

1. Introduction and Survey

In the 21st century computers are in the process of changing all aspects of our lives. That includes also mathematics - both the way we do it and even the kind of problems that interest us. The ability to make computations far beyond anything that was possible without the aid of computers has opened up previously inaccessible areas of research to anyone equipped with a computer, suitable software and some mathematical knowledge. This has led to the creation of sophisticated programs intended for dealing with all aspects of computational mathematics - symbolic manipulation, numerical computation and visualization (including interactive one). A leading program of this type is *Mathematica* produced by Wolfram Research. *Mathematica* is, of course, not only a tool of research and exploring new areas of computational mathematics but also an extremely effective aid in studying traditional ones. The aim of this essay is to give a brief introduction to some of the possibilities offered by this remarkable program. For this purpose we have made use of a number of examples, some of which are due to the authors of this text, some have been borrowed from *Mathematica*'s official documentation and some from various sources on the Internet.

Basic information about Mathematica

■ The main web-page

The main website for current information, useful projects, plug-ins, learning center and documentation is www.wolfram.com. At this website one can also find a free CDF-Player, thousands of demonstrations with source files and educational videos.

Mathematica is one of the most powerful and sophisticated systems for symbolic and numerical computation and visualization. But it is also more than that. Wolfram Research used to describe *Mathematica* as a "system for doing mathematics by computer" but it has since changed this to "the only development platform fully integrating computation into complete work-flows". In other words, Wolfram Research now conceives *Mathematica* as essentially a universal tool for almost every purpose - symbolic and numerical computation and programming being a central aspect (but this is not the whole story). For example, *Mathematica* is also an advanced technical typesetting tool, which can produce mathematical documents of quality comparable to $\text{T}_{\text{E}}\text{X}$ and Latex but in a fully WYSIWYG (What You See is What You Get) way. At the same time, these documents can contain "live" mathematical formulas and graphics that can be sent to another person (a collaborator, a student or a professor) who can use them to verify the correctness of results or to perform additional computations etc.

In terms of its overall abilities *Mathematica* currently has no comparable rival (except, perhaps, for suites of several applications). But if we restrict our attention only to computation, there are other programs (e.g. Maple, MatLab and others) that can do similar things. As a mathematical tool, *Mathematica* is a "general purpose" system. It is very strong in both symbolic and numeric computation. It has very many powerful specialized functions for subjects as different as polynomial algebra, graph theory, statistics of financial mathematics. However, in some areas (particularly in pure mathematics) there are specialized programs (such as MAGMA, Singular, Macaulay2 etc.) which can do some things that *Mathematica* cannot do without additional programming or can do them faster (which in certain situations can be crucial).

The official distributor of Mathematica in Poland is the company Gambit <<http://www.mathematica.pl>>.

■ Wolfram Research sites and projects

Wolfram Research has a number of sites and projects of interest both to *Mathematica* users and the general public:

□ Mathworld <<http://mathworld.wolfram.com/>>

Mathworld is a very useful and extensive web resource with definitions, examples and main theorems in mathematics (and often - *Mathematica* files in the NB format, i.e., *Mathematica* notebooks .)

□ Wolfram Alpha <<http://www.wolframalpha.com/>>

Wolfram Alpha - *Mathematica* (and more) for everyone!

□ Wolfram Demonstrations Project <<http://demonstrations.wolfram.com>>

□ Education

Wolfram Research has an extensive educational program. To register for on-line Wolfram Educational Group (WEG) seminars and classes, one can visit

<<http://www.wolfram.com/services/education/calendar.cgi>>

The Wolfram Education Group (WEG) offers a wide range of free on-line seminars featuring the latest version of *Mathematica* (see <http://www.wolfram.com/services/education/seminars/>).

There are conferences/master classes/other events in Europe and in particular, in Poland. In Warsaw, ICM (the interdisciplinary center at UW) has *Mathematica* and provides some training.

□ The *Mathematica* Journal

The Mathematica Journal <<http://www.mathematica-journal.com/>> is an on-line journal with research and educational papers.

■ Other *Mathematica* related sites

There is an excellent and free introduction to *Mathematica* programming due to Leonid Shifrin:

<http://www.mathprogramming-intro.org/>

Also the *Mathematica* Guidebooks <<http://www.mathematicaguidebooks.org/index.shtml>> provide a wealth of applications and examples, particularly from physics but also from other areas. Unfortunately the guidebooks are still not fully compatible with *Mathematica* 6 and later versions so they are only suitable for users who are advanced enough to update the relevant parts themselves.

Finally there is an excellent *Mathematica* discussion group (known as the *MathGroup*)

<http://groups.google.com/group/comp.soft-sys.math.mathematica/topics?pli=1>

You can ask any questions about any aspect of *Mathematica* and get a variety of answers from *Mathematica* experts, including Wolfram Research employees. Sometimes you can even have your math problems solved for you.

This is really the best *Mathematica* centered resource on the Web, if you know how to make use of it.

The above is, of course, not intended to be an exhaustive account of *Mathematica* related resources on the Internet. Far from it, the number of both general purpose and specialised sites intended for beginners as well as advanced users is huge and constantly increasing (e.g. blog.wolfram.com, *Mathematica* Tips on Twitter etc).

SELECTED EXAMPLES OF WHAT MATHEMATICA CAN DO

■ Computational applications

▣ Built-in functions

A large number of problems (even some “real life” ones) can be solved by simply applying one of Mathematica’s built-in “functions” (the word “function” in connection with *Mathematica* is used in a somewhat different sense than in mathematics, closer to what in other programming languages is called a “procedure”). Among the most useful of these functions are `Solve` and `Reduce`. These are very general functions that use a large ensemble of advanced algorithms to solve all kinds of equations and inequalities, many of which would appear unsolvable even to people with good knowledge of mathematics. We will look at some remarkable examples of the sort of thing that can be accomplished with `Reduce`.

(In version 8 the functions `Solve` became very enhanced and can solve many of the problems that `Reduce` can, but it often uses somewhat different techniques. Probably the main difference between the two functions is that `Reduce` always attempts to return the complete solutions of a problem while `Solve` will in some situations return a partial solution.)

