2^^10111, 2^^11101], 2]
Out[63]//BaseForm= 10101_2
```

Bitwise operations are used in various combinatorial algorithms. They are also commonly used in manipulating bitfields in low-level computer languages. In such languages, however, integers normally have a limited number of digits, typically a multiple of 8. Bitwise operations in *Mathematica* in effect allow integers to have an unlimited number of digits. When an integer is negative, it is taken to be represented in two's complement form, with an infinite sequence of ones on the left. This allows `BitNot[`n`]` to be equivalent simply to $-1-n$.

`SquareFreeQ[`n`]`	give `True` if n does not contain a squared factor, `False` otherwise

Testing for a squared factor.

`SquareFreeQ[n]` checks to see if n has a square prime factor. This is done by computing `MoebiusMu[n]` and seeing if the result is zero; if it is, then n is not squarefree, otherwise it is. Computing `MoebiusMu[n]` involves finding the smallest prime factor q of n. If n has a small prime factor (less than or equal to 1223), this is very fast. Otherwise, `FactorInteger` is used to find q.

This product of primes contains no squared factors.

```
In[64]:= SquareFreeQ[2 × 3 × 5 × 7]
Out[64]= True
```

The square number 4 divides 60.

```
In[65]:= SquareFreeQ[60]
Out[65]= False
```

`SquareFreeQ` can handle large integers.

```
In[66]:= SquareFreeQ[2^101 - 1]
Out[66]= True
```

`NextPrime[n]`	give the smallest prime larger than n
`RandomPrime[{min,max}]`	return a random prime number between min and max
`RandomPrime[max]`	return a random prime number less than or equal to max
`RandomPrime[{min,max},n]`	return n random prime numbers between min and max
`RandomPrime[max,n]`	return n random prime numbers less than or equal to max

Finding prime numbers.

`NextPrime[n]` finds the smallest prime p such that $p > n$. For n less than 20 digits, the algorithm does a direct search using `PrimeQ` on the odd numbers greater than n. For n with more than 20 digits, the algorithm builds a small sieve and first checks to see whether the candidate prime is divisible by a small prime before using `PrimeQ`. This seems to be slightly faster than a direct search.

This gives the next prime after 10.

```
In[67]:= NextPrime[10]
Out[67]= 11
```

Even for large numbers, the next prime can be computed rather quickly.

```
In[68]:= NextPrime[10^100] // Timing
Out[68]= {0.06,
 10 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 267}
```

This gives the largest prime less than 34.

```
In[69]:= -NextPrime[-34]
Out[69]= 31
```

For `RandomPrime[{min, max}]` and `RandomPrime[max]`, a random prime p is obtained by randomly selecting from a prime lookup table if max is small and by a random search of integers in the range if max is large. If no prime exists in the specified range, the input is returned unevaluated with an error message.

Here is a random prime between 10 and 100.

```
In[70]:= RandomPrime[{10, 100}]
Out[70]= 79
```

`PrimePowerQ[n]`	determine whether n is a positive integer power of a rational prime

Testing for involving prime powers.

The algorithm for `PrimePowerQ` involves first computing the least prime factor p of n and then attempting division by n until either 1 is obtained, in which case n is a prime power, or until division is no longer possible, in which case n is not a prime power.

Here is a number that is a power of a single prime.

```
In[71]:= PrimePowerQ[12 167]
Out[71]= True
```

Over the `GaussianIntegers` this is a prime power.

```
In[72]:= PrimePowerQ[(1 + I)^3]
Out[72]= True
```

ChineseRemainder[$list_1$, $list_2$]	give the smallest non-negative integer r with Mod[r, $list_2$] == $list_1$

Solving simultaneous congruences.

The Chinese remainder theorem states that a certain class of simultaneous congruences always has a solution. ChineseRemainder[$list_1$, $list_2$] finds the smallest non-negative integer r such that Mod[r, $list_2$] is $list_1$. The solution is unique modulo the least common multiple of the elements of $list_2$.

This means that $244 \equiv 0 \bmod 4$, $244 \equiv 1 \bmod 9$, and $244 \equiv 2 \bmod 121$.

```
In[73]:= ChineseRemainder[{0, 1, 2}, {4, 9, 121}]
Out[73]= 244
```

This confirms the result.

```
In[74]:= Mod[244, {4, 9, 121}]
Out[74]= {0, 1, 2}
```

Longer lists are still quite fast.

```
In[75]:= ChineseRemainder[Range[20], Prime[Range[20]]]
Out[75]= 169 991 099 649 125 127 278 835 143
```

PrimitiveRoot[n]	give a primitive root of n, where n is a prime power or twice a prime power

Computing primitive roots.

PrimitiveRoot[n] returns a generator for the group of numbers relatively prime to n under multiplication $\bmod n$. This has a generator if and only if n is 2, 4, a power of an odd prime, or twice a power of an odd prime. If n is a prime or prime power, the least positive primitive root will be returned.

Here is a primitive root of 5.

```
In[76]:= PrimitiveRoot[5]
Out[76]= 2
```

This confirms that it does generate the group.

```
In[77]:= Sort[Mod[2^Range[4], 5]]
Out[77]= {1, 2, 3, 4}
```

Here is a primitive root of a prime power.

```
In[78]:= PrimitiveRoot[1093^3]
Out[78]= 5
```

Here is a primitive root of twice a prime power.

```
In[79]:= PrimitiveRoot[2 5^5]
Out[79]= 3127
```

If the argument is composite and not a prime power or twice a prime power, the function does not evaluate.

```
In[80]:= PrimitiveRoot[11 × 13]
Out[80]= PrimitiveRoot[143]
```

SquaresR[d,n]	give the number of representations of an integer n as a sum of d squares
PowersRepresentations[n,k,p]	give the distinct representations of the integer n as a sum of k non-negative p^{th} integer powers

Representing an integer as a sum of squares or other powers.

Here are the representations of 101 as a sum of 3 squares.

```
In[81]:= PowersRepresentations[101, 3, 2]
Out[81]= {{0, 1, 10}, {1, 6, 8}, {2, 4, 9}, {4, 6, 7}}
```

Combinatorial Functions

$n!$	factorial $n(n-1)(n-2)\times\ldots\times 1$
$n!!$	double factorial $n(n-2)(n-4)\times\ldots$
`Binomial[n,m]`	binomial coefficient $\binom{n}{m} = n!/[m!(n-m)!]$
`Multinomial[n1,n2,...]`	multinomial coefficient $(n_1+n_2+\ldots)!/(n_1!\,n_2!\ldots)$
`CatalanNumber[n]`	Catalan number c_n
`Hyperfactorial[n]`	hyperfactorial $1^1\,2^2\ldots n^n$
`BarnesG[n]`	Barnes G-function $1!\,2!\ldots(n-2)!$
`Subfactorial[n]`	number of derangements of n objects
`Fibonacci[n]`	Fibonacci number F_n
`Fibonacci[n,x]`	Fibonacci polynomial $F_n(x)$
`LucasL[n]`	Lucas number L_n
`LucasL[n,x]`	Lucas polynomial $L_n(x)$
`HarmonicNumber[n]`	harmonic number H_n
`HarmonicNumber[n,r]`	harmonic number H_n^r of order r
`BernoulliB[n]`	Bernoulli number B_n
`BernoulliB[n,x]`	Bernoulli polynomial $B_n(x)$
`NorlundB[n,a]`	Nörlund polynomial $B_n^{(a)}$
`NorlundB[n,a,x]`	generalized Bernoulli polynomial $B_n^{(a)}(x)$
`EulerE[n]`	Euler number E_n
`EulerE[n,x]`	Euler polynomial $E_n(x)$
`StirlingS1[n,m]`	Stirling number of the first kind $S_n^{(m)}$
`StirlingS2[n,m]`	Stirling number of the second kind $\mathcal{S}_n^{(m)}$
`BellB[n]`	Bell number B_n
`BellB[n,x]`	Bell polynomial $B_n(x)$
`PartitionsP[n]`	the number $p(n)$ of unrestricted partitions of the integer n
`IntegerPartitions[n]`	partitions of an integer
`PartitionsQ[n]`	the number $q(n)$ of partitions of n into distinct parts
`Signature[{i1,i2,...}]`	the signature of a permutation

Combinatorial functions.

The *factorial function* $n!$ gives the number of ways of ordering n objects. For non-integer n, the numerical value of $n!$ is obtained from the gamma function, discussed in "Special Functions".

The *binomial coefficient* `Binomial[n, m]` can be written as $\binom{n}{m} = n!/[m!(n-m)!]$. It gives the number of ways of choosing m objects from a collection of n objects, without regard to order. The *Catalan numbers*, which appear in various tree enumeration problems, are given in terms of binomial coefficients as $c_n = \binom{2n}{n} \Big/ (n+1)$.

The *subfactorial* `Subfactorial[n]` gives the number of permutations of n objects that leave no object fixed. Such a permutation is called a derangement. The subfactorial is given by $n! \sum_{k=0}^{n} (-1)^k / k!$.

The *multinomial coefficient* `Multinomial[n1, n2, ...]`, denoted `(N; n1, n2, …, nm) = N! / (n1! n2! … nm!)`, gives the number of ways of partitioning N distinct objects into m sets of sizes n_i (with $N = \sum_{i=1}^{m} n_i$).

Mathematica gives the exact integer result for the factorial of an integer.

```
In[1]:= 30!
Out[1]= 265 252 859 812 191 058 636 308 480 000 000
```

For non-integers, *Mathematica* evaluates factorials using the gamma function.

```
In[2]:= 3.6!
Out[2]= 13.3813
```

Mathematica can give symbolic results for some binomial coefficients.

```
In[3]:= Binomial[n, 2]
Out[3]= 1/2 (-1 + n) n
```

This gives the number of ways of partitioning $6 + 5 = 11$ objects into sets containing 6 and 5 objects.

```
In[4]:= Multinomial[6, 5]
Out[4]= 462
```

The result is the same as $\binom{11}{6}$.

```
In[5]:= Binomial[11, 6]
Out[5]= 462
```

The *Fibonacci numbers* `Fibonacci[n]` satisfy the recurrence relation $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. They appear in a wide range of discrete mathematical problems. For large n, F_n/F_{n-1} approaches the golden ratio. The *Lucas numbers* `LucasL[n]` satisfy the same recurrence relation as the Fibonacci numbers do, but with initial conditions $L_1 = 1$ and $L_2 = 3$.

The *Fibonacci polynomials* `Fibonacci[n, x]` appear as the coefficients of t^n in the expansion of $t/(1 - x\,t - t^2) = \sum_{n=0}^{\infty} F_n(x)\,t^n$.

The *harmonic numbers* `HarmonicNumber[n]` are given by $H_n = \sum_{i=1}^{n} 1/i$; the harmonic numbers of order r `HarmonicNumber[n, r]` are given by $H_n^{(r)} = \sum_{i=1}^{n} 1/i^r$. Harmonic numbers appear in many combinatorial estimation problems, often playing the role of discrete analogs of logarithms.

The *Bernoulli polynomials* `BernoulliB[n, x]` satisfy the generating function relation $t\,e^{x\,t}/(e^t - 1) = \sum_{n=0}^{\infty} B_n(x)\,t^n/n!$. The *Bernoulli numbers* `BernoulliB[n]` are given by $B_n = B_n(0)$. The B_n appear as the coefficients of the terms in the Euler-Maclaurin summation formula for approximating integrals. The Bernoulli numbers are related to the *Genocchi numbers* by $G_n = 2\,(1 - 2^n)\,B_n$.

Numerical values for Bernoulli numbers are needed in many numerical algorithms. You can always get these numerical values by first finding exact rational results using `BernoulliB[n]`, and then applying `N`.

The *Euler polynomials* `EulerE[n, x]` have generating function $2\,e^{x\,t}/(e^t + 1) = \sum_{n=0}^{\infty} E_n(x)\,t^n/n!$, and the *Euler numbers* `EulerE[n]` are given by $E_n = 2^n\,E_n\left(\frac{1}{2}\right)$.

The *Nörlund polynomials* `NorlundB[n, a]` satisfy the generating function relation $t^a/(e^t - 1)^a = \sum_{n=0}^{\infty} B_n^{(a)}\,t^n/n!$. The Nörlund polynomials give the Bernoulli numbers when $a = 1$. For other positive integer values of a, the Nörlund polynomials give higher-order Bernoulli numbers. The *generalized Bernoulli polynomials* `NorlundB[n, a, x]` satisfy the generating function relation $t^a\,e^{x\,t}/(e^t - 1)^a = \sum_{n=0}^{\infty} B_n^{(a)}(x)\,t^n/n!$.

This gives the second Bernoulli polynomial $B_2(x)$.

```
In[6]:= BernoulliB[2, x]

Out[6]= 1/6 - x + x^2
```

You can also get Bernoulli polynomials by explicitly computing the power series for the generating function.

```
In[7]:= Series[t Exp[x t] / (Exp[t] - 1), {t, 0, 4}]

Out[7]= 1 + (-1/2 + x) t + 1/12 (1 - 6 x + 6 x^2) t^2 + 1/12 (x - 3 x^2 + 2 x^3) t^3 + 1/720 (-1 + 30 x^2 - 60 x^3 + 30 x^4) t^4 + O[t]^5
```

`BernoulliB[n]` gives exact rational-number results for Bernoulli numbers.

```
In[8]:= BernoulliB[20]

Out[8]= -174 611/330
```

Stirling numbers show up in many combinatorial enumeration problems. For *Stirling numbers of the first kind* `StirlingS1[n, m]`, $(-1)^{n-m} S_n^{(m)}$ gives the number of permutations of n elements which contain exactly m cycles. These Stirling numbers satisfy the generating function relation $x(x-1)\ldots(x-n+1) = \sum_{m=0}^{n} S_n^{(m)} x^m$. Note that some definitions of the $S_n^{(m)}$ differ by a factor $(-1)^{n-m}$ from what is used in *Mathematica*.

Stirling numbers of the second kind `StirlingS2[n, m]`, sometimes denoted $\mathcal{S}_n^{(m)}$, give the number of ways of partitioning a set of n elements into m non-empty subsets. They satisfy the relation $x^n = \sum_{m=0}^{n} \mathcal{S}_n^{(m)} x(x-1)\ldots(x-m+1)$.

The *Bell numbers* `BellB[n]` give the total number of ways that a set of n elements can be partitioned into non-empty subsets. The *Bell polynomials* `BellB[n, x]` satisfy the generating function relation $e^{(e^t-1)x} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!}$.

The *partition function* `PartitionsP[n]` gives the number of ways of writing the integer n as a sum of positive integers, without regard to order. `PartitionsQ[n]` gives the number of ways of writing n as a sum of positive integers, with the constraint that all the integers in each sum are distinct.

`IntegerPartitions[n]` gives a list of the partitions of n, with length `PartitionsP[n]`.

This gives a table of Stirling numbers of the first kind.

```
In[9]:= Table[StirlingS1[5, i], {i, 5}]
Out[9]= {24, -50, 35, -10, 1}
```

The Stirling numbers appear as coefficients in this product.

```
In[10]:= Expand[Product[x - i, {i, 0, 4}]]
Out[10]= 24 x - 50 x^2 + 35 x^3 - 10 x^4 + x^5
```

Here are the partitions of 4.

```
In[11]:= IntegerPartitions[4]
Out[11]= {{4}, {3, 1}, {2, 2}, {2, 1, 1}, {1, 1, 1, 1}}
```

The number of partitions is given by `PartitionsP[4]`.

```
In[12]:= Length[%] == PartitionsP[4]
Out[12]= True
```

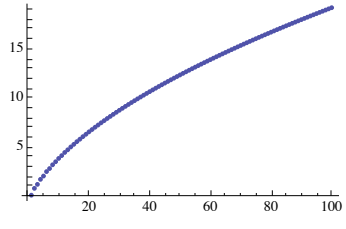
This gives the number of partitions of 100, with and without the constraint that the terms should be distinct.

```
In[13]:= {PartitionsQ[100], PartitionsP[100]}
Out[13]= {444 793, 190 569 292}
```

The partition function $p(n)$ increases asymptotically like $e^{\sqrt{n}}$. Note that you cannot simply use `Plot` to generate a plot of a function like `PartitionsP` because the function can only be evaluated with integer arguments.

```
In[14]:= ListPlot[Table[N[Log[PartitionsP[n]]], {n, 100}]]
```

Out[14]=



Most of the functions here allow you to *count* various kinds of combinatorial objects. Functions like `IntegerPartitions` and `Permutations` allow you instead to *generate* lists of various combinations of elements.

The *signature function* `Signature[{i1, i2, ...}]` gives the signature of a permutation. It is equal to $+1$ for even permutations (composed of an even number of transpositions), and to -1 for odd permutations. The signature function can be thought of as a totally antisymmetric tensor, *Levi-Civita symbol* or *epsilon symbol*.

`ClebschGordan[{j1,m1},{j2,m2},{j,m}]`	Clebsch-Gordan coefficient
`ThreeJSymbol[{j1,m1},{j2,m2},{j3,m3}]`	Wigner 3-j symbol
`SixJSymbol[{j1,j2,j3},{j4,j5,j6}]`	Racah 6-j symbol

Rotational coupling coefficients.

Clebsch-Gordan coefficients and n-j symbols arise in the study of angular momenta in quantum mechanics, and in other applications of the rotation group. The *Clebsch-Gordan coefficients* `ClebschGordan[{j1, m1}, {j2, m2}, {j, m}]` give the coefficients in the expansion of the quantum mechanical angular momentum state $|j, m\rangle$ in terms of products of states $|j_1, m_1\rangle\,|j_2, m_2\rangle$.

The *3-j symbols* or *Wigner coefficients* `ThreeJSymbol[{j1, m1}, {j2, m2}, {j3, m3}]` are a more symmetrical form of Clebsch-Gordan coefficients. In *Mathematica*, the Clebsch-Gordan coefficients are given in terms of 3-j symbols by $C^{j_1\,j_2\,j_3}_{m_1\,m_2\,m_3} = (-1)^{m_3+j_1-j_2}\sqrt{2\,j_3+1}\begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & -m_3 \end{pmatrix}$.

The *6-j symbols* `SixJSymbol[{j1, j2, j3}, {j4, j5, j6}]` give the couplings of three quantum mechanical angular momentum states. The *Racah coefficients* are related by a phase to the 6-j symbols.

You can give symbolic parameters in 3-j symbols.

```
In[15]:= ThreeJSymbol[{j, m}, {j + 1 / 2, -m - 1 / 2}, {1 / 2, 1 / 2}]
```

$$\text{Out[15]}= -\frac{(-1)^{-j+m}\sqrt{\frac{1+j+m}{1+3\,j+2\,j^2}}}{\sqrt{2}}$$

Elementary Transcendental Functions

`Exp[z]`	exponential function e^z
`Log[z]`	logarithm $\log_e(z)$
`Log[b,z]`	logarithm $\log_b(z)$ to base b
`Log2[z]` , `Log10[z]`	logarithm to base 2 and 10
`Sin[z]` , `Cos[z]` , `Tan[z]` , `Csc[z]` , `Sec[z]` , `Cot[z]`	trigonometric functions (with arguments in radians)
`ArcSin[z]` , `ArcCos[z]` , `ArcTan[z]` , `ArcCsc[z]` , `ArcSec[z]` , `ArcCot[z]`	inverse trigonometric functions (giving results in radians)
`ArcTan[x,y]`	the argument of $x + iy$
`Sinh[z]` , `Cosh[z]` , `Tanh[z]` , `Csch[z]` , `Sech[z]` , `Coth[z]`	hyperbolic functions
`ArcSinh[z]` , `ArcCosh[z]` , `ArcTanh[z]` , `ArcCsch[z]` , `ArcSech[z]` , `ArcCoth[z]`	inverse hyperbolic functions
`Sinc[z]`	sinc function $\sin(z)/z$
`Haversine[z]`	haversine function $\mathrm{hav}(z)$
`InverseHaversine[z]`	inverse haversine function $\mathrm{hav}^{-1}(z)$
`Gudermannian[z]`	Gudermannian function $\mathrm{gd}(z)$
`InverseGudermannian[z]`	inverse Gudermannian function $\mathrm{gd}^{-1}(z)$

Elementary transcendental functions.

Mathematica gives exact results for logarithms whenever it can. Here is $\log_2 1024$.

```
In[1]:= Log[2, 1024]
Out[1]= 10
```

You can find the numerical values of mathematical functions to any precision.

```
In[2]:= N[Log[2], 40]
Out[2]= 0.6931471805599453094172321214581765680755
```

This gives a complex number result.

```
In[3]:= N[Log[-2]]
Out[3]= 0.693147 + 3.14159 i
```

Mathematica can evaluate logarithms with complex arguments.

```
In[4]:= N[Log[2 + 8 I]]
Out[4]= 2.10975 + 1.32582 i
```

The arguments of trigonometric functions are always given in radians.

```
In[5]:= Sin[Pi / 2]
Out[5]= 1
```

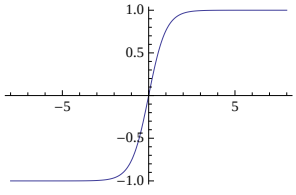
You can convert from degrees by explicitly multiplying by the constant `Degree`.

```
In[6]:= N[Sin[30 Degree]]
Out[6]= 0.5
```

Here is a plot of the hyperbolic tangent function. It has a characteristic "sigmoidal" form.

```
In[7]:= Plot[Tanh[x], {x, -8, 8}]
```

Out[7]=



The *haversine function* `Haversine[z]` is defined by $\sin^2(z/2)$. The *inverse haversine function* `InverseHaversine[z]` is defined by $2\sin^{-1}\left(\sqrt{z}\right)$. The *Gudermannian function* `Gudermannian[z]` is defined as $\text{gd}(z) = 2\tan^{-1}(e^z) - \frac{\pi}{2}$. The *inverse Gudermannian function* `InverseGudermannian[z]` is defined by $\text{gd}^{-1}(z) = \log[\tan(z/2 + \pi/4)]$. The Gudermannian satisfies such relations as $\sinh(z) = \tan[\text{gd}(x)]$. The *sinc function* `Sinc[z]` is the Fourier transform of a square signal.

There are a number of additional trigonometric and hyperbolic functions that are sometimes used. The *versine* function is sometimes encountered in the literature and simply is $\text{vers}(z) = 2\,\text{hav}(z)$. The *coversine* function is defined as $\text{covers}(z) = 1 - \sin(z)$. The complex exponential e^{ix} is sometimes written as $\text{cis}(x)$.

Functions That Do Not Have Unique Values

When you ask for the square root s of a number a, you are effectively asking for the solution to the equation $s^2 = a$. This equation, however, in general has two different solutions. Both $s = 2$ and $s = -2$ are, for example, solutions to the equation $s^2 = 4$. When you evaluate the "function" $\sqrt{4}$, however, you usually want to get a single number, and so you have to choose one of these two solutions. A standard choice is that $\sqrt{x}$ should be positive for $x > 0$. This is what the *Mathematica* function `Sqrt[x]` does.

The need to make one choice from two solutions means that `Sqrt[x]` cannot be a true *inverse function* for `x^2`. Taking a number, squaring it, and then taking the square root can give you a different number than you started with.

$\sqrt{4}$ gives $+2$, not -2.

```
In[1]:= Sqrt[4]
Out[1]= 2
```

Squaring and taking the square root does not necessarily give you the number you started with.

```
In[2]:= Sqrt[(-2)^2]
Out[2]= 2
```

When you evaluate $\sqrt{-2i}$, there are again two possible answers: $-1 + i$ and $1 - i$. In this case, however, it is less clear which one to choose.

There is in fact no way to choose $\sqrt{z}$ so that it is continuous for all complex values of z. There has to be a "branch cut"—a line in the complex plane across which the function $\sqrt{z}$ is discontinuous. *Mathematica* adopts the usual convention of taking the branch cut for $\sqrt{z}$ to be along the negative real axis.

This gives $1 - i$, not $-1 + i$.

```
In[3]:= N[Sqrt[-2 I]]
Out[3]= 1. - 1. I
```

The branch cut in `Sqrt` along the negative real axis means that values of `Sqrt[z]` with z just above and below the axis are very different.

```
In[4]:= {Sqrt[-2 + 0.1 I], Sqrt[-2 - 0.1 I]}
Out[4]= {0.0353443 + 1.41466 i, 0.0353443 - 1.41466 i}
```

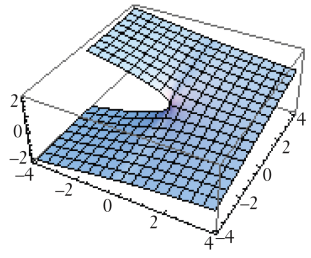
Their squares are nevertheless close.

```
In[5]:= %^2
Out[5]= {-2. + 0.1 i, -2. - 0.1 i}
```

The discontinuity along the negative real axis is quite clear in this three-dimensional picture of the imaginary part of the square root function.

```
In[6]:= Plot3D[Im[Sqrt[x + I y]], {x, -4, 4}, {y, -4, 4}]
```

Out[6]=



When you find an n^{th} root using $z^{1/n}$, there are, in principle, n possible results. To get a single value, you have to choose a particular *principal root*. There is absolutely no guarantee that taking the n^{th} root of an n^{th} power will leave you with the same number.

This takes the tenth power of a complex number. The result is unique.

```
In[7]:= (2.5 + I)^10
Out[7]= -15 781.2 - 12 335.8 i
```

There are 10 possible tenth roots. *Mathematica* chooses one of them. In this case it is not the number whose tenth power you took.

```
In[8]:= %^(1 / 10)
Out[8]= 2.61033 - 0.660446 i
```

There are many mathematical functions which, like roots, essentially give solutions to equations. The logarithm function and the inverse trigonometric functions are examples. In almost all cases, there are many possible solutions to the equations. Unique "principal" values neverthe-

less have to be chosen for the functions. The choices cannot be made continuous over the whole complex plane. Instead, lines of discontinuity, or branch cuts, must occur. The positions of these branch cuts are often quite arbitrary. *Mathematica* makes the most standard mathematical choices for them.

`Sqrt[z]` and `z^s`	$(-\infty, 0)$ for `Re` $s > 0$, $(-\infty, 0]$ for `Re` $s \le 0$ (s not an integer)
`Exp[z]`	none
`Log[z]`	$(-\infty, 0]$
trigonometric functions	none
`ArcSin[z]` and `ArcCos[z]`	$(-\infty, -1)$ and $(+1, +\infty)$
`ArcTan[z]`	$(-i\infty, -i]$ and $(i, i\infty]$
`ArcCsc[z]` and `ArcSec[z]`	$(-1, +1)$
`ArcCot[z]`	$[-i, +i]$
hyperbolic functions	none
`ArcSinh[z]`	$(-i\infty, -i)$ and $(+i, +i\infty)$
`ArcCosh[z]`	$(-\infty, +1)$
`ArcTanh[z]`	$(-\infty, -1]$ and $[+1, +\infty)$
`ArcCsch[z]`	$(-i, i)$
`ArcSech[z]`	$(-\infty, 0]$ and $(+1, +\infty)$
`ArcCoth[z]`	$[-1, +1]$

Some branch-cut discontinuities in the complex plane.

`ArcSin` is a multiple-valued function, so there is no guarantee that it always gives the "inverse" of `Sin`.

```
In[9]:= ArcSin[Sin[4.5]]
Out[9]= -1.35841
```

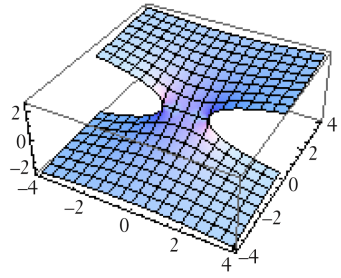
Values of `ArcSin[z]` on opposite sides of the branch cut can be very different.

```
In[10]:= {ArcSin[2 + 0.1 I], ArcSin[2 - 0.1 I]}
Out[10]= {1.51316 + 1.31888 i, 1.51316 - 1.31888 i}
```

A three-dimensional picture, showing the two branch cuts for the function $\sin^{-1}(z)$.

```
In[11]:= Plot3D[Im[ArcSin[x + I y]], {x, -4, 4}, {y, -4, 4}]
```

Out[11]=



Mathematical Constants

`I`	$i = \sqrt{-1}$
`Infinity`	∞
`Pi`	$\pi \simeq 3.14159$
`Degree`	$\pi/180$: degrees to radians conversion factor
`GoldenRatio`	$\phi = \left(1 + \sqrt{5}\right)\Big/ 2 \simeq 1.61803$
`E`	$e \simeq 2.71828$
`EulerGamma`	Euler's constant $\gamma \simeq 0.577216$
`Catalan`	Catalan's constant $C \simeq 0.915966$
`Khinchin`	Khinchin's constant $K \simeq 2.68545$
`Glaisher`	Glaisher's constant $A \simeq 1.28243$

Mathematical constants.

Euler's constant `EulerGamma` is given by the limit $\gamma = \lim_{m\to\infty}\left(\sum_{k=1}^{m} \frac{1}{k} - \log m\right)$. It appears in many integrals, and asymptotic formulas. It is sometimes known as the *Euler-Mascheroni constant*, and denoted C.

Catalan's constant `Catalan` is given by the sum $\sum_{k=0}^{\infty} (-1)^k (2k+1)^{-2}$. It often appears in asymptotic estimates of combinatorial functions. It is variously denoted by C, K, or G.

Khinchin's constant `Khinchin` (sometimes called Khintchine's constant) is given by $\prod_{s=1}^{\infty}\left(1 + \frac{1}{s(s+2)}\right)^{\log_2 s}$. It gives the geometric mean of the terms in the continued fraction representation for a typical real number.

Glaisher's constant `Glaisher` A (sometimes called the Glaisher-Kinkelin constant) satisfies $\log(A) = \frac{1}{12} - \zeta'(-1)$, where ζ is the Riemann zeta function. It appears in various sums and integrals, particularly those involving gamma and zeta functions.

Mathematical constants can be evaluated to arbitrary precision.