Perhaps the most basic and frequently performed task in mathematics is solving equations. `Reduce` can be used, of course, to solve equations (and systems of equations) far too complicated to solve by hand. We start by looking at a simple cubic equation, which *Mathematica* can solve using the famous formula of del Ferro (often attributed to Tartaglia):

`Reduce[x^3 - x + 1 == 0, x, Cubics -> True]`

$$x = -\sqrt[3]{\frac{2}{3(9 - \sqrt{69})}} - \frac{\sqrt[3]{\frac{1}{2}(9 - \sqrt{69})}}{3^{2/3}} \vee x = \left((1 + i\sqrt{3}) \sqrt[3]{\frac{1}{2}(9 - \sqrt{69})} \right) / (2 \times 3^{2/3}) + \frac{1 - i\sqrt{3}}{2^{2/3} \sqrt[3]{3(9 - \sqrt{69})}} \vee$$

$$x = \left((1 - i\sqrt{3}) \sqrt[3]{\frac{1}{2}(9 - \sqrt{69})} \right) / (2 \times 3^{2/3}) + \frac{1 + i\sqrt{3}}{2^{2/3} \sqrt[3]{3(9 - \sqrt{69})}}$$

We can also ask *Mathematica* to compute the real root only:

`Reduce[x^3 - x + 1 == 0, x, Reals, Cubics -> True]`

$$x = -\sqrt[3]{\frac{2}{3(9 - \sqrt{69})}} - \frac{\sqrt[3]{\frac{1}{2}(9 - \sqrt{69})}}{3^{2/3}}$$

We can compute its numerical value to arbitrary precision:

`N[%, 30]`

$x = -1.32471795724474602596090885448$

From the work of Abel and Galois it is known that no solutions of the above kind (in terms of radicals) can be given for polynomial equations of degree higher than 5. However, this does not stop *Mathemat-*

ica :

```
Reduce[x^5 - x + 1 == 0, x]
```

$$x = \text{Root}[\#1^5 - \#1 + 1 \&, 1] \vee x = \text{Root}[\#1^5 - \#1 + 1 \&, 2] \vee \\ x = \text{Root}[\#1^5 - \#1 + 1 \&, 3] \vee x = \text{Root}[\#1^5 - \#1 + 1 \&, 4] \vee x = \text{Root}[\#1^5 - \#1 + 1 \&, 5]$$

Again the values can be computed to arbitrary precision:

```
N[%, 30]
```

$$x = -1.16730397826141868425604589985 \vee \\ x = -0.181232444469875383901800237781 - 1.083954101317710668430344492981 i \vee \\ x = -0.181232444469875383901800237781 + 1.083954101317710668430344492981 i \vee \\ x = 0.764884433600584726029823187709 - 0.352471546031726249317947091403 i \vee \\ x = 0.764884433600584726029823187709 + 0.352471546031726249317947091403 i$$

Reduce can also deal with purely symbolic problems. For example, consider the quadratic equation $ax^2 + bx + c = 0$. Let's obtain the well known condition for it to have two equal roots.

```
Reduce[ $\exists_{x,a} x^2 + b x + c = 0 \wedge \forall_{y,a} y^2 + b y + c = 0 \rightarrow x = y$ , {a, b, c}]
```

$$(a = 0 \wedge b \neq 0) \vee \left(a \neq 0 \wedge c = \frac{b^2}{4a} \right)$$

Let's now try something non-polynomial. Here is a trigonometric equation. We ask Reduce to solve it for a range of values in an interval, where there is a finite number of solutions:

```
Reduce[Cos[x] == Sin[x] && 0 < x < 2 Pi, x] // FullSimplify
```

$$4x = \pi \vee 4x = 5\pi$$

We can also obtain a complete solution without a restriction on the domain of solutions.

```
Reduce[Cos[x] == Sin[x], x] // FullSimplify
```

$$c_1 \in \mathbb{Z} \wedge (\pi(8c_1 - 3) = 4x \vee 8\pi c_1 + \pi = 4x)$$

Here is an equation that seems impossible to solve by hand, but Reduce can do it:

```
Reduce[Cos[Cos[x]] == Sin[Sin[x]] && Abs[x] < 1, x]
```

$$x = \text{Root}[\{\cos(\cos(\#1)) - \sin(\sin(\#1)) \&, 0.7853981633974483096156608458198757210492923498437764552437 - \\ 0.4663385348278305845718632848784660354269560408360176474839 i\}] \vee \\ x = \text{Root}[\{\cos(\cos(\#1)) - \sin(\sin(\#1)) \&, 0.7853981633974483096156608458199 + \\ 0.4663385348278305845718632848785 i\}]$$


```
N[%, 10]
```

$$x = 0.7853981634 - 0.4663385348 i \vee x = 0.7853981634 + 0.4663385348 i$$

All the solutions are complex numbers. Reduce can prove that there are no real solutions:

```
Reduce[Cos[Cos[x]] == Sin[Sin[x]], x, Reals]
```

```
False
```

Here is a completely different kind of equation, this time over the integers.

```
Reduce[n! + n == 726 && n > 0, n, Integers]
```

```
n = 6
```

Let's now try something harder - a "real life" problem. There is an Internet forum, called the *MathGroup*, where people post questions about *Mathematica*, including mathematical questions they try to solve with *Mathematica*. Solutions are posted by other users including some of the staff of Wolfram Research. One such question was posted by Ivan Smirnov in January 2011 (the whole thread can be found here: [Smirnov's problem](#))

Are there any integer solutions of $x^{10} + y^{10} + z^{10} = t^2$?

It is easy to find trivial solutions where two of the three variables x, y, z are zero, so let's look for such solutions. Reduce cannot solve the complete problem but it can quickly verify that there are no solutions for $t \leq 10^4$.

```
Reduce[x^10 + y^10 + z^10 == t^2 && 0 < x && 0 < y && x < y && y < z && 1 < t <= 10^4,
{x, y, z, t}, Integers] // Timing
```

```
{1.53308, False}
```

In fact (after changing certain settings which limit the number of cases Reduce will consider, one can verify that there are no solutions for $1 \leq t \leq 10^{10}$.

Reduce can be very useful in many undergraduate courses. For example, in Analysis 1 one often needs to prove that a certain integer sequence is monotonic. Consider, for example, the problem

```
Reduce[n^(1/n) > (n+1)^(1/(n+1)) && n > 0, n, Integers]
```

```
n ∈ Z ∧ n ≥ 3
```

□ Programming

Although many problems can be solved just by applying built-in functions, in many cases there is no built-in function that will do all the work by itself. In such cases we need to do our own programming. Here are some recent problems taken from the *MathGroup*.

Find three 2 - digit prime numbers such that :

- (i) The average of any two of the three is a prime number, and
- (ii) The average of all three is also a prime number

There is no *Mathematica* function that will automatically answer a question like this, but the answer can be found with a few lines of *Mathematica* code:

```
ls1 = Select[Range[11, 99], PrimeQ];
ls2 = Tuples[ls1, {3}];
ls3 = DeleteCases[Union[Sort /@ ls2], {___, x_, ___, x_, ___}];
Select[ls3, And @@ PrimeQ /@ Mean /@ Partition[#1, 2, 1, {1, 1}] &]
```

```
( 11 23 71 )
( 11 23 83 )
( 11 47 71 )
( 13 61 73 )
( 17 29 89 )
( 23 59 83 )
( 29 53 89 )
```

Here is another question from the *MathGroup*:

What' s the easiest way to determine the length of the repeating cycle for decimal expansions of fractions? For example, $1/7 == 0.14285714285714285714...$..so the length of its repeating cycle (142857) is 6. For $1/3$ the length of the cycle is obviously 1. For some fractions, e.g., $1/4$, the decimal expansion is not cyclical (in base 10).

We use programming to define a function `lengthOfCycle`. We make use of several built-in *Mathematica* functions, in particular, `IntegerExponent` and `MultiplicativeOrder`. If we could not use these functions the program would have to be much longer, more complicated and less efficient.

```
lengthOfCycle[x_Rational] :=
Module[{n = Denominator[x], a, b}, a = IntegerExponent[n, 2];
b = IntegerExponent[n, 5]; MultiplicativeOrder[10, n / (2^a * 5^b)]]
```

```
lengthOfCycle[1 / 7]
```

```
6
```

In certain situations programs written in *Mathematica* programming language can be much slower than programs written in typed and compiled languages such as C, Java, etc (although, of course, writing such programs in *Mathematica* is almost always much quicker). However, for many types of programs this difference can be greatly reduced by “compiling”. Not every kind of *Mathematica* program can be successfully compiled but when it can, this can make a very big difference to performance. Here is an example where a non-compiled *Mathematica* program performs rather poorly. We will give only a compiled version, that is very fast. The program constructs an Ulam spiral and is due to Daniel Lichtblau of Wolfram Research:

```
ulamSpiral = Compile[{{len, _Integer}},
  Module[{dat = Range[len], x = 0, y = 0, shift = 1, i = 0, j = 0, xincr = {1, 0, -1, 0},
    yincr = {0, 1, 0, -1}, shiftincr = {0, 1, 0, 1}}, Select[Table[i++;
    If[i ≥ shift, i = 1; shift += shiftincr[Mod[j, 4, 1]]; j++;];
    x += xincr[Mod[j, 4, 1]]; y += yincr[Mod[j, 4, 1]];
    If[PrimeQ[dat[[num]]], {x, y}, {1000, 1000}], {num, len}],
  Abs[First[#]] < 1000 &]], {{PrimeQ[_], True | False}}];
```

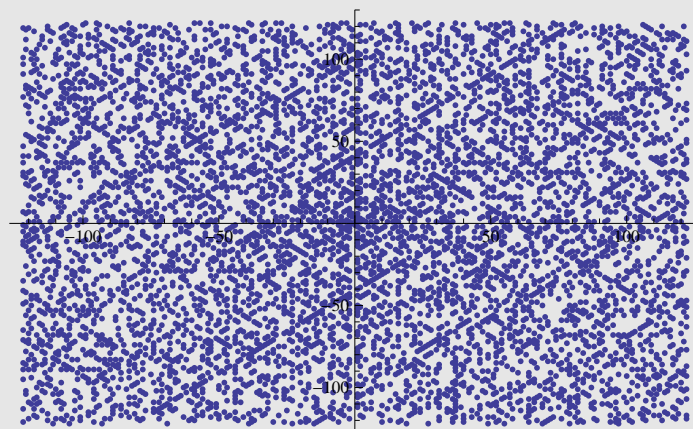
Making a spiral with 60 000 points takes only a fraction of a second:

```
ls = ulamSpiral[60 000]; // Timing
```

```
{0.088024, Null}
```

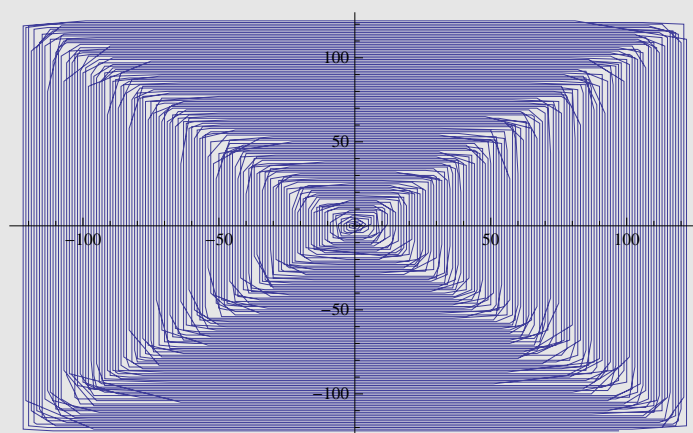
We can view the spiral as a collection of points:

```
ListPlot[ls]
```



or of lines

```
ListLinePlot[ls]
```



In *Mathematica* version 8, this can be speeded up further by using the option `CompilationTarget -> "C"`, which however, requires that a C-compiler be installed on the computer.

■ Specialized Mathematics

Mathematica contains a large number of specialized functions for various areas of mathematics, ranging from Group Theory and Number Theory to Statistics (a vast number of statistical distributions are available as built-in functions) and financial mathematics. A particularly interesting aspect is *Mathematica*'s ability to obtain live financial data from the Internet and analyze them by means of a variety of specific financial functions. As an example we compare the performance of the value of the index of the American NASDAQ stock exchange (on which most US technology companies are represented) and of Apple Computer Inc.