```
In[1]:= N[EulerGamma, 40]
Out[1]= 0.5772156649015328606065120900824024310422
```

Exact computations can also be done with them.

```
In[2]:= IntegerPart[GoldenRatio^100]
Out[2]= 792 070 839 848 372 253 126
```

Orthogonal Polynomials

`LegendreP[n,x]`	Legendre polynomials $P_n(x)$
`LegendreP[n,m,x]`	associated Legendre polynomials $P_n^m(x)$
`SphericalHarmonicY[l,m,θ,ϕ]`	spherical harmonics $Y_l^m(\theta, \phi)$
`GegenbauerC[n,m,x]`	Gegenbauer polynomials $C_n^{(m)}(x)$
`ChebyshevT[n,x]` `, ChebyshevU[n,x]`	Chebyshev polynomials $T_n(x)$ and $U_n(x)$ of the first and second kinds
`HermiteH[n,x]`	Hermite polynomials $H_n(x)$
`LaguerreL[n,x]`	Laguerre polynomials $L_n(x)$
`LaguerreL[n,a,x]`	generalized Laguerre polynomials $L_n^a(x)$
`ZernikeR[n,m,x]`	Zernike radial polynomials $R_n^{(m)}(x)$
`JacobiP[n,a,b,x]`	Jacobi polynomials $P_n^{(a,b)}(x)$

Orthogonal polynomials.

Legendre polynomials `LegendreP[n, x]` arise in studies of systems with three-dimensional spherical symmetry. They satisfy the differential equation $(1-x^2)\,y'' - 2\,x\,y' + n(n+1)\,y = 0$, and the orthogonality relation $\int_{-1}^{1} P_m(x)\,P_n(x)\,dx = 0$ for $m \neq n$.

The *associated Legendre polynomials* `LegendreP[n, m, x]` are obtained from derivatives of the Legendre polynomials according to $P_n^m(x) = (-1)^m \left(1 - x^2\right)^{m/2} d^m [P_n(x)]/d\, x^m$. Notice that for odd integers $m \le n$, the $P_n^m(x)$ contain powers of $\sqrt{1 - x^2}$, and are therefore not strictly polynomials. The $P_n^m(x)$ reduce to $P_n(x)$ when $m = 0$.

The *spherical harmonics* `SphericalHarmonicY[l, m, θ, ϕ]` are related to associated Legendre polynomials. They satisfy the orthogonality relation $\int Y_l^m(\theta, \phi)\, \overline{Y}_{l'}^{m'}(\theta, \phi)\, d\omega = 0$ for $l \neq l'$ or $m \neq m'$, where $d\,\omega$ represents integration over the surface of the unit sphere.

This gives the algebraic form of the Legendre polynomial $P_8(x)$.

```
In[1]:= LegendreP[8, x]
```

$$\text{Out[1]}= \frac{1}{128}\left(35 - 1260\, x^2 + 6930\, x^4 - 12\,012\, x^6 + 6435\, x^8\right)$$

The integral $\int_{-1}^{1} P_7(x)\, P_8(x)\, d\, x$ gives zero by virtue of the orthogonality of the Legendre polynomials.

```
In[2]:= Integrate[LegendreP[7, x] LegendreP[8, x], {x, -1, 1}]
Out[2]= 0
```

Integrating the square of a single Legendre polynomial gives a nonzero result.

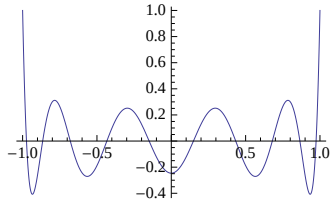
```
In[3]:= Integrate[LegendreP[8, x]^2, {x, -1, 1}]
```

$$\text{Out[3]}= \frac{2}{17}$$

High-degree Legendre polynomials oscillate rapidly.

```
In[4]:= Plot[LegendreP[10, x], {x, -1, 1}]
```

Out[4]=



The associated Legendre "polynomials" involve fractional powers.

```
In[5]:= LegendreP[8, 3, x]
```

$$\text{Out[5]}= \frac{3465}{8}\sqrt{1 - x^2}\left(-1 + x^2\right)\left(3\, x - 26\, x^3 + 39\, x^5\right)$$

"Special Functions" discusses the generalization of Legendre polynomials to Legendre functions, which can have non-integer degrees.

```
In[6]:= LegendreP[8.1, 0]
Out[6]= 0.268502
```

Gegenbauer polynomials `GegenbauerC[n, m, x]` can be viewed as generalizations of the Legendre polynomials to systems with $(m+2)$-dimensional spherical symmetry. They are sometimes known as *ultraspherical polynomials*.

`GegenbauerC[n, 0, x]` is always equal to zero. `GegenbauerC[n, x]` is however given by the limit $\lim_{m\to 0} C_n^{(m)}(x)/m$. This form is sometimes denoted $C_n^{(0)}(x)$.

Series of Chebyshev polynomials are often used in making numerical approximations to functions. The *Chebyshev polynomials of the first kind* `ChebyshevT[n, x]` are defined by $T_n(\cos\theta) = \cos(n\theta)$. They are normalized so that $T_n(1) = 1$. They satisfy the orthogonality relation $\int_{-1}^{1} T_m(x)\, T_n(x) \left(1-x^2\right)^{-1/2} dx = 0$ for $m \neq n$. The $T_n(x)$ also satisfy an orthogonality relation under summation at discrete points in x corresponding to the roots of $T_n(x)$.

The *Chebyshev polynomials of the second kind* `ChebyshevU[n, z]` are defined by $U_n(\cos\theta) = \sin[(n+1)\theta]/\sin\theta$. With this definition, $U_n(1) = n+1$. The U_n satisfy the orthogonality relation $\int_{-1}^{1} U_m(x)\, U_n(x) \left(1-x^2\right)^{1/2} dx = 0$ for $m \neq n$.

The name "Chebyshev" is a transliteration from the Cyrillic alphabet; several other spellings, such as "Tschebyscheff", are sometimes used.

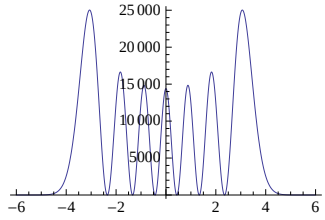
Hermite polynomials `HermiteH[n, x]` arise as the quantum-mechanical wave functions for a harmonic oscillator. They satisfy the differential equation $y'' - 2x\,y' + 2n\,y = 0$, and the orthogonality relation $\int_{-\infty}^{\infty} H_m(x)\, H_n(x)\, e^{-x^2} dx = 0$ for $m \neq n$. An alternative form of Hermite polynomials sometimes used is $He_n(x) = 2^{-n/2} H_n\left(x/\sqrt{2}\right)$ (a different overall normalization of the $He_n(x)$ is also sometimes used).

The Hermite polynomials are related to the *parabolic cylinder functions* or *Weber functions* $D_n(x)$ by $D_n(x) = 2^{-n/2}\, e^{-x^2/4}\, H_n\left(x/\sqrt{2}\right)$.

This gives the density for an excited state of a quantum-mechanical harmonic oscillator. The average of the wiggles is roughly the classical physics result.

```
In[7]:= Plot[(HermiteH[6, x] Exp[-x^2 / 2])^2, {x, -6, 6}]
```

Out[7]=



Generalized Laguerre polynomials `LaguerreL[n, a, x]` are related to hydrogen atom wave functions in quantum mechanics. They satisfy the differential equation $x\,y'' + (a + 1 - x)\,y' + n\,y = 0$, and the orthogonality relation $\int_0^\infty L_m^a(x)\,L_n^a(x)\,x^a\,e^{-x}\,dx = 0$ for $m \neq n$. The *Laguerre polynomials* `LaguerreL[n, x]` correspond to the special case $a = 0$.

You can get formulas for generalized Laguerre polynomials with arbitrary values of a.

```
In[8]:= LaguerreL[2, a, x]
```

Out[8]=

$$\frac{1}{2}\left(2 + 3\,a + a^2 - 4\,x - 2\,a\,x + x^2\right)$$

Zernike radial polynomials `ZernikeR[n, m, x]` are used in studies of aberrations in optics. They satisfy the orthogonality relation $\int_0^1 R_n^{(m)}(x)\,R_k^{(m)}(x)\,x\,dx = 0$ for $n \neq k$.

Jacobi polynomials `JacobiP[n, a, b, x]` occur in studies of the rotation group, particularly in quantum mechanics. They satisfy the orthogonality relation $\int_{-1}^1 P_m^{(a,b)}(x)\,P_n^{(a,b)}(x)\,(1-x)^a\,(1+x)^b\,dx = 0$ for $m \neq n$. Legendre, Gegenbauer, Chebyshev and Zernike polynomials can all be viewed as special cases of Jacobi polynomials. The Jacobi polynomials are sometimes given in the alternative form $G_n(p, q, x) = n!\,\Gamma(n + p)/\Gamma(2\,n + p)\,P_n^{(p-q,q-1)}(2\,x - 1)$.

Special Functions

Mathematica includes all the common special functions of mathematical physics found in standard handbooks. We will discuss each of the various classes of functions in turn.

One point you should realize is that in the technical literature there are often several conflicting definitions of any particular special function. When you use a special function in *Mathematica*, therefore, you should be sure to look at the definition given here to confirm that it is exactly what you want.

Mathematica gives exact results for some values of special functions.

```
In[1]:= Gamma[15 / 2]
```

$$\text{Out[1]}= \frac{135\,135\,\sqrt{\pi}}{128}$$

No exact result is known here.

```
In[2]:= Gamma[15 / 7]
```

$$\text{Out[2]}= \text{Gamma}\left[\frac{15}{7}\right]$$

A numerical result, to arbitrary precision, can nevertheless be found.

```
In[3]:= N[%, 40]
Out[3]= 1.069071500448624397994137689702693267367
```

You can give complex arguments to special functions.

```
In[4]:= Gamma[3 + 4 I] // N
Out[4]= 0.00522554 - 0.172547 i
```

Special functions automatically get applied to each element in a list.

```
In[5]:= Gamma[{3 / 2, 5 / 2, 7 / 2}]
```

$$\text{Out[5]}= \left\{\frac{\sqrt{\pi}}{2}, \frac{3\sqrt{\pi}}{4}, \frac{15\sqrt{\pi}}{8}\right\}$$

Mathematica knows analytical properties of special functions, such as derivatives.

```
In[6]:= D[Gamma[x], {x, 2}]
Out[6]= Gamma[x] PolyGamma[0, x]^2 + Gamma[x] PolyGamma[1, x]
```

You can use `FindRoot` to find roots of special functions.