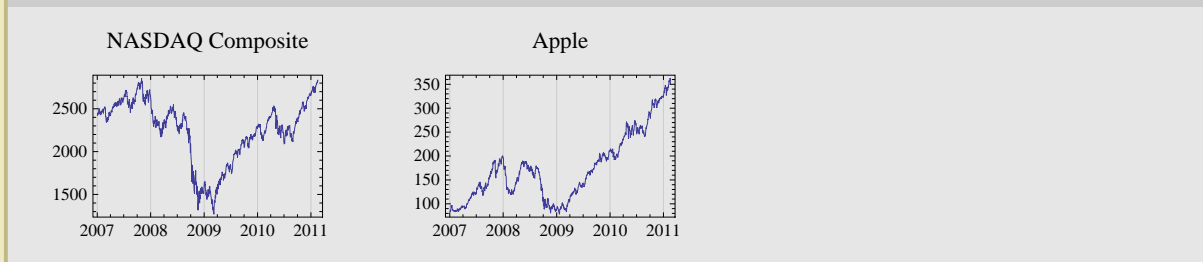
```
data1 = FinancialData["^IXIC", "Jan. 1, 2007"];
```

```
g1 = DateListPlot[data1, Joined -> True, PlotLabel -> "NASDAQ Composite"];
```

```
data2 = FinancialData["AAPL", "Jan. 1, 2007"];
```

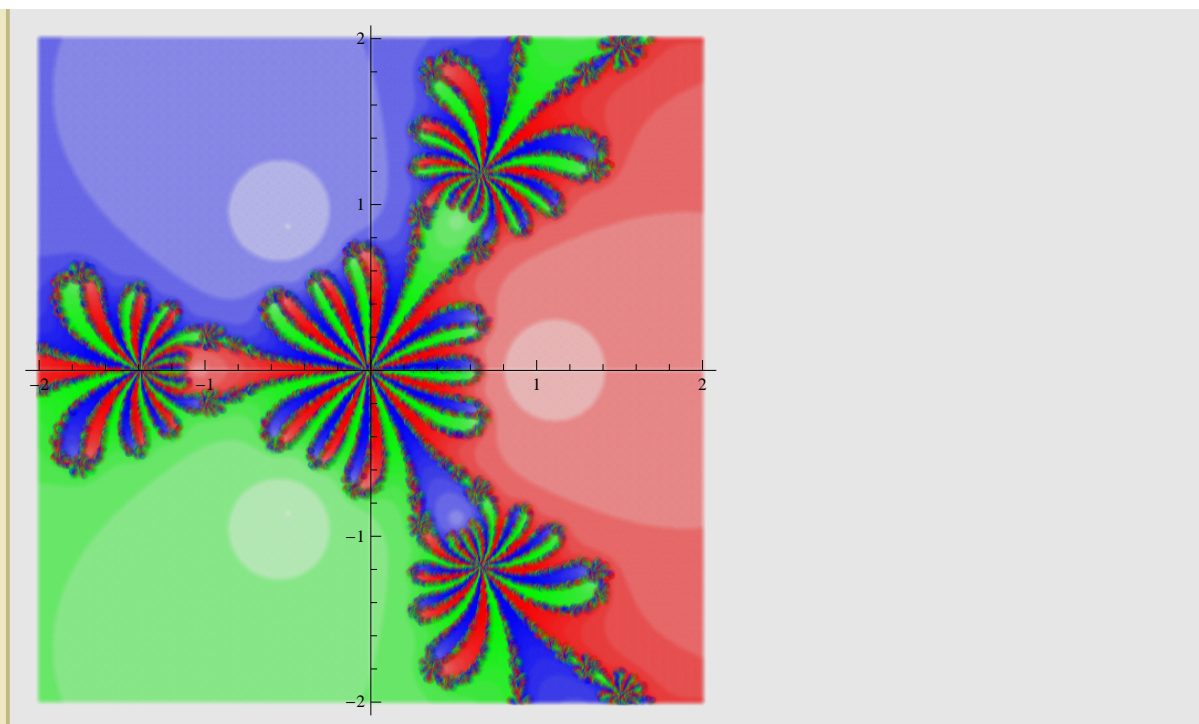
```
g2 = DateListPlot[data2, Joined -> True, PlotLabel -> "Apple"];
```

```
GraphicsGrid[{{g1, g2}}]
```



■ Graphics

Mathematica has remarkable graphic capabilities. Here is an example of a mathematical graphic related to the subject of iteration of functions in the complex plane and “fractals”.



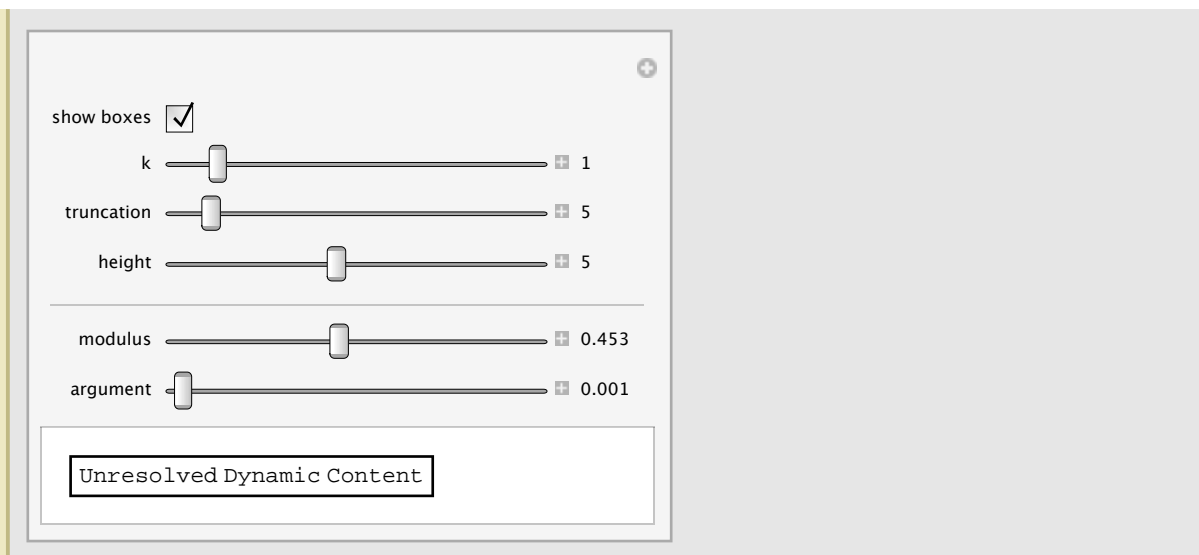
■ Beyond Graphics - Interactive Dynamics

A completely new set of features, that have nothing quite similar in other programs, appeared in *Mathematica* 6. These involve interactive “dynamic” abilities that are difficult to describe in a static format, but can be seen below.

Wolfram Demonstrations Project

It is difficult to describe on a static page how remarkably useful this functionality is in all kinds of situations, including both research and teaching. A single interactive *Mathematica* notebook can replace dozens of static pictures.

Here we see one example. It shows the convergence of the power series $\sum_{n=1}^{\infty} \frac{z^n}{n^k}$ on the unit disk $|z| \leq 1$ for $k = 0, 1, 2, \dots$. Here we only see the case $n = 1$, when the series converges in the unit disk and everywhere on the boundary except at the point $z = 1$. The graphic on the left shows the complex values over the unit disk of a finite sum of terms of the series, on the right we see the values of the analytic function defined by the series. The modulus of the complex values of a function is represented by the height of the graph and the argument by colour. In the first graphic a shorter finite sum is used, in the second a longer. We can see the improvement in the approximation and the singular behaviour at $z = 1$. We can also interactively choose any point on the unit disk and see the modulus of the difference between the values of the corresponding finite sum and the infinite series. Again, this is the sort of thing that would be very difficult to reproduce by other means.



The CDF Format and the CDF Player

With version 8 of *Mathematica* a new file format for *Mathematica* files was introduced.


The CDF player makes it easy to use *Mathematica* in class or at home even when students do not have *Mathematica* themselves. It is a free program that can be downloaded from Wolfram's web site.

Wolfram CDF Player

Wolfram Alpha

Wolfram Alpha - *Mathematica* for everyone!

Wolfram Alpha received a great deal of publicity when it first appeared and it may even be better known than *Mathematica*, but not many people seem to realize that Wolfram Alpha is based on *Mathematica* and, in effect, provides the general public with free access to much of *Mathematica*'s functionality. Moreover, it does not require learning the *Mathematica* syntax. Indeed, the most famous aspect of Wolfram Alpha is its ability to use "free form" mathematical input - just type in what you want Wolfram Alpha to do for you in ordinary English (other language input is planned for the future) and Wolfram Alpha will attempt to guess what you want and then will use *Mathematica* to obtain the answer. In fact, the answers returned by Wolfram Alpha are generally more complete than *Mathematica* would normally return (they can all be obtained with *Mathematica* but it may require several commands or even some programming). Here is an example of computing the integral $\int x \log(x) dx$ with Wolfram Alpha. Just type: "indefinite integral of $x \log(x)$ " and you will obtain the output shown below. In versions of *Mathematica* earlier than 8, a very precise syntax would be needed to obtain the same result (version 8 of *Mathematica* can also use "free form" input).


WolframAlpha[™] computational knowledge engine

indefinite integral of x log(x) =

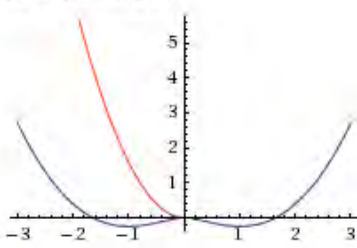
Assuming "log" is the natural logarithm | Use [the base 10 logarithm](#) instead

Indefinite Integral: [Show steps](#)

$$\int x \log(x) dx = \frac{1}{4} x^2 (2 \log(x) - 1) + \text{constant}$$

log(x) is the natural logarithm »

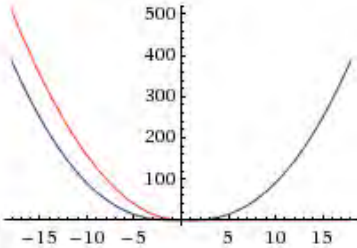
Plots of the Integral:



(x from -3 to 3)

— real part

— imaginary part



(x from -15 to 15)

— real part

— imaginary part

Alternate form of the Integral:

$$\frac{1}{2} x^2 \log(x) - \frac{x^2}{4} + \text{constant}$$

1. Basic Principles

■ 1. Overview of Mathematica Features. Mathematica as a Calculator.

You can get a lot of information from the Help Browser (to access it press F1 or use the Help menu).

One can use Mathematica just like a calculator: one types in formulas and Mathematica returns back their values. Just press SHIFT + ENTER (RETURN) to tell Mathematica to evaluate the input you have given it.

Example.

```
2 + 2
```

(press SHIFT + ENTER after putting the cursor after 2 + 2 to see the output)

```
2 + 2
```

```
4
```

With a text - based interface, you interact with Mathematica just by typing successive lines of input, and getting back successive lines of output on your screen.

At each stage, Mathematica prints a prompt of the form In[n] := to tell you that it is ready to receive input. When you have entered your input, Mathematica processes it, and then displays the result with a label of the form Out[n] =.

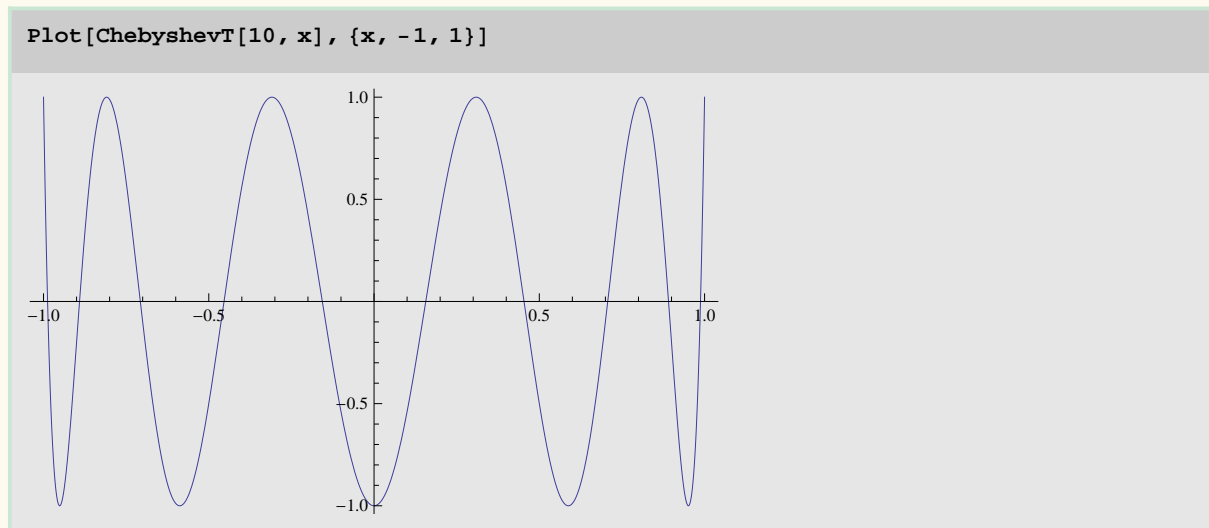
Different text - based interfaces use slightly different schemes for letting Mathematica know when you have finished typing your input. With some interfaces you press Shift - Return, while in others Return alone is sufficient.