```
In[7]:= FindRoot[BesselJ[0, x], {x, 1}]
Out[7]= {x → 2.40483}
```

Special functions in *Mathematica* can usually be evaluated for arbitrary complex values of their arguments. Often, however, the defining relations given in this tutorial apply only for some special choices of arguments. In these cases, the full function corresponds to a suitable extension or "analytic continuation" of these defining relations. Thus, for example, integral representations of functions are valid only when the integral exists, but the functions themselves can usually be defined elsewhere by analytic continuation.

As a simple example of how the domain of a function can be extended, consider the function represented by the sum $\sum_{k=0}^{\infty} x^k$. This sum converges only when $|x| < 1$. Nevertheless, it is easy to show analytically that for any x, the complete function is equal to $1/(1-x)$. Using this form, you can easily find a value of the function for any x, at least so long as $x \neq 1$.

Gamma and Related Functions

`Beta[a,b]`	Euler beta function $\mathrm{B}(a, b)$
`Beta[z,a,b]`	incomplete beta function $\mathrm{B}_z(a, b)$
`BetaRegularized[z,a,b]`	regularized incomplete beta function $I(z, a, b)$
`Gamma[z]`	Euler gamma function $\Gamma(z)$
`Gamma[a,z]`	incomplete gamma function $\Gamma(a, z)$
`Gamma[a,z0,z1]`	generalized incomplete gamma function $\Gamma(a, z_0) - \Gamma(a, z_1)$
`GammaRegularized[a,z]`	regularized incomplete gamma function $Q(a, z)$
`InverseBetaRegularized[s,a,b]`	inverse beta function
`InverseGammaRegularized[a,s]`	inverse gamma function
`Pochhammer[a,n]`	Pochhammer symbol $(a)_n$
`PolyGamma[z]`	digamma function $\psi(z)$
`PolyGamma[n,z]`	n^{th} derivative of the digamma function $\psi^{(n)}(z)$
`LogGamma[z]`	Euler log-gamma function $\log \Gamma(z)$
`LogBarnesG[z]`	logarithm of Barnes G-function $\log \mathrm{G}(z)$
`BarnesG[z]`	Barnes G-function $G(z)$
`Hyperfactorial[n]`	hyperfactorial function $H(n)$

Gamma and related functions.

The *Euler gamma function* `Gamma[z]` is defined by the integral $\Gamma(z) = \int_0^\infty t^{z-1}\, e^{-t}\, dt$. For positive integer n, $\Gamma(n) = (n-1)!$. $\Gamma(z)$ can be viewed as a generalization of the factorial function, valid for complex arguments z.

There are some computations, particularly in number theory, where the logarithm of the gamma function often appears. For positive real arguments, you can evaluate this simply as `Log[Gamma[z]]`. For complex arguments, however, this form yields spurious discontinuities. *Mathematica* therefore includes the separate function `LogGamma[z]`, which yields the *logarithm of the gamma function* with a single branch cut along the negative real axis.

The *Euler beta function* `Beta[a, b]` is $\mathrm{B}(a, b) = \Gamma(a)\,\Gamma(b)/\Gamma(a+b) = \int_0^1 t^{a-1}\,(1-t)^{b-1}\, dt$.

The *Pochhammer symbol* or *rising factorial* `Pochhammer[a, n]` is $(a)_n = a(a+1)\ldots(a+n-1) = \Gamma(a+n)/\Gamma(a)$. It often appears in series expansions for hypergeometric functions. Note that the Pochhammer symbol has a definite value even when the gamma functions which appear in its definition are infinite.

The *incomplete gamma function* `Gamma[a, z]` is defined by the integral $\Gamma(a, z) = \int_z^\infty t^{a-1}\, e^{-t}\, dt$. *Mathematica* includes a generalized incomplete gamma function `Gamma[a, z0, z1]` defined as $\int_{z_0}^{z_1} t^{a-1}\, e^{-t}\, dt$.

The alternative incomplete gamma function $\gamma(a, z)$ can therefore be obtained in *Mathematica* as `Gamma[a, 0, z]`.

The *incomplete beta function* `Beta[z, a, b]` is given by $\mathrm{B}_z(a, b) = \int_0^z t^{a-1}\,(1-t)^{b-1}\, dt$. Notice that in the incomplete beta function, the parameter z is an *upper* limit of integration, and appears as the *first* argument of the function. In the incomplete gamma function, on the other hand, z is a *lower* limit of integration, and appears as the *second* argument of the function.

In certain cases, it is convenient not to compute the incomplete beta and gamma functions on their own, but instead to compute *regularized forms* in which these functions are divided by complete beta and gamma functions. *Mathematica* includes the *regularized incomplete beta function* `BetaRegularized[z, a, b]` defined for most arguments by $I(z, a, b) = \mathrm{B}(z, a, b)/\mathrm{B}(a, b)$, but taking into account singular cases. *Mathematica* also includes the *regularized incomplete gamma function* `GammaRegularized[a, z]` defined by $Q(a, z) = \Gamma(a, z)/\Gamma(a)$, with singular cases taken into account.

The incomplete beta and gamma functions, and their inverses, are common in statistics. The *inverse beta function* `InverseBetaRegularized[s, a, b]` is the solution for z in $s = I(z, a, b)$. The *inverse gamma function* `InverseGammaRegularized[a, s]` is similarly the solution for z in $s = Q(a, z)$.

Derivatives of the gamma function often appear in summing rational series. The *digamma function* `PolyGamma[z]` is the logarithmic derivative of the gamma function, given by $\psi(z) = \Gamma'(z)/\Gamma(z)$. For integer arguments, the digamma function satisfies the relation $\psi(n) = -\gamma + H_{n-1}$, where γ is Euler's constant (`EulerGamma` in *Mathematica*) and H_n are the harmonic numbers.

The *polygamma functions* `PolyGamma[n, z]` are given by $\psi^{(n)}(z) = d^n \psi(z)/d z^n$. Notice that the digamma function corresponds to $\psi^{(0)}(z)$. The general form $\psi^{(n)}(z)$ is the $(n+1)^{\text{th}}$, not the n^{th}, logarithmic derivative of the gamma function. The polygamma functions satisfy the relation $\psi^{(n)}(z) = (-1)^{n+1}\, n!\, \sum_{k=0}^{\infty} 1/(z+k)^{n+1}$. `PolyGamma[ν, z]` is defined for arbitrary complex ν by fractional calculus analytic continuation.

`BarnesG[z]` is a generalization of the `Gamma` function and is defined by its functional identity `BarnesG[z + 1] = Gamma[z] BarnesG[z]`, where the third derivative of the logarithm of `BarnesG` is positive for positive z. `BarnesG` is an entire function in the complex plane.

`LogBarnesG[z]` is a holomorphic function with a branch cut along the negative real-axis such that `Exp[LogBarnesG[z]] = BarnesG[z]`.

`Hyperfactorial[n]` is a generalization of $\prod_{k=1}^{n} k^k$ to the complex plane.

Many exact results for gamma and polygamma functions are built into *Mathematica*.

```
In[1]:= PolyGamma[6]

Out[1]= 137/60 - EulerGamma
```

Here is a contour plot of the gamma function in the complex plane.

```
In[2]:= ContourPlot[Abs[Gamma[x + I y]], {x, -3, 3}, {y, -2, 2}, PlotPoints -> 50]
```

Zeta and Related Functions

`DirichletL[k,j,s]`	Dirichlet L-function $L(\chi, s)$
`LerchPhi[z,s,a]`	Lerch's transcendent $\Phi(z, s, a)$
`PolyLog[n,z]`	polylogarithm function $\mathrm{Li}_n(z)$
`PolyLog[n,p,z]`	Nielsen generalized polylogarithm function $S_{n,p}(z)$
`RamanujanTau[n]`	Ramanujan τ function $\tau(n)$
`RamanujanTauL[n]`	Ramanujan τ Dirichlet L function $L(s)$
`RamanujanTauTheta[n]`	Ramanujan τ theta function $\theta(t)$
`RamanujanTauZ[n]`	Ramanujan τ Z function $Z(t)$
`RiemannSiegelTheta[t]`	Riemann-Siegel function $\vartheta(t)$
`RiemannSiegelZ[t]`	Riemann-Siegel function $Z(t)$
`StieltjesGamma[n]`	Stieltjes constants γ_n
`Zeta[s]`	Riemann zeta function $\zeta(s)$
`Zeta[s,a]`	generalized Riemann zeta function $\zeta(s, a)$
`HurwitzZeta[s,a]`	Hurwitz zeta function $\zeta(s, a)$
`HurwitzLerchPhi[z,s,a]`	Hurwitz-Lerch transcendent $\Phi(z, s, a)$

Zeta and related functions.

The Dirichlet-L function `DirichletL[k, j, s]` is implemented as $L(\chi, s) = \sum_{n=1}^{\infty} \chi(n)\, n^{-s}$ (for $\mathrm{Re}(s) > 1$) where $\chi(n)$ is a Dirichlet character with modulus k and index j.

The *Riemann zeta function* `Zeta[s]` is defined by the relation $\zeta(s) = \sum_{k=1}^{\infty} k^{-s}$ (for $s > 1$). Zeta functions with integer arguments arise in evaluating various sums and integrals. *Mathematica* gives exact results when possible for zeta functions with integer arguments.

There is an analytic continuation of $\zeta(s)$ for arbitrary complex $s \neq 1$. The zeta function for complex arguments is central to number theoretic studies of the distribution of primes. Of particular importance are the values on the critical line $\mathrm{Re}(s) = \frac{1}{2}$.

In studying $\zeta\left(\frac{1}{2} + i\,t\right)$, it is often convenient to define the two *Riemann-Siegel functions* `RiemannSiegelZ[t]` and `RiemannSiegelTheta[t]` according to $Z(t) = e^{i\,\vartheta(t)}\,\zeta\left(\frac{1}{2} + i\,t\right)$ and $\vartheta(t) = \mathrm{Im}\log\Gamma\left(\frac{1}{4} + i\,t\,/\,2\right) - t\log(\pi)/2$ (for t real). Note that the Riemann-Siegel functions are both real as long as t is real.

The *Stieltjes constants* `StieltjesGamma[n]` are generalizations of Euler's constant which appear in the series expansion of $\zeta(s)$ around its pole at $s=1$; the coefficient of $(1-s)^n$ is $\gamma_n/n!$. Euler's constant is γ_0.

The *generalized Riemann zeta function* `Zeta[s, a]` is implemented as $\zeta(s,a)=\sum_{k=0}^{\infty}\left((k+a)^2\right)^{-s/2}$, where any term with $k+a=0$ is excluded.

The Hurwitz zeta function `HurwitzZeta[s, a]` is implemented as $\zeta(s,a)=\sum_{k=0}^{\infty}(k+a)^{-s}$.

The Ramanujan τ Dirichlet L function `RamanujanTauL[s]` is defined by $L(s)=\sum_{n=1}^{\infty}\frac{\tau(n)}{n^s}$ (for $\mathrm{Re}(s)>6$), with coefficients `RamanujanTau[n]`. In analogy with the *Riemann zeta function*, it is again convenient to define the functions `RamanujanTauZ[t]` and `RamanujanTauTheta[t]`.

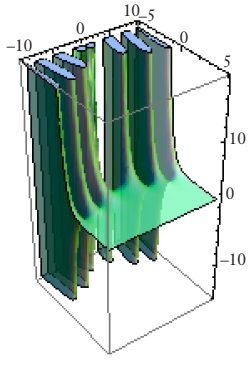
Here is the numerical approximation for $L(6, 2, 1.0+i)$.

```
In[77]:= DirichletL[6, 2, 1. + i]
Out[77]= 0.978008 + 0.0954731 i
```

Here is a three-dimensional picture of the real part of a Dirichlet L-function.

```
In[4]:= Plot3D[Re@DirichletL[6, 2, u + i v], {u, -5, 5}, {v, -10, 10}, Mesh → None,
         PlotStyle → Directive[Opacity[0.7], Green, Specularity[10]], BoxRatios → {1, 1, 2}]
```

Out[4]=



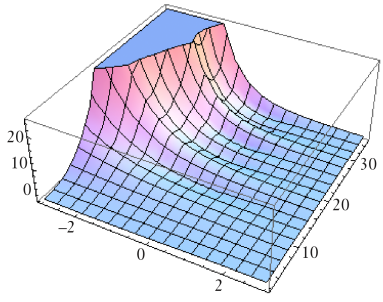
Mathematica gives exact results for $\zeta(2n)$.

```
In[1]:= Zeta[6]
Out[1]= π^6/945
```

Here is a three-dimensional picture of the Riemann zeta function in the complex plane.

```
In[14]:= Plot3D[Abs[Zeta[x + I y]], {x, -3, 3}, {y, 2, 35}]
```

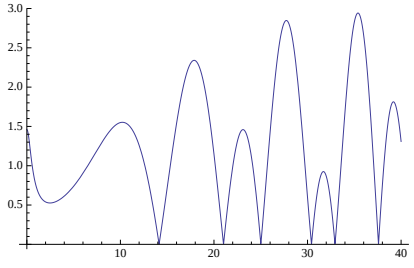
Out[14]=



This is a plot of the absolute value of the Riemann zeta function on the critical line $\mathrm{Re}\, z = \frac{1}{2}$. You can see the first few zeros of the zeta function.

```
In[15]:= Plot[Abs[Zeta[1 / 2 + I y]], {y, 0, 40}]
```

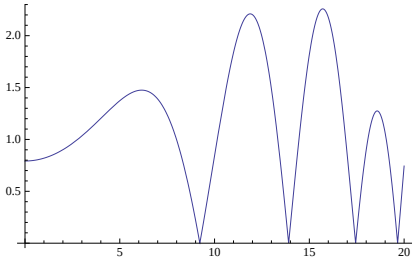
Out[15]=



This is a plot of the absolute value of the Ramanujan τ L function on its critical line $\mathrm{Re}\, z = 6$.

```
In[4]:= Plot[Abs[RamanujanTauL[6 + I y]], {y, 0, 20}]
```

Out[4]=



The *polylogarithm functions* `PolyLog[n, z]` are given by $\mathrm{Li}_n(z) = \sum_{k=1}^{\infty} z^k / k^n$. The polylogarithm function is sometimes known as *Jonquière's function*. The *dilogarithm* `PolyLog[2, z]` satisfies $\mathrm{Li}_2(z) = \int_z^0 \log(1-t)/t\, dt$. Sometimes $\mathrm{Li}_2(1-z)$ is known as *Spence's integral*. The *Nielsen generalized polylogarithm functions* or *hyperlogarithms* `PolyLog[n, p, z]` are given by

$S_{n,p}(z) = (-1)^{n+p-1} / ((n-1)!\,p!) \int_0^1 \log^{n-1}(t) \log^p(1-zt)/t\,dt$. Polylogarithm functions appear in Feynman diagram integrals in elementary particle physics, as well as in algebraic K-theory.

The *Lerch transcendent* `LerchPhi[z, s, a]` is a generalization of the zeta and polylogarithm functions, given by $\Phi(z, s, a) = \sum_{k=0}^{\infty} z^k / \left((a+k)^2\right)^{s/2}$, where any term with $a + k = 0$ is excluded. Many sums of reciprocal powers can be expressed in terms of the Lerch transcendent. For example, the *Catalan beta function* $\beta(s) = \sum_{k=0}^{\infty} (-1)^k (2k+1)^{-s}$ can be obtained as $2^{-s}\,\Phi\left(-1, s, \frac{1}{2}\right)$.

The Lerch transcendent is related to integrals of the *Fermi-Dirac* distribution in statistical mechanics by $\int_0^{\infty} k^s / \left(e^{k-\mu} + 1\right) dk = e^{\mu}\,\Gamma(s+1)\,\Phi(-e^{\mu}, s+1, 1)$.

The Lerch transcendent can also be used to evaluate *Dirichlet L-series* which appear in number theory. The basic L-series has the form $L(s, \chi) = \sum_{k=1}^{\infty} \chi(k)\,k^{-s}$, where the "character" $\chi(k)$ is an integer function with period m. L-series of this kind can be written as sums of Lerch functions with z a power of $e^{2\pi i/m}$.

`LerchPhi[z, s, a, DoublyInfinite -> True]` gives the doubly infinite sum $\sum_{k=-\infty}^{\infty} z^k / \left((a+k)^2\right)^{s/2}$.

The *Hurwitz-Lerch transcendent* `HurwitzLerchPhi[z, s, a]` generalizes `HurwitzZeta[s, a]` and is defined by $\Phi(z, s, a) = \sum_{k=0}^{\infty} z^k / ((a+k)^s)$.

`ZetaZero[k]`	the k^{th} zero of the zeta function $\zeta(z)$ on the critical line
`ZetaZero[k,x0]`	the k^{th} zero above height x_0

Zeros of the zeta function.

`ZetaZero[1]` represents the first nontrivial zero of $\zeta(z)$.

```
In[1]:= Zeta[ZetaZero[1]]
Out[1]= 0
```

This gives its numerical value.

```
In[2]:= N[ZetaZero[1]]
Out[2]= 0.5 + 14.1347 i
```

This gives the first zero with height greater than 15.