An important feature of Mathematica is its ability to handle formulas as well as numbers. Whenever you use Mathematica, you are accessing the world' s largest collection of computational algorithms. Mathematica knows about all the hundreds of special functions in pure and applied mathematics (e.g., Chebyshev polynomials, Bessel functions).

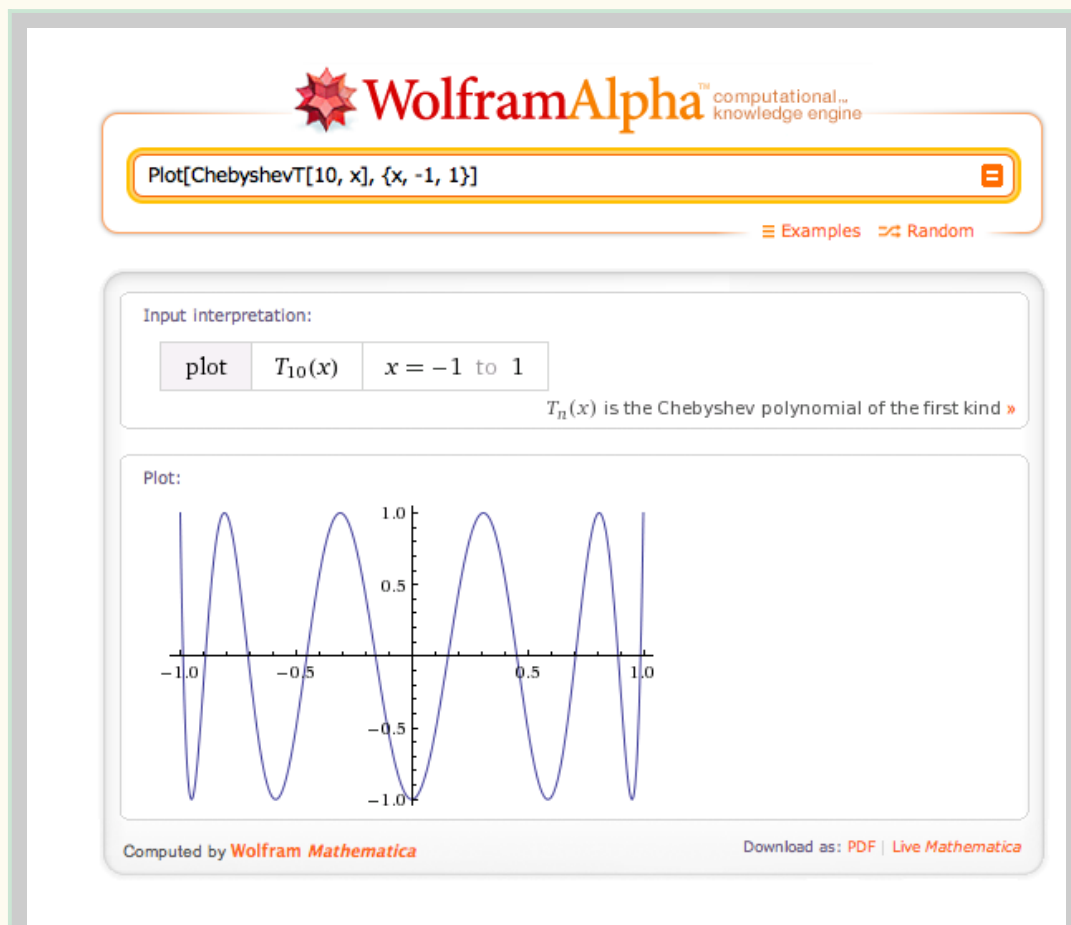
Example. The following function computes the 10th degree Chebyshev polynomial and the next one draws the function on the interval [-1, 1].

```
ChebyshevT[10, x]
```

```
- 1 + 50 x2 - 400 x4 + 1120 x6 - 1280 x8 + 512 x10
```

Let's see what happens when we use the same input in WolframAlpha:



Note the two links at the lower right hand corner: Download as PDF and Live *Mathematica*. The first one is obvious. The second one needs the CDF Player plug-in to be installed.

In general, Mathematica notebooks allow importing and exporting of many formats. One can prepare even a slide show in *Mathematica*.

■ Kernel and FrontEnd
















Mathematica consists of two independent computational environments called the FrontEnd and the Kernel, which communicate by means of a protocol called *MathLink*. The Kernel does all the computations. The FrontEnd is what you see in front of you, including the window, menu, etc. You can use many FrontEnds with one Kernel but the usual FrontEnd is what is known as a notebook FrontEnd (there are also ASCII front ends you can run using a terminal interface).

The Kernel is the basic programming environment and in fact it can be used to completely control the FrontEnd. We will give a few examples, but we will not use much of this. For example:


```
nb1 = CreateDocument[{Plot[x^2, {x, -1, 1}]}]
```

```
NotebookObject[ Untitled-18]
```

```
ls = Notebooks[]
```

```
{NotebookObject[ Writing Assistant],
 NotebookObject[ Foundations of Programming in Mathematica Part 1],
 NotebookObject[ Untitled-18], NotebookObject[ Untitled-13],
 NotebookObject[ Untitled-11], NotebookObject[ Untitled-15],
 NotebookObject[ NotebookClose - Wolfram Mathematica],
 NotebookObject[ Installed Add-ons - Wolfram Mathematica], NotebookObject[ Untitled-7],
 NotebookObject[ Untitled-6], NotebookObject[ Untitled-5],
 NotebookObject[ Untitled-4], NotebookObject[ Untitled-3],
 NotebookObject[ Untitled-2], NotebookObject[ Messages]}
```

```
SelectedNotebook[]
```

```
NotebookObject[ Foundations of Programming in Mathematica Part 1]
```

```
SetSelectedNotebook[ls[[3]]]
```

```
NotebookObject[ Untitled-18]
```

The FrontEnd itself can also be “programmed” independently of the Kernel. This will be more important for us later, in building interfaces.

■ Mathematica notebooks

Mathematica is one of the largest single application programs ever developed, and it contains a vast array of algorithms and important technical innovations. Among these innovations is the concept of platform - independent interactive documents known as notebooks.

Every Mathematica notebook is a complete interactive document combining text, tables, graphics, calculations, and other elements. A Mathematica notebook consists of a list of cells, which you can group (sections etc).

Exercise. Click on different brackets on the right in this notebook with a right mouse button to find out the style being used.

Palettes and buttons provide a simple but fully customizable point - and - click interface to Mathematica (for Greek symbols, signs of integral, simple build - in functions, etc).

Recently Wolfram Research has expanded the concept of a notebook by introducing a new document format called CDF ("Computable Document Format") which unlike traditional notebooks allows interactive "dynamic" content.

■ The Unifying Idea of Mathematica

Mathematica is built on the powerful unifying idea that everything can be represented as a symbolic expression.

■ Main Features of Mathematica

Once one starts experimenting in Mathematica, one immediately notices some of its main features.

1. One important feature of Mathematica that differs from other computer languages, and from conventional mathematical notation, is that function arguments are enclosed in square brackets, not parentheses. Parentheses in Mathematica are reserved specifically for indicating the grouping of terms. There is obviously a need to distinguish giving arguments to a function from grouping terms together.
2. Names of built-in functions start with a capital letter.
3. Multiplication is represented either by * or by a space.
4. Powers are denoted by ^.
5. Numbers in scientific notation are entered, for example, as 2.5×10^{-4} or $2.5 \cdot 10^{-4}$.
6. There is a general convention in Mathematica that all function names are spelled out as full English words, unless there is a standard mathematical abbreviation for them. The great advantage of this scheme is that it is predictable. Once you know what a function does, you will usually be able to guess exactly what its name is. If the names were abbreviated, you would always have to remember which shortening of the standard English words was used.
7. Another feature of built - in Mathematica names is that they all start with capital letters. The capital letter convention makes it easy to distinguish built - in objects. If Mathematica used max instead of Max to represent the operation of finding a maximum, then you would never be able to use max as the name of one of your variables. In addition, when you read programs written in Mathematica, the capitalization of built - in names makes them easier to pick out.
8. N is a function that turns exact numbers and certain symbols into approximate numbers. For example:

```
N[Pi]
```

```
3.14159
```

```
N[Sqrt[2], 30]
```

```
1.41421356237309504880168872421
```

Hence N cannot be used for a function or a variable name. The same is true of some other symbols

written with a capital letter (e.g. E,C). For that reason it is important to follow the convention that user defined symbols begin with a small letter.

A quick access to help information is achieved by typing the question mark :

? FullForm

FullForm[*expr*] prints as the full form of *expr*, with no special syntax. >>

? Part

expr[[*i*]] or Part[*expr*, *i*] gives the i^{th} part of *expr*.
expr[[$-i$]] counts from the end.
expr[[*i*, *j*, \ddots]] or Part[*expr*, *i*, *j*, \ddots] is equivalent to *expr*[[*i*]][[*j*]][\ddots].
expr[[{*i*₁, *i*₂, \ddots }]] gives a list of the parts *i*₁, *i*₂, \ddots of *expr*.
expr[[*m* ;; *n*]] gives parts *m* through *n*.
expr[[*m* ;; *n* ;; *s*]] gives parts *m* through *n* in steps of *s*. >>

The quick access to help is also by highlighting the word and then pressing F1.

To get help for the command/operator you know you need to type ? and the command/operator. If you do not know the operator, search the Help Browser.

? >>

expr >> filename writes *expr* to a file. Put[*expr*₁, *expr*₂, ... , "filename"] writes a sequence of expressions *expr*_{*i*} to a file. More...

Mathematica understands lists as {*a*, *b*, *c*} (in the full form it is List[*a*, *b*, *c*]). One can learn later on that many objects in *Mathematica* are written by using lists. For instance, a matrix can be inserted in the following way: go to the main menu; insert; tables/matrices:

```
□ □ □
□ □ □
□ □ □
```

Next one just needs to put the brackets and fill in the matrix elements by clicking on each empty square :

```
( 1 2 □ )
( □ □ □ )
( □ □ □ )
```

The result is :

```
( 1 2 3 )
( 2 3 4 )
( 5 6 7 )
```

The same matrix can be entered like this :

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

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

```
% // MatrixForm
```

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

Here % means the last expression.

```
2 + 2
```

```
4
```

```
% + 4
```

```
8
```

```
%% + 2
```

```
6
```

Here %% means the last but one expression.

Example.

```
Plus[Power[x, 2], Sqrt[x]]
```

$$\sqrt{x} + x^2$$

The same can be entered by either using palettes or by the following sequence : Control key + 2 gives $\sqrt{\square}$; next one needs to type in "x" ; this gives \sqrt{x} ; next + x. To type in the square one can type in alt + 6 which gives \square and then type in 2 in the empty square.

One can type many symbols without using palettes. For instance, to type in π , one needs to press esc then type in pi then press esc once again.

Example.

The use of Ctrl+6:

```
x2
```

esc+i+i+esc

```
i
```

esc+pi+esc

```
π
```

Alt+7 (applying to the blue bracket on the right): gives text in the notebook.

A very useful trick is the formula completion feature. Suppose, for example, you wish to use a function whose name begins with Plot but you can't quite remember the rest of it. Just type in the beginning of

the name and press the Control key (Command key on the Macintosh) and the letter K. You will see a pop up menu of all functions whose name begins with Plot. If you decide you want to use the function Plot3D you can type the name Plot3D and press Control and Shift keys together with the K key. You will see a template:

```
Plot3D[f, {x, xmin, xmax}, {y, ymin, ymax}]
```

■ An overview of programming techniques

For most of the more complex problems that one wants to solve with Mathematica, one has to create Mathematica programs oneself. Mathematica supports several styles of programming, and one is free to choose the one, one is most comfortable with. However, it turns out that no single type of programming suits all cases equally. As a result, it is useful to learn several different types of programming.

Traditional programming language such as C or Fortran use procedural programming (assignments and loops such as Do, For, While and so on). They also exist in *Mathematica*. But while any Mathematica program can, in principle, be written in a procedural way, this is rarely the best approach. In a symbolic system like Mathematica, functional and rule - based programming typically yields programs that are more efficient, and easier to understand.

Some types of programming :

Procedural Programming

List - based Programming (Many operations are automatically threaded over lists, a starting point to learn).

Functional Programming

Rule - Based Programming

Mixed Programming Paradigms

There are typically many different ways to formulate a given problem in Mathematica. In almost all cases, however, the most direct, precise and simple formulations will be best.