```
In[3]:= N[ZetaZero[1, 15]]
Out[3]= 0.5 + 21.022 i
```

Exponential Integral and Related Functions

`CosIntegral[z]`	cosine integral function $\mathrm{Ci}(z)$
`CoshIntegral[z]`	hyperbolic cosine integral function $\mathrm{Chi}(z)$
`ExpIntegralE[n,z]`	exponential integral $E_n(z)$
`ExpIntegralEi[z]`	exponential integral $\mathrm{Ei}(z)$
`LogIntegral[z]`	logarithmic integral $\mathrm{li}(z)$
`SinIntegral[z]`	sine integral function $\mathrm{Si}(z)$
`SinhIntegral[z]`	hyperbolic sine integral function $\mathrm{Shi}(z)$

Exponential integral and related functions.

Mathematica has two forms of exponential integral: `ExpIntegralE` and `ExpIntegralEi`.

The *exponential integral function* `ExpIntegralE[n, z]` is defined by $E_n(z) = \int_1^\infty e^{-z\,t}/t^n \, dt$.

The second *exponential integral function* `ExpIntegralEi[z]` is defined by $\mathrm{Ei}(z) = -\int_{-z}^\infty e^{-t}/t \, dt$ (for $z > 0$), where the principal value of the integral is taken.

The *logarithmic integral function* `LogIntegral[z]` is given by $\mathrm{li}(z) = \int_0^z dt/\log t$ (for $z > 1$), where the principal value of the integral is taken. $\mathrm{li}(z)$ is central to the study of the distribution of primes in number theory. The logarithmic integral function is sometimes also denoted by $\mathrm{Li}(z)$. In some number theoretic applications, $\mathrm{li}(z)$ is defined as $\int_2^z dt/\log t$, with no principal value taken. This differs from the definition used in *Mathematica* by the constant $\mathrm{li}(2)$.

The *sine and cosine integral functions* `SinIntegral[z]` and `CosIntegral[z]` are defined by $\mathrm{Si}(z) = \int_0^z \sin(t)/t \, dt$ and $\mathrm{Ci}(z) = -\int_z^\infty \cos(t)/t \, dt$. The *hyperbolic sine and cosine integral functions* `SinhIntegral[z]` and `CoshIntegral[z]` are defined by $\mathrm{Shi}(z) = \int_0^z \sinh(t)/t \, dt$ and $\mathrm{Chi}(z) = \gamma + \log(z) + \int_0^z (\cosh(t) - 1)/t \, dt$.

Error Function and Related Functions

`Erf[z]`	error function $\mathrm{erf}(z)$
`Erf[z0,z1]`	generalized error function $\mathrm{erf}(z_1) - \mathrm{erf}(z_0)$
`Erfc[z]`	complementary error function $\mathrm{erfc}(z)$
`Erfi[z]`	imaginary error function $\mathrm{erfi}(z)$
`FresnelC[z]`	Fresnel integral $C(z)$
`FresnelS[z]`	Fresnel integral $S(z)$
`InverseErf[s]`	inverse error function
`InverseErfc[s]`	inverse complementary error function

Error function and related functions.

The *error function* `Erf[z]` is the integral of the Gaussian distribution, given by $\mathrm{erf}(z) = 2/\sqrt{\pi}\ \int_0^z e^{-t^2}\,dt$. The *complementary error function* `Erfc[z]` is given simply by $\mathrm{erfc}(z) = 1 - \mathrm{erf}(z)$. The *imaginary error function* `Erfi[z]` is given by $\mathrm{erfi}(z) = \mathrm{erf}(iz)/i$. The generalized error function `Erf[z0, z1]` is defined by the integral $2/\sqrt{\pi}\ \int_{z_0}^{z_1} e^{-t^2}\,dt$. The error function is central to many calculations in statistics.

The *inverse error function* `InverseErf[s]` is defined as the solution for z in the equation $s = \mathrm{erf}(z)$. The inverse error function appears in computing confidence intervals in statistics as well as in some algorithms for generating Gaussian random numbers.

Closely related to the error function are the *Fresnel integrals* `FresnelC[z]` defined by $C(z) = \int_0^z \cos\left(\pi t^2/2\right) dt$ and `FresnelS[z]` defined by $S(z) = \int_0^z \sin\left(\pi t^2/2\right) dt$. Fresnel integrals occur in diffraction theory.

Bessel and Related Functions

`AiryAi[z]` and `AiryBi[z]`	Airy functions $\mathrm{Ai}(z)$ and $\mathrm{Bi}(z)$
`AiryAiPrime[z]` and `AiryBiPrime[z]`	derivatives of Airy functions $\mathrm{Ai}'(z)$ and $\mathrm{Bi}'(z)$
`BesselJ[n,z]` and `BesselY[n,z]`	Bessel functions $J_n(z)$ and $Y_n(z)$

`BesselI[n,z]` and `BesselK[n,z]`	modified Bessel functions $I_n(z)$ and $K_n(z)$
`KelvinBer[n,z]` and `KelvinBei[n,z]`	Kelvin functions $\text{ber}_n(z)$ and $\text{bei}_n(z)$
`KelvinKer[n,z]` and `KelvinKei[n,z]`	Kelvin functions $\text{ker}_n(z)$ and $\text{kei}_n(z)$
`HankelH1[n,z]` and `HankelH2[n,z]`	Hankel functions $H_n^{(1)}(z)$ and $H_n^{(2)}(z)$
`SphericalBesselJ[n,z]` and `SphericalBesselY[n,z]`	spherical Bessel functions $j_n(z)$ and $y_n(z)$
`SphericalHankelH1[n,z]` and `SphericalHankelH2[n,z]`	spherical Hankel functions $h_n^{(1)}(z)$ and $h_n^{(2)}(z)$
`StruveH[n,z]` and `StruveL[n,z]`	Struve function $\mathbf{H}_n(z)$ and modified Struve function $\mathbf{L}_n(z)$

Bessel and related functions.

The *Bessel functions* `BesselJ[n, z]` and `BesselY[n, z]` are linearly independent solutions to the differential equation $z^2 y'' + z y' + \left(z^2 - n^2\right) y = 0$. For integer n, the $J_n(z)$ are regular at $z = 0$, while the $Y_n(z)$ have a logarithmic divergence at $z = 0$.

Bessel functions arise in solving differential equations for systems with cylindrical symmetry.

$J_n(z)$ is often called the *Bessel function of the first kind*, or simply *the* Bessel function. $Y_n(z)$ is referred to as the *Bessel function of the second kind*, the *Weber function*, or the *Neumann function* (denoted $N_n(z)$).

The *Hankel functions* (or *Bessel functions of the third kind*) `HankelH1[n, z]` and `HankelH2[n, z]` give an alternative pair of solutions to the Bessel differential equation, related according to $H_n^{(1,2)}(z) = J_n(z) \pm i Y_n(z)$.

The *spherical Bessel functions* `SphericalBesselJ[n, z]` and `SphericalBesselY[n, z]`, as well as the *spherical Hankel functions* `SphericalHankelH1[n, z]` and `SphericalHankelH2[n, z]`, arise in studying wave phenomena with spherical symmetry. These are related to the ordinary

functions by $f_n(z) = \sqrt{\pi/2} \Big/ \sqrt{z}\, F_{n+\frac{1}{2}}(z)$, where f and F can be j and J, y and Y, or h^i and H^i. For integer n, spherical Bessel functions can be expanded in terms of elementary functions by using `FunctionExpand`.

The *modified Bessel functions* `BesselI[n, z]` and `BesselK[n, z]` are solutions to the differential equation $z^2\, y'' + z\, y' - \left(z^2 + n^2\right) y = 0$. For integer n, $I_n(z)$ is regular at $z = 0$; $K_n(z)$ always has a logarithmic divergence at $z = 0$. The $I_n(z)$ are sometimes known as *hyperbolic Bessel functions*.

Particularly in electrical engineering, one often defines the *Kelvin functions* `KelvinBer[n, z]`, `KelvinBei[n, z]`, `KelvinKer[n, z]` and `KelvinKei[n, z]`. These are related to the ordinary Bessel functions by $\mathrm{ber}_n(z) + i\,\mathrm{bei}_n(z) = e^{n\pi i}\, J_n\left(z\, e^{-\pi i/4}\right)$, $\mathrm{ker}_n(z) + i\,\mathrm{kei}_n(z) = e^{-n\pi i/2}\, K_n\left(z\, e^{\pi i/4}\right)$.

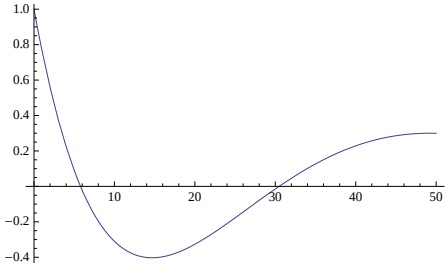
The *Airy functions* `AiryAi[z]` and `AiryBi[z]` are the two independent solutions $\mathrm{Ai}(z)$ and $\mathrm{Bi}(z)$ to the differential equation $y'' - z\, y = 0$. $\mathrm{Ai}(z)$ tends to zero for large positive z, while $\mathrm{Bi}(z)$ increases unboundedly. The Airy functions are related to modified Bessel functions with one-third-integer orders. The Airy functions often appear as the solutions to boundary value problems in electromagnetic theory and quantum mechanics. In many cases the *derivatives of the Airy functions* `AiryAiPrime[z]` and `AiryBiPrime[z]` also appear.

The *Struve function* `StruveH[n, z]` appears in the solution of the inhomogeneous Bessel equation which for integer n has the form $z^2\, y'' + z\, y' + \left(z^2 - n^2\right) y = \frac{2}{\pi}\, \frac{z^{n+1}}{(2n-1)!!}$; the general solution to this equation consists of a linear combination of Bessel functions with the Struve function $\mathbf{H}_n(z)$ added. The *modified Struve function* `StruveL[n, z]` is given in terms of the ordinary Struve function by $\mathbf{L}_n(z) = -i\, e^{-i n \pi/2}\, \mathbf{H}_n(z)$. Struve functions appear particularly in electromagnetic theory.

Here is a plot of $J_0\left(\sqrt{x}\right)$. This is a curve that an idealized chain hanging from one end can form when you wiggle it.

```
In[16]:= Plot[BesselJ[0, Sqrt[x]], {x, 0, 50}]
```

Out[16]=



Mathematica generates explicit formulas for half-integer-order Bessel functions.

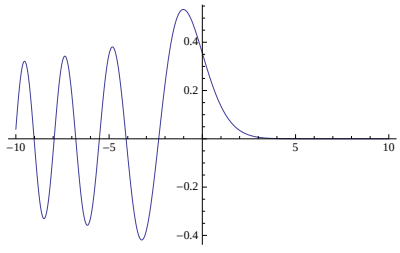
```
In[2]:= BesselK[3 / 2, x]
```

$$\text{Out[2]}= \frac{e^{-x}\sqrt{\frac{\pi}{2}}\left(1+\frac{1}{x}\right)}{\sqrt{x}}$$

The Airy function plotted here gives the quantum-mechanical amplitude for a particle in a potential that increases linearly from left to right. The amplitude is exponentially damped in the classically inaccessible region on the right.

```
In[17]:= Plot[AiryAi[x], {x, -10, 10}]
```

Out[17]=



`BesselJZero[n,k]`	the k^{th} zero of the Bessel function $J_n(z)$
`BesselJZero[n,k,x0]`	the k^{th} zero greater than x_0
`BesselYZero[n,k]`	the k^{th} zero of the Bessel function $Y_n(z)$
`BesselYZero[n,k,x0]`	the k^{th} zero greater than x_0
`AiryAiZero[k]`	the k^{th} zero of the Airy function $\text{Ai}(z)$
`AiryAiZero[k,x0]`	the k^{th} zero less than x_0
`AiryBiZero[k]`	the k^{th} zero of the Airy function $\text{Bi}(z)$
`AiryBiZero[k,x0]`	the k^{th} zero less than x_0

Zeros of Bessel and Airy functions.

`BesselJZero[1, 5]` represents the fifth zero of $J_1(z)$.

```
In[18]:= BesselJ[1, BesselJZero[1, 5]]
Out[18]= 0
```

This gives its numerical value.

```
In[19]:= N[BesselJZero[1, 5]]
Out[19]= 16.4706
```

Spheroidal Functions

`SpheroidalS1[n,m,γ,z]` and `SpheroidalS2[n,m,γ,z]`	radial spheroidal functions $S^{(1)}_{n,m}(\gamma, z)$ and $S^{(2)}_{n,m}(\gamma, z)$
`SpheroidalS1Prime[n,m,γ,z]` and `SpheroidalS2Prime[n,m,γ,z]`	z derivatives of radial spheroidal functions
`SpheroidalPS[n,m,γ,z]` and `SpheroidalQS[n,m,γ,z]`	angular spheroidal functions $PS_{n,m}(\gamma, z)$ and $QS_{n,m}(\gamma, z)$
`SpheroidalPSPrime[n,m,γ,z]` and `SpheroidalQSPrime[n,m,γ,z]`	z derivatives of angular spheroidal functions
`SpheroidalEigenvalue[n,m,γ]`	spheroidal eigenvalue of degree n and order m

Spheroidal functions.

The *radial spheroidal functions* `SpheroidalS1[n, m, γ, z]` and `SpheroidalS2[n, m, γ, z]` and *angular spheroidal functions* `SpheroidalPS[n, m, γ, z]` and `SpheroidalQS[n, m, γ, z]` appear in solutions to the wave equation in spheroidal regions. Both types of functions are solutions to the equation $\left(1-z^2\right) y'' - 2\,z y' + \left(\lambda + \gamma^2\left(1-z^2\right) - \frac{m^2}{1-z^2}\right) y == 0$. This equation has normalizable solutions only when λ is a *spheroidal eigenvalue* given by `SpheroidalEigenvalue[n, m, γ]`. The spheroidal functions also appear as eigenfunctions of finite analogs of Fourier transforms.

`SpheroidalS1` and `SpheroidalS2` are effectively spheroidal analogs of the spherical Bessel functions $j_n(z)$ and $y_n(z)$, while `SpheroidalPS` and `SpheroidalQS` are effectively spheroidal analogs of the Legendre functions $P_n^m(z)$ and $Q_n^m(z)$. $\gamma^2 > 0$ corresponds to a *prolate spheroidal* geometry, while $\gamma^2 < 0$ corresponds to an *oblate spheroidal* geometry.

function		γ	z	*range*	*name*
$PS_{n,m}(\gamma, \eta)$	$QS_{n,m}(\gamma, \eta)$	γ	η	$-1 \le \eta \le 1$	angular prolate
$S^{(1)}_{n,m}(\gamma, \zeta)$	$S^{(2)}_{n,m}(\gamma, \zeta)$	γ	ζ	$\zeta \ge 1$	radial prolate
$PS_{n,m}(-i\,\gamma, \eta)$	$QS_{n,m}(-i\,\gamma, \eta)$	$-i\,\gamma$	η	$-1 \le \eta \le 1$	angular oblate
$S^{(1)}_{n,m}(-i\,\gamma, \zeta)$	$S^{(2)}_{n,m}(-i\,\gamma, \zeta)$	$-i\,\gamma$	$i\,\zeta$	$\zeta \ge 0$	radial oblate

Many different normalizations for spheroidal functions are used in the literature. *Mathematica* uses the Meixner-Schäfke normalization scheme.

Angular spheroidal functions can be viewed as deformations of Legendre functions.

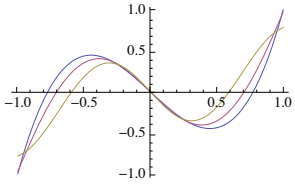
```
In[1]:= Series[SpheroidalPS[n, 0, γ, η], {γ, 0, 3}]
```

$$\text{Out[1]= LegendreP}[n, \eta] + \left(-\frac{(-1+n)\, n\, \text{LegendreP}[-2+n, \eta]}{2\,(-1+2n)^2\,(1+2n)} + \frac{(1+n)\,(2+n)\,\text{LegendreP}[2+n, \eta]}{2\,(1+2n)\,(3+2n)^2}\right)\gamma^2 + O[\gamma]^3$$

This plots angular spheroidal functions for various spheroidicity parameters.

```
In[2]:= Plot[{SpheroidalPS[3, 0, 0, η],
   SpheroidalPS[3, 0, 3, η], SpheroidalPS[3, 0, 5, η]}, {η, -1, 1}]
```

Out[2]=



Angular spheroidal functions $PS_{n,0}(\gamma, \eta)$ for integers $n \geq 0$ are eigenfunctions of a band-limited Fourier transform.

```
In[3]:= Integrate[SpheroidalPS[3, 0, γ, η] Exp[i ω γ η], {η, -1, 1}]
Out[3]= -2 i SpheroidalPS[3, 0, γ, ω] SpheroidalS1[3, 0, γ, 1]
```

The Mathieu functions are a special case of spheroidal functions.

An angular spheroidal function with $m = \frac{1}{2}$ gives Mathieu angular functions.

```
In[4]:= SpheroidalPS[1 / 2, 1 / 2, c, z]
```

$$\text{Out[4]= MathieuC}\left[\text{MathieuCharacteristicA}\left[1, \frac{c^2}{4}\right], \frac{c^2}{4}, \text{ArcCos}[z]\right] \Big/ \left(\sqrt{\pi}\,\left(1 - z^2\right)^{1/4}\right)$$

Legendre and Related Functions

`LegendreP[n,z]`	Legendre functions of the first kind $P_n(z)$
`LegendreP[n,m,z]`	associated Legendre functions of the first kind $P_n^m(z)$
`LegendreQ[n,z]`	Legendre functions of the second kind $Q_n(z)$
`LegendreQ[n,m,z]`	associated Legendre functions of the second kind $Q_n^m(z)$

Legendre and related functions.

The *Legendre functions* and *associated Legendre functions* satisfy the differential equation $\left(1-z^2\right) y'' - 2\,z\,y' + \left[n\,(n+1) - m^2/\left(1-z^2\right)\right] y = 0$. The *Legendre functions of the first kind*, `LegendreP[n, z]` and `LegendreP[n, m, z]`, reduce to Legendre polynomials when n and m are integers. The *Legendre functions of the second kind* `LegendreQ[n, z]` and `LegendreQ[n, m, z]` give the second linearly independent solution to the differential equation. For integer m they have logarithmic singularities at $z = \pm 1$. The $P_n(z)$ and $Q_n(z)$ solve the differential equation with $m = 0$.

Legendre functions arise in studies of quantum-mechanical scattering processes.

`LegendreP[n,m,z]` or `LegendreP[n,m,1,z]`	type 1 function containing $\left(1-z^2\right)^{m/2}$
`LegendreP[n,m,2,z]`	type 2 function containing $(1+z)^{m/2}/(1-z)^{m/2}$
`LegendreP[n,m,3,z]`	type 3 function containing $(1+z)^{m/2}/(-1+z)^{m/2}$

Types of Legendre functions. Analogous types exist for `LegendreQ`.

Legendre functions of type 1 and *Legendre functions of type 2* have different symbolic forms, but the same numerical values. They have branch cuts from $-\infty$ to -1 and from $+1$ to $+\infty$. *Legendre functions of type 3*, sometimes denoted $\mathcal{P}_n^m(z)$ and $\mathcal{Q}_n^m(z)$, have a single branch cut from $-\infty$ to $+1$.

Toroidal functions or *ring functions*, which arise in studying systems with toroidal symmetry, can be expressed in terms of the Legendre functions $P^{\mu}_{\nu-\frac{1}{2}}(\cosh\eta)$ and $Q^{\mu}_{\nu-\frac{1}{2}}(\cosh\eta)$.

Conical functions can be expressed in terms of $P^{\mu}_{-\frac{1}{2}+i\,p}(\cos\theta)$ and $Q^{\mu}_{-\frac{1}{2}+i\,p}(\cos\theta)$.

When you use the function `LegendreP[n, x]` with an integer n, you get a Legendre polynomial. If you take n to be an arbitrary complex number, you get, in general, a Legendre function.

In the same way, you can use the functions `GegenbauerC` and so on with arbitrary complex indices to get *Gegenbauer functions*, *Chebyshev functions*, *Hermite functions*, *Jacobi functions* and *Laguerre functions*. Unlike for associated Legendre functions, however, there is no need to distinguish different types in such cases.

Hypergeometric Functions and Generalizations

`Hypergeometric0F1[a,z]`	hypergeometric function ${}_0F_1(;a;z)$
`Hypergeometric0F1Regularized[a,z]`	regularized hypergeometric function ${}_0F_1(;a;z)/\Gamma(a)$
`Hypergeometric1F1[a,b,z]`	Kummer confluent hypergeometric function ${}_1F_1(a;b;z)$
`Hypergeometric1F1Regularized[a,b,z]`	regularized confluent hypergeometric function ${}_1F_1(a;b;z)/\Gamma(b)$
`HypergeometricU[a,b,z]`	confluent hypergeometric function $U(a, b, z)$
`WhittakerM[k,m,z]` and `WhittakerW[k,m,z]`	Whittaker functions $M_{k,m}(z)$ and $W_{k,m}(z)$
`ParabolicCylinderD[ν,z]`	parabolic cylinder function $D_\nu(z)$

Confluent hypergeometric functions and related functions.

Many of the special functions that we have discussed so far can be viewed as special cases of the *confluent hypergeometric function* `Hypergeometric1F1[a, b, z]`.

The confluent hypergeometric function can be obtained from the series expansion ${}_1F_1(a;b;z) = 1 + az/b + a(a+1)/b(b+1)z^2/2! + \cdots = \sum_{k=0}^{\infty}(a)_k/(b)_k z^k/k!$. Some special results are obtained when a and b are both integers. If $a<0$, and either $b>0$ or $b<a$, the series yields a polynomial with a finite number of terms.

If b is zero or a negative integer, then ${}_1F_1(a;b;z)$ itself is infinite. But the *regularized confluent hypergeometric function* `Hypergeometric1F1Regularized[a, b, z]` given by ${}_1F_1(a;b;z)/\Gamma(b)$ has a finite value in all cases.

Among the functions that can be obtained from ${}_1F_1$ are the Bessel functions, error function, incomplete gamma function, and Hermite and Laguerre polynomials.

The function ${}_1F_1(a;b;z)$ is sometimes denoted $\Phi(a;b;z)$ or $M(a, b, z)$. It is often known as the *Kummer function*.

The ${}_1F_1$ function can be written in the integral representation ${}_1F_1(a;b;z) = \Gamma(b)/[\Gamma(b-a)\Gamma(a)]\int_0^1 e^{zt}\, t^{a-1}\, (1-t)^{b-a-1}\, dt$.

The ${}_1F_1$ confluent hypergeometric function is a solution to Kummer's differential equation $z\,y'' + (b-z)\,y' - a\,y = 0$, with the boundary conditions ${}_1F_1(a; b; 0) = 1$ and $\partial[{}_1F_1(a; b; z)]/\partial z\,|_{z=0} = a/b$.

The function `HypergeometricU[a, b, z]` gives a second linearly independent solution to Kummer's equation. For $\mathrm{Re}\,b > 1$ this function behaves like z^{1-b} for small z. It has a branch cut along the negative real axis in the complex z plane.

The function $U(a, b, z)$ has the integral representation $U(a, b, z) = 1/\Gamma(a) \int_0^\infty e^{-z\,t}\,t^{a-1}\,(1+t)^{b-a-1}\,d\,t$.

$U(a, b, z)$, like ${}_1F_1(a; b; z)$, is sometimes known as the *Kummer function*. The U function is sometimes denoted by Ψ.

The *Whittaker functions* `WhittakerM[k, m, z]` and `WhittakerW[k, m, z]` give a pair of solutions to the normalized Kummer differential equation, known as Whittaker's differential equation. The Whittaker function $M_{\kappa,\mu}$ is related to ${}_1F_1$ by $M_{\kappa,\mu}(z) = e^{-z/2}\,z^{1/2+\mu}\,{}_1F_1\left(\frac{1}{2}+\mu-\kappa; 1+2\,\mu; z\right)$. The second Whittaker function $W_{\kappa,\mu}$ obeys the same relation, with ${}_1F_1$ replaced by U.

The *parabolic cylinder functions* `ParabolicCylinderD[ν, z]` are related to the Hermite functions by $D_\nu(z) = 2^{-\nu/2}\,e^{-(z/2)^2} \times H_\nu\left(z\Big/\sqrt{2}\right)$.

The *Coulomb wave functions* are also special cases of the confluent hypergeometric function. Coulomb wave functions give solutions to the radial Schrödinger equation in the Coulomb potential of a point nucleus. The regular Coulomb wave function is given by $F_L(\eta, \rho) = C_L(\eta)\,\rho^{L+1}\,e^{-i\,\rho}\,{}_1F_1(L+1-i\,\eta; 2\,L+2; 2\,i\,\rho)$, where $C_L(\eta) = 2^L\,e^{-\pi\eta/2}\,\big|\,\Gamma(L+1+i\,\eta)\,\big|\,/\Gamma(2\,L+2)$.

Other special cases of the confluent hypergeometric function include the *Toronto functions* $T(m, n, r)$, *Poisson-Charlier polynomials* $\rho_n(\nu, x)$, *Cunningham functions* $\omega_{n,m}(x)$ and *Bateman functions* $k_\nu(x)$.

A limiting form of the confluent hypergeometric function which often appears is `Hypergeometric0F1[a, z]`. This function is obtained as the limit ${}_0F_1(;a;z) = \lim_{q\to\infty} {}_1F_1(q; a; z/q)$.

The ${}_0F_1$ function has the series expansion ${}_0F_1(;a;z) = \sum_{k=0}^\infty 1/(a)_k\,z^k\Big/k!$ and satisfies the differential equation $z\,y'' + a\,y' - y = 0$.

Bessel functions of the first kind can be expressed in terms of the ${}_0F_1$ function.

`Hypergeometric2F1[a,b,c,z]`	hypergeometric function ${}_2F_1(a, b; c; z)$
`Hypergeometric2F1Regularized[a,b,c,z]`	regularized hypergeometric function ${}_2F_1(a, b; c; z)/\Gamma(c)$
`HypergeometricPFQ[{a1,...,ap},{b1,...,bq},z]`	generalized hypergeometric function ${}_pF_q(a; b; z)$
`HypergeometricPFQRegularized[{a1,...,ap},{b1,...,bq},z]`	regularized generalized hypergeometric function
`MeijerG[{{a1,...,an},{an+1,...,ap}},{{b1,...,bm},{bm+1,...,bq}},z]`	Meijer G function
`AppellF1[a,b1,b2,c,x,y]`	Appell hypergeometric function of two variables $F_1(a; b_1, b_2; c; x, y)$

Hypergeometric functions and generalizations.

The *hypergeometric function* `Hypergeometric2F1[a, b, c, z]` has series expansion ${}_2F_1(a, b; c; z) = \sum_{k=0}^{\infty} (a)_k (b)_k / (c)_k \, z^k / k!$. The function is a solution of the hypergeometric differential equation $z(1-z)\,y'' + [c - (a+b+1)\,z]\,y' - a\,b\,y = 0$.

The hypergeometric function can also be written as an integral: ${}_2F_1(a, b; c; z) = \Gamma(c)/[\Gamma(b)\,\Gamma(c-b)] \times \int_0^1 t^{b-1} (1-t)^{c-b-1} (1-t\,z)^{-a}\, d\,t$.

The hypergeometric function is also sometimes denoted by F, and is known as the *Gauss series* or the *Kummer series*.

The Legendre functions, and the functions which give generalizations of other orthogonal polynomials, can be expressed in terms of the hypergeometric function. Complete elliptic integrals can also be expressed in terms of the ${}_2F_1$ function.

The *Riemann P function*, which gives solutions to Riemann's differential equation, is also a ${}_2F_1$ function.

The *generalized hypergeometric function* or *Barnes extended hypergeometric function* `HypergeometricPFQ[{a1, ..., ap}, {b1, ..., bq}, z]` has series expansion ${}_p F_q(a; b; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k \ldots (a_p)_k}{(b_1)_k \ldots (b_q)_k} \frac{z^k}{k!}$.

The *Meijer G function* `MeijerG[{{a1, ..., an}, {an + 1, ..., ap}}, {{b1, ..., bm}, {bm + 1, ..., bq}}, z]` is defined by the contour integral representation $G_{p\,q}^{m\,n}\left(z \,\middle|\, \begin{matrix} a_1, \ldots, a_p \\ b_1, \ldots, b_q \end{matrix}\right) = \frac{1}{2\pi i} \int \Gamma(1 - a_1 - s) \ldots \Gamma(1 - a_n - s) \times \Gamma(b_1 + s) \ldots \Gamma(b_m + s) \Big/ \left(\Gamma(a_{n+1} + s) \ldots \Gamma\left(a_p + s\right)\right.$ $\left.\Gamma(1 - b_{m+1} - s) \ldots \Gamma\left(1 - b_q - s\right)\right) z^{-s}\, ds$, where the contour of integration is set up to lie between the poles of $\Gamma(1 - a_i - s)$ and the poles of $\Gamma(b_i + s)$. `MeijerG` is a very general function whose special cases cover most of the functions discussed in the past few sections.

The *Appell hypergeometric function of two variables* `AppellF1[a, b1, b2, c, x, y]` has series expansion $F_1(a; b_1, b_2; c; x, y) = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} (a)_{m+n} (b_1)_m (b_2)_n / (m!\, n!\, (c)_{m+n})\, x^m\, y^n$. This function appears for example in integrating cubic polynomials to arbitrary powers.

The Product Log Function

`ProductLog[z]`	product log function $W(z)$

The product log function.

The *product log function* gives the solution for w in $z = w\, e^w$. The function can be viewed as a generalization of a logarithm. It can be used to represent solutions to a variety of transcendental equations. The *tree generating function* for counting distinct oriented trees is related to the product log by $T(z) = -W(-z)$.

Elliptic Integrals and Elliptic Functions

Even more so than for other special functions, you need to be very careful about the arguments you give to elliptic integrals and elliptic functions. There are several incompatible conventions in common use, and often these conventions are distinguished only by the specific names given to arguments or by the presence of separators other than commas between arguments.

- Amplitude ϕ (used by *Mathematica*, in radians)
- Argument u (used by *Mathematica*): related to amplitude by $\phi = \text{am}(u)$
- Delta amplitude $\Delta(\phi)$: $\Delta(\phi) = \sqrt{1 - m \sin^2(\phi)}$
- Coordinate x: $x = \sin(\phi)$
- Characteristic n (used by *Mathematica* in elliptic integrals of the third kind)
- Parameter m (used by *Mathematica*): preceded by |, as in $I(\phi \mid m)$
- Complementary parameter m_1: $m_1 = 1 - m$
- Modulus k: preceded by comma, as in $I(\phi, k)$; $m = k^2$
- Modular angle α: preceded by \, as in $I(\phi \backslash \alpha)$; $m = \sin^2(\alpha)$
- Nome q: preceded by comma in θ functions; $q = \exp[-\pi K(1-m)/K\{m\}] = \exp(i\pi\omega'/\omega)$
- Invariants g_2, g_3 (used by *Mathematica*)
- Half-periods ω, ω': $g_2 = 60 \sum'_{r,s} w^{-4}$, $g_3 = 140 \sum'_{r,s} w^{-6}$, where $w = 2r\omega + 2s\omega'$
- Ratio of periods τ: $\tau = \omega'/\omega$
- Discriminant Δ: $\Delta = g_2^3 - 27 g_3^2$
- Parameters of curve a, b (used by *Mathematica*)
- Coordinate y (used by *Mathematica*): related by $y^2 = x^3 + a x^2 + b x$

Common argument conventions for elliptic integrals and elliptic functions.

`JacobiAmplitude[u,m]`	give the amplitude ϕ corresponding to argument u and parameter m
`EllipticNomeQ[m]`	give the nome q corresponding to parameter m
`InverseEllipticNomeQ[q]`	give the parameter m corresponding to nome q
`WeierstrassInvariants[{ω,ω'}]`	give the invariants $\{g_2, g_3\}$ corresponding to the half-periods $\{\omega, \omega'\}$
`WeierstrassHalfPeriods[{g2,g3}]`	give the half-periods $\{\omega, \omega'\}$ corresponding to the invariants $\{g_2, g_3\}$

Converting between different argument conventions.

Elliptic Integrals

`EllipticK[m]`	complete elliptic integral of the first kind $K(m)$
`EllipticF[ϕ,m]`	elliptic integral of the first kind $F(\phi \mid m)$
`EllipticE[m]`	complete elliptic integral of the second kind $E(m)$
`EllipticE[ϕ,m]`	elliptic integral of the second kind $E(\phi \mid m)$
`EllipticPi[n,m]`	complete elliptic integral of the third kind $\Pi(n \mid m)$
`EllipticPi[n,ϕ,m]`	elliptic integral of the third kind $\Pi(n; \phi \mid m)$
`JacobiZeta[ϕ,m]`	Jacobi zeta function $Z(\phi \mid m)$

Elliptic integrals.

Integrals of the form $\int R(x, y)\, dx$, where R is a rational function, and y^2 is a cubic or quartic polynomial in x, are known as *elliptic integrals*. Any elliptic integral can be expressed in terms of the three standard kinds of *Legendre-Jacobi elliptic integrals*.

The *elliptic integral of the first kind* `EllipticF[ϕ, m]` is given for $-\pi/2 < \phi < \pi/2$ by $F(\phi \mid m) = \int_0^{\phi} \left[1 - m \sin^2(\theta)\right]^{-1/2} d\theta = \int_0^{\sin(\phi)} \left[\left(1 - t^2\right)\left(1 - m\, t^2\right)\right]^{-1/2} dt$. This elliptic integral arises in solving the equations of motion for a simple pendulum. It is sometimes known as an *incomplete elliptic integral of the first kind*.

Note that the arguments of the elliptic integrals are sometimes given in the opposite order from what is used in *Mathematica*.

The *complete elliptic integral of the first kind* `EllipticK[m]` is given by $K(m) = F\left(\frac{\pi}{2} \,\middle|\, m\right)$. Note that K is used to denote the *complete* elliptic integral of the first kind, while F is used for its incomplete form. In many applications, the parameter m is not given explicitly, and $K(m)$ is denoted simply by K. The *complementary complete elliptic integral of the first kind* $K'(m)$ is given by $K(1 - m)$. It is often denoted K'. K and $i\,K'$ give the "real" and "imaginary" quarter-periods of the corresponding Jacobi elliptic functions discussed in "Elliptic Functions".

The *elliptic integral of the second kind* `EllipticE[ϕ, m]` is given for $-\pi/2 < \phi < \pi/2$ by $E(\phi \mid m) = \int_0^{\phi} \left[1 - m \sin^2(\theta)\right]^{1/2} d\theta = \int_0^{\sin(\phi)} \left(1 - t^2\right)^{-1/2} \left(1 - m\, t^2\right)^{1/2} dt$.

The *complete elliptic integral of the second kind* `EllipticE[m]` is given by $E(m) = E\left(\frac{\pi}{2} \,\middle|\, m\right)$. It is often denoted E. The complementary form is $E'(m) = E(1 - m)$.

The *Jacobi zeta function* `JacobiZeta[ϕ, m]` is given by $Z(\phi \mid m) = E(\phi \mid m) - E(m) F(\phi \mid m)/K(m)$.

The *Heuman lambda function* is given by $\Lambda_0(\phi \mid m) = F(\phi \mid 1-m)/K(1-m) + \frac{2}{\pi} K(m) Z(\phi \mid 1-m)$.

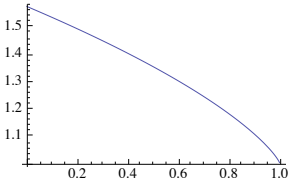
The *elliptic integral of the third kind* `EllipticPi[n, ϕ, m]` is given by $\Pi(n; \phi \mid m) = \int_0^{\phi} \left(1 - n \sin^2(\theta)\right)^{-1} \left[1 - m \sin^2(\theta)\right]^{-1/2} d\theta$.

The *complete elliptic integral of the third kind* `EllipticPi[n, m]` is given by $\Pi(n \mid m) = \Pi\left(n; \frac{\pi}{2} \middle| m\right)$.

Here is a plot of the complete elliptic integral of the second kind $E(m)$.

```
In[1]:= Plot[EllipticE[m], {m, 0, 1}]
```

Out[1]=



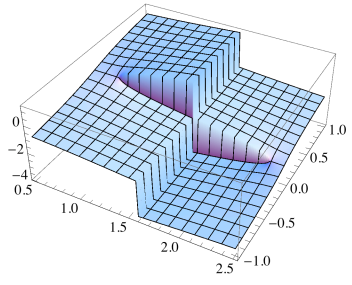
Here is $K(\alpha)$ with $\alpha = 30^\circ$.

```
In[2]:= EllipticK[Sin[30 Degree]^2] // N
Out[2]= 1.68575
```

The elliptic integrals have a complicated structure in the complex plane.

```
In[1]:= Plot3D[Im[EllipticF[px + I py, 2]], {px, 0.5, 2.5}, {py, -1, 1}, PlotPoints -> 60]
```

Out[1]=



Elliptic Functions

`JacobiAmplitude[u,m]`	amplitude function $\mathrm{am}(u \mid m)$
`JacobiSN[u,m]` , `JacobiCN[u,m]` , etc.	Jacobi elliptic functions $\mathrm{sn}(u \mid m)$, etc.
`InverseJacobiSN[v,m]` , `InverseJacobiCN[v,m]` , etc.	inverse Jacobi elliptic functions $\mathrm{sn}^{-1}(v \mid m)$, etc.
`EllipticTheta[a,u,q]`	theta functions $\vartheta_a(u, q)$ $(a = 1, \ldots, 4)$
`EllipticThetaPrime[a,u,q]`	derivatives of theta functions $\vartheta'_a(u, q)$ $(a = 1, \ldots, 4)$
`SiegelTheta[τ,s]`	Siegel theta function $\Theta(\tau, s)$
`SiegelTheta[v,τ,s]`	Siegel theta function $\Theta[v](\tau, s)$
`WeierstrassP[u,{g2,g3}]`	Weierstrass elliptic function $\wp(u; g_2, g_3)$
`WeierstrassPPrime[u,{g2,g3}]`	derivative of Weierstrass elliptic function $\wp'(u; g_2, g_3)$
`InverseWeierstrassP[p,{g2,g3}]`	inverse Weierstrass elliptic function
`WeierstrassSigma[u,{g2,g3}]`	Weierstrass sigma function $\sigma(u; g_2, g_3)$
`WeierstrassZeta[u,{g2,g3}]`	Weierstrass zeta function $\zeta(u; g_2, g_3)$

Elliptic and related functions.

Rational functions involving square roots of quadratic forms can be integrated in terms of inverse trigonometric functions. The trigonometric functions can thus be defined as inverses of the functions obtained from these integrals.

By analogy, *elliptic functions* are defined as inverses of the functions obtained from elliptic integrals.

The *amplitude* for Jacobi elliptic functions `JacobiAmplitude[u, m]` is the inverse of the elliptic integral of the first kind. If $u = F(\phi \mid m)$, then $\phi = \mathrm{am}(u \mid m)$. In working with Jacobi elliptic functions, the argument m is often dropped, so $\mathrm{am}(u \mid m)$ is written as $\mathrm{am}(u)$.

The *Jacobi elliptic functions* `JacobiSN[u, m]` and `JacobiCN[u, m]` are given respectively by $\mathrm{sn}(u) = \sin(\phi)$ and $\mathrm{cn}(u) = \cos(\phi)$, where $\phi = \mathrm{am}(u \mid m)$. In addition, `JacobiDN[u, m]` is given by $\mathrm{dn}(u) = \sqrt{1 - m \sin^2(\phi)} = \Delta(\phi)$.

There are a total of twelve Jacobi elliptic functions `JacobiPQ[u, m]`, with the letters *P* and *Q* chosen from the set `S`, `C`, `D` and `N`. Each Jacobi elliptic function `JacobiPQ[u, m]` satisfies the relation $\mathrm{pq}(u) = \mathrm{pn}(u)/\mathrm{qn}(u)$, where for these purposes $\mathrm{nn}(u) = 1$.

There are many relations between the Jacobi elliptic functions, somewhat analogous to those between trigonometric functions. In limiting cases, in fact, the Jacobi elliptic functions reduce to trigonometric functions. So, for example, $\mathrm{sn}(u \mid 0) = \sin(u)$, $\mathrm{sn}(u \mid 1) = \tanh(u)$, $\mathrm{cn}(u \mid 0) = \cos(u)$, $\mathrm{cn}(u \mid 1) = \mathrm{sech}(u)$, $\mathrm{dn}(u \mid 0) = 1$ and $\mathrm{dn}(u \mid 1) = \mathrm{sech}(u)$.

The notation $\mathrm{Pq}(u)$ is often used for the integrals $\int_0^u \mathrm{pq}^2(t)\,dt$. These integrals can be expressed in terms of the Jacobi zeta function defined in "Elliptic Integrals".

One of the most important properties of elliptic functions is that they are *doubly periodic* in the complex values of their arguments. Ordinary trigonometric functions are singly periodic, in the sense that $f(z + s\omega) = f(z)$ for any integer s. The elliptic functions are doubly periodic, so that $f(z + r\omega + s\omega') = f(z)$ for any pair of integers r and s.

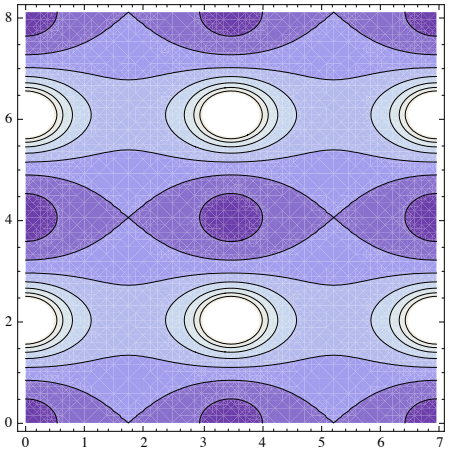
The Jacobi elliptic functions $\mathrm{sn}(u \mid m)$, etc. are doubly periodic in the complex u plane. Their periods include $\omega = 4\,K(m)$ and $\omega' = 4\,iK(1 - m)$, where K is the complete elliptic integral of the first kind.

The choice of p and q in the notation $\mathrm{pq}(u \mid m)$ for Jacobi elliptic functions can be understood in terms of the values of the functions at the quarter periods K and $i\,K'$.

This shows two complete periods in each direction of the absolute value of the Jacobi elliptic function $\mathrm{sn}\left(u \mid \frac{1}{3}\right)$.

```
In[3]:= ContourPlot[Abs[JacobiSN[ux + I uy, 1 / 3]],
          {ux, 0, 4 EllipticK[1 / 3]}, {uy, 0, 4 EllipticK[2 / 3]}]
```

Out[3]=



Also built into *Mathematica* are the *inverse Jacobi elliptic functions* `InverseJacobiSN[v, m]`, `InverseJacobiCN[v, m]`, etc. The inverse function $\text{sn}^{-1}(v \mid m)$, for example, gives the value of u for which $v = \text{sn}(u \mid m)$. The inverse Jacobi elliptic functions are related to elliptic integrals.

The four *theta functions* $\vartheta_a(u, q)$ are obtained from `EllipticTheta[a, u, q]` by taking a to be `1`, `2`, `3` or `4`. The functions are defined by: $\vartheta_1(u, q) = 2\, q^{1/4} \sum_{n=0}^{\infty} (-1)^n\, q^{n(n+1)} \sin[(2n+1)\, u]$, $\vartheta_2(u, q) = 2\, q^{1/4} \sum_{n=0}^{\infty} q^{n(n+1)} \cos[(2n+1)\, u]$, $\vartheta_3(u, q) = 1 + 2 \sum_{n=1}^{\infty} q^{n^2} \cos(2nu)$, $\vartheta_4(u, q) = 1 + 2 \sum_{n=1}^{\infty} (-1)^n\, q^{n^2} \cos(2nu)$. The theta functions are often written as $\vartheta_a(u)$ with the parameter q not explicitly given. The theta functions are sometimes written in the form $\vartheta(u \mid m)$, where m is related to q by $q = \exp[-\pi K(1-m)/K(m)]$. In addition, q is sometimes replaced by τ, given by $q = e^{i\pi\tau}$. All the theta functions satisfy a diffusion-like differential equation $\partial^2 \vartheta(u, \tau) \big/ \partial u^2 = 4\pi i\, \partial \vartheta(u, \tau)/\partial \tau$.

The Siegel theta function `SiegelTheta[τ, s]` with Riemann square modular matrix τ of dimension p and vector s generalizes the elliptic theta functions to complex dimension p. It is defined by $\Theta(\tau, s) = \sum_n \exp(i\pi(n.\tau.n + 2\, n.s))$, where n runs over all p-dimensional integer vectors. The Siegel theta function with characteristic `SiegelTheta[v, τ, s]` is defined by $\Theta(v, \tau, s) = \sum_n \exp(i\pi((n+\alpha).\tau.(n+\alpha) + 2\,(n+\alpha).(s+\beta)))$, where the characteristic v is a pair of p-dimensional vectors $\{\alpha, \beta\}$.

The Jacobi elliptic functions can be expressed as ratios of the theta functions.

An alternative notation for theta functions is $\Theta(u \mid m) = \vartheta_4(v \mid m)$, $\Theta_1(u \mid m) = \vartheta_3(v \mid m)$, $H(u \mid m) = \vartheta_1(v)$, $H_1(u \mid m) = \vartheta_2(v)$, where $v = \pi u / 2K(m)$.

The *Neville theta functions* can be defined in terms of the theta functions as $\vartheta_s(u) = 2K(m)\, \vartheta_1(v \mid m) / \pi \vartheta_1'(0 \mid m)$, $\vartheta_c(u) = \vartheta_2(v \mid m)/\vartheta_2(0 \mid m)$, $\vartheta_d(u) = \vartheta_3(v \mid m)/\vartheta_3(0 \mid m)$, $\vartheta_n(u) = \vartheta_4(v \mid m)/\vartheta_4(0 \mid m)$, where $v = \pi u / 2K(m)$. The Jacobi elliptic functions can be represented as ratios of the Neville theta functions.

The *Weierstrass elliptic function* `WeierstrassP[u, {g2, g3}]` can be considered as the inverse of an elliptic integral. The Weierstrass function $\wp(u; g_2, g_3)$ gives the value of x for which $u = \int_{\infty}^{x} \left(4t^3 - g_2 t - g_3\right)^{-1/2} dt$. The function `WeierstrassPPrime[u, {g2, g3}]` is given by $\wp'(u; g_2, g_3) = \frac{\partial}{\partial u} \wp(u; g_2, g_3)$.

The Weierstrass functions are also sometimes written in terms of their *fundamental half-periods* ω and ω', obtained from the invariants g_2 and g_3 using `WeierstrassHalfPeriods[{u, {g2, g3}}]`.

The function `InverseWeierstrassP[p, {g2, g3}]` finds one of the two values of u for which $p = \wp(u; g_2, g_3)$. This value always lies in the parallelogram defined by the complex number half-periods ω and ω'.

`InverseWeierstrassP[{p, q}, {g2, g3}]` finds the unique value of u for which $p = \wp(u; g_2, g_3)$ and $q = \wp'(u; g_2, g_3)$. In order for any such value of u to exist, p and q must be related by $q^2 = 4\,p^3 - g_2\,p - g_3$.

The *Weierstrass zeta function* `WeierstrassZeta[u, {g2, g3}]` and *Weierstrass sigma function* `WeierstrassSigma[u, {g2, g3}]` are related to the Weierstrass elliptic functions by $\zeta'(z; g_2, g_3) = -\wp(z; g_2, g_3)$ and $\sigma'(z; g_2, g_3)/\sigma(z; g_2, g_3) = \zeta(z; g_2, g_3)$.

The Weierstrass zeta and sigma functions are not strictly elliptic functions since they are not periodic.

Elliptic Modular Functions

`DedekindEta[τ]`	Dedekind eta function $\eta(\tau)$
`KleinInvariantJ[τ]`	Klein invariant modular function $J(\tau)$
`ModularLambda[τ]`	modular lambda function $\lambda(\tau)$

Elliptic modular functions.

The *modular lambda function* `ModularLambda[τ]` relates the ratio of half-periods $\tau = \omega'/\omega$ to the parameter according to $m = \lambda(\tau)$.

The *Klein invariant modular function* `KleinInvariantJ[τ]` and the *Dedekind eta function* `DedekindEta[τ]` satisfy the relations $\Delta = g_2^3 / J(\tau) = (2\,\pi)^{12}\,\eta^{24}(\tau)$.

Modular elliptic functions are defined to be invariant under certain fractional linear transformations of their arguments. Thus for example $\lambda(\tau)$ is invariant under any combination of the transformations $\tau \to \tau + 2$ and $\tau \to \tau/(1 - 2\,\tau)$.

Generalized Elliptic Integrals and Functions

`ArithmeticGeometricMean[a,b]`	the arithmetic-geometric mean of a and b
`EllipticExp[u,{a,b}]`	generalized exponential associated with the elliptic curve $y^2 = x^3 + a\,x^2 + b\,x$
`EllipticLog[{x,y},{a,b}]`	generalized logarithm associated with the elliptic curve $y^2 = x^3 + a\,x^2 + b\,x$

Generalized elliptic integrals and functions.

The definitions for elliptic integrals and functions given above are based on traditional usage. For modern algebraic geometry, it is convenient to use slightly more general definitions.

The function `EllipticLog[{x, y}, {a, b}]` is defined as the value of the integral $\frac{1}{2}\int_{\infty}^{x}\left(t^3 + a\,t^2 + b\,t\right)^{-1/2} dt$, where the sign of the square root is specified by giving the value of y such that $y = \sqrt{x^3 + a\,x^2 + b\,x}$. Integrals of the form $\int_{\infty}^{x}\left(t^2 + a\,t\right)^{-1/2} dt$ can be expressed in terms of the ordinary logarithm (and inverse trigonometric functions). You can think of `EllipticLog` as giving a generalization of this, where the polynomial under the square root is now of degree three.

The function `EllipticExp[u, {a, b}]` is the inverse of `EllipticLog`. It returns the list $\{x, y\}$ that appears in `EllipticLog`. `EllipticExp` is an elliptic function, doubly periodic in the complex u plane.

`ArithmeticGeometricMean[a, b]` gives the *arithmetic-geometric mean* (*AGM*) of two numbers a and b. This quantity is central to many numerical algorithms for computing elliptic integrals and other functions. For positive reals a and b the AGM is obtained by starting with $a_0 = a$, $b_0 = b$, then iterating the transformation $a_{n+1} = \frac{1}{2}(a_n + b_n)$, $b_{n+1} = \sqrt{a_n\,b_n}$ until $a_n = b_n$ to the precision required.

Mathieu and Related Functions

`MathieuC[a,q,z]`	even Mathieu functions with characteristic value a and parameter q
`MathieuS[b,q,z]`	odd Mathieu functions with characteristic value b and parameter q
`MathieuCPrime[a,q,z]` and `MathieuSPrime[b,q,z]`	z derivatives of Mathieu functions
`MathieuCharacteristicA[r,q]`	characteristic value a_r for even Mathieu functions with characteristic exponent r and parameter q
`MathieuCharacteristicB[r,q]`	characteristic value b_r for odd Mathieu functions with characteristic exponent r and parameter q
`MathieuCharacteristicExponent[a,q]`	characteristic exponent r for Mathieu functions with characteristic value a and parameter q

Mathieu and related functions.

The *Mathieu functions* `MathieuC[a, q, z]` and `MathieuS[a, q, z]` are solutions to the equation $y'' + [a - 2q\cos(2z)]y = 0$. This equation appears in many physical situations that involve elliptical shapes or periodic potentials. The function `MathieuC` is defined to be even in z, while `MathieuS` is odd.

When $q = 0$ the Mathieu functions are simply $\cos\left(\sqrt{a}\ z\right)$ and $\sin\left(\sqrt{a}\ z\right)$. For nonzero q, the Mathieu functions are only periodic in z for certain values of a. Such *Mathieu characteristic values* are given by `MathieuCharacteristicA[r, q]` and `MathieuCharacteristicB[r, q]` with r an integer or rational number. These values are often denoted by a_r and b_r.

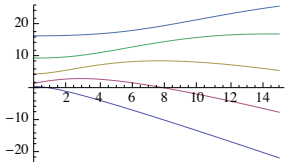
For integer r, the even and odd Mathieu functions with characteristic values a_r and b_r are often denoted $ce_r(z, q)$ and $se_r(z, q)$, respectively. Note the reversed order of the arguments z and q.

According to Floquet's theorem any Mathieu function can be written in the form $e^{i\ rz} f(z)$, where $f(z)$ has period 2π and r is the *Mathieu characteristic exponent* `MathieuCharacteristicExponent[a, q]`. When the characteristic exponent r is an integer or rational number, the Mathieu function is therefore periodic. In general, however, when r is not a real integer, a_r and b_r turn out to be equal.

This shows the first five characteristic values a_r as functions of q.

```
In[1]:= Plot[Evaluate[Table[MathieuCharacteristicA[r, q], {r, 0, 4}]], {q, 0, 15}]
```

Out[1]=



Working with Special Functions

automatic evaluation	exact results for specific arguments
N[*expr*, *n*]	numerical approximations to any precision
D[*expr*, *x*]	exact results for derivatives
N[D[*expr*, *x*]]	numerical approximations to derivatives
Series[*expr*, {*x*, x_0, *n*}]	series expansions
Integrate[*expr*, *x*]	exact results for integrals
NIntegrate[*expr*, *x*]	numerical approximations to integrals
FindRoot[*expr*==0, {*x*, x_0}]	numerical approximations to roots

Some common operations on special functions.

Most special functions have simpler forms when given certain specific arguments. *Mathematica* will automatically simplify special functions in such cases.

Mathematica automatically writes this in terms of standard mathematical constants.

```
In[1]:= PolyLog[2, 1 / 2]
```

Out[1]= $\frac{\pi^2}{12} - \frac{\text{Log}[2]^2}{2}$

Here again *Mathematica* reduces a special case of the Airy function to an expression involving gamma functions.

```
In[2]:= AiryAi[0]
```

Out[2]= $\frac{1}{3^{2/3}\,\text{Gamma}\left[\frac{2}{3}\right]}$

For most choices of arguments, no exact reductions of special functions are possible. But in such cases, *Mathematica* allows you to find numerical approximations to any degree of precision. The algorithms that are built into *Mathematica* cover essentially all values of parameters—real and complex—for which the special functions are defined.

There is no exact result known here.

```
In[3]:= AiryAi[1]
Out[3]= AiryAi[1]
```

This gives a numerical approximation to 40 digits of precision.

```
In[4]:= N[AiryAi[1], 40]
Out[4]= 0.1352924163128814155241474235154663061749
```

The result here is a huge complex number, but *Mathematica* can still find it.

```
In[5]:= N[AiryAi[1000 I]]
Out[5]= -4.780266637767027×10^6472 + 3.674920907226875×10^6472 i
```

Most special functions have derivatives that can be expressed in terms of elementary functions or other special functions. But even in cases where this is not so, you can still use `N` to find numerical approximations to derivatives.

This derivative comes out in terms of elementary functions.

```
In[6]:= D[FresnelS[x], x]
Out[6]= Sin[π x^2 / 2]
```

This evaluates the derivative of the gamma function at the point 3.

```
In[7]:= Gamma'[3]
Out[7]= 2 (3/2 - EulerGamma)
```

There is no exact formula for this derivative of the zeta function.

```
In[8]:= Zeta'[Pi]
Out[8]= Zeta'[π]
```

Applying N gives a numerical approximation.

```
In[9]:= N[%]
Out[9]= -0.167603
```

Mathematica incorporates a vast amount of knowledge about special functions—including essentially all the results that have been derived over the years. You access this knowledge whenever you do operations on special functions in *Mathematica*.

Here is a series expansion for a Fresnel function.

```
In[10]:= Series[FresnelS[x], {x, 0, 15}]
```

$$\text{Out[10]}= \frac{\pi x^3}{6} - \frac{\pi^3 x^7}{336} + \frac{\pi^5 x^{11}}{42\,240} - \frac{\pi^7 x^{15}}{9\,676\,800} + O[x]^{16}$$

Mathematica knows how to do a vast range of integrals involving special functions.

```
In[11]:= Integrate[AiryAi[x]^2, {x, 0, Infinity}]
```

$$\text{Out[11]}= \frac{1}{3^{2/3}\,\text{Gamma}\left[\frac{1}{3}\right]^2}$$

One feature of working with special functions is that there are a large number of relations between different functions, and these relations can often be used in simplifying expressions.

FullSimplify[*expr*]	try to simplify *expr* using a range of transformation rules

Simplifying expressions involving special functions.

This uses the reflection formula for the gamma function.

```
In[12]:= FullSimplify[Gamma[x] Gamma[1 - x]]
Out[12]= π Csc[π x]
```

This makes use of a representation for Chebyshev polynomials.

```
In[13]:= FullSimplify[ChebyshevT[n, z] - k Cos[n ArcCos[z]]]
Out[13]= -(-1 + k) Cos[n ArcCos[z]]
```

The Airy functions are related to Bessel functions.

```
In[14]:= FullSimplify[3 AiryAi[1] + Sqrt[3] AiryBi[1]]
```

$$\text{Out[14]}= 2\,\text{BesselI}\left[-\frac{1}{3}, \frac{2}{3}\right]$$

`FunctionExpand[`*expr*`]` try to expand out special functions

Manipulating expressions involving special functions.

This expands the Gauss hypergeometric function into simpler functions.

```
In[15]:= FunctionExpand[Hypergeometric2F1[1 / 2, 3 / 2, 3, x]]
```

$$\text{Out[15]}= \frac{16\,(2-x)\,\text{EllipticE}[x]}{3\,\pi\,x^2}+\frac{16\,(-2+2\,x)\,\text{EllipticK}[x]}{3\,\pi\,x^2}$$

Here is an example involving Bessel functions.

```
In[16]:= FunctionExpand[BesselY[n, I x]]
```

$$\text{Out[16]}= -\frac{2\,(i\,x)^{-n}\,x^n\,\text{BesselK}[n,\,x]}{\pi}+\text{BesselI}[n,\,x]\,\left(-(i\,x)^{-n}\,x^n+(i\,x)^n\,x^{-n}\,\text{Cos}[n\,\pi]\right)\,\text{Csc}[n\,\pi]$$

In this case the final result does not even involve `PolyGamma`.

```
In[17]:= FunctionExpand[Im[PolyGamma[0, 3 I]]]
```

$$\text{Out[17]}= \frac{1}{6}+\frac{1}{2}\,\pi\,\text{Coth}[3\,\pi]$$

This finds an expression for a derivative of the Hurwitz zeta function.

```
In[18]:= FunctionExpand[Derivative[1, 0][Zeta][-1, 4]]
```

$$\text{Out[18]}= \frac{1}{12}+2\,\text{Log}[2]+3\,\text{Log}[3]-\text{Log}[\text{Glaisher}]$$