There are dozen of definitions of the factorial function (see later on).

■ Expressions

All objects in *Mathematica* programming language are expressions. For example

```
a + b
```

```
a + b
```

```
{2, 3, 5}
```

```
{2, 3, 5}
```

```
StringTake["hello", 4]
```

```
hell
```

```
Sin[x]
```

```
Sin[x]
```

```
Sin[ $\pi$ ]
```

```
0
```

```
First[{a, b, c}]
```

```
a
```

are all different kinds of expressions. These expressions often look like mathematical formulas (more about that later on), which makes them more understandable and memorable to humans, but actually that have an internal form that is very simple and very consistent. It is called the “Full Form” of an expression and can be seen by applying the function `FullForm` to it (but there is a caveat, see below).

■ FullForm of expressions

Each expression is either an Atom or has the form

```
F[a1, a2, ..., an]
```

where `F` is called the Head of the expression and `a1`, `a2`, are expressions. Examples of atoms are `2`, `a`, `3/4`, `3.2`, `"cat"`. Whether something is an atom can be tested with the function `AtomQ`:

```
AtomQ[2]
```

```
True
```

```
AtomQ[{2, 3}]
```

```
False
```

Expressions often do not look like their FullForms, for example `a+b` has FullForm:

```
FullForm[a + b]
```

```
Plus[a, b]
```

```
Head[a + b]
```

```
Plus
```

```
Head[a]
```

```
Symbol
```

Head[2]

Integer

FullForm[{a, b}]

List[a, b]

Head[{a, b}]

List

Note that atoms also have Head:

Head[2]

Integer

Head["cat"]

String

Head[cat]

Symbol

Note also that :

Head[x]

Symbol

x = 1

1

x + 2

3

Head[x]

Integer

Evaluation of x to 1 caused this to happen. You can see the original Head by preventing evaluating e.g.


```
Head[Unevaluated[x]]
```

```
Symbol
```

```
2 + 3
```

```
5
```

```
FullForm[Hold[2 + 3]]
```

```
Hold[Plus[2, 3]]
```

```
ReleaseHold[%]
```

```
5
```

The full form of

```
x = 1
```

is

```
FullForm[Hold[x = 1]]
```

```
Hold[Set[x, 1]]
```

```
Clear[x]
```

It is important to distinguish the assignment Set from Equal, which is usually written as == and has

```
FullForm[Hold[x == 1]]
```

```
Hold[Equal[x, 1]]
```

```
Clear[x]
```

```
x == 1
```

```
x == 1
```

```
x = 1;
```

```
x == 2
```

```
False
```

```
Clear[x]
```

■ Parts of Expressions

A very important skill is extracting parts of expressions. An expression is really a tree-like object, as can be seen using the function `TreeForm`:

? TreeForm

`TreeForm[expr]` displays *expr* as a tree with different levels at different depths.

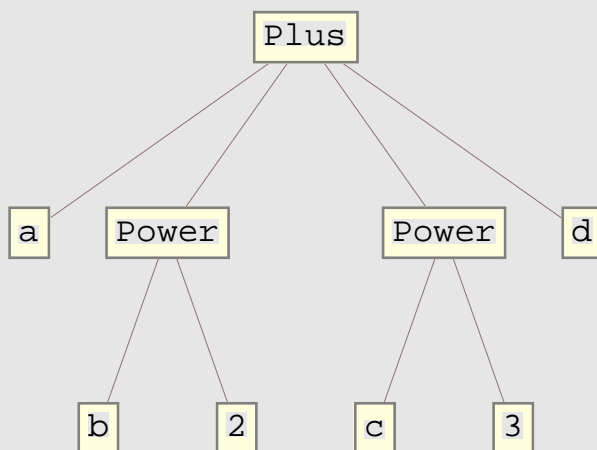
`TreeForm[expr, n]` displays *expr* as a tree only down to level *n*. >>

$g = a + b^2 + c^3 + d;$

FullForm[g]

`Plus[a, Power[b, 2], Power[c, 3], d]`

TreeForm[g]



Level[g, {1}]

`{a, b2, c3, d}`

Level[g, {2}]

`{b, 2, c, 3}`

Part[g, 0]

`Plus`

```
g[[1]]
```

```
a
```

```
g[[2]]
```

```
b2
```

```
g[[3]]
```

```
c3
```

```
g[[3, 3]]
```

Part::partw : Part 3 of c³ does not exist. >>

```
(a + b2 + c3 + d) [[3, 3]]
```

```
g[[2, 2]]
```

```
2
```

```
g[[2, 0]]
```

```
Power
```

and so on.

You can also do this from the back :

```
g[[-2, 1]]
```

```
c
```

```
g
```

```
a + b2 + c3 + d
```

```
g[[1 ;; 3]]
```

```
a + b2 + c3
```

```
a + b2 + c3
```

```
g[[2 ;; 4]]
```

$$b^2 + c^3 + d$$

Now, here comes a very nice and important fact: you can change an expression by an assignment to a part of it. For example;

```
g[[1]] = x + y;
```

```
g
```

$$b^2 + c^3 + d + x + y$$

```
g[[3 ;; 4]] = z; g
```

$$1 + b^2 + y + 2 z$$

■ List, Vectors, Matrices, Tensors

A very important thing to notice that in *Mathematica* lists are just expressions with Head List:

```
m = {a, b, c, d};
```

```
Length[m]
```

```
4
```

```
List[a, b, c, d]
```

```
{a, b, c, d}
```

A matrix is simply a list of lists of the same length:

```
mat = {{a, b, d}, {c, d, e}}
```

$$\begin{pmatrix} a & b & d \\ c & d & e \end{pmatrix}$$

```
mat[[1, 1]]
```

```
a
```

```
mat[[2, 2]]
```

```
d
```

```
mat[[All, 1]]
```

```
{a, c}
```

```
mat[[All, 2]]
```

```
{b, d}
```

```
mat[[1, All]]
```

```
{a, b}
```

```
mat[[2, All]]
```

```
{c, d}
```

We will later see how to easily create arbitrarily large matrices using the functions `Table` and `Array`.

```
Table[i2, {i, 1, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
Table[i * j, {i, 1, 5}, {j, 1, 5}]
```

```

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 & 10 \\ 3 & 6 & 9 & 12 & 15 \\ 4 & 8 & 12 & 16 & 20 \\ 5 & 10 & 15 & 20 & 25 \end{pmatrix}$$

```

▣ A note on forms of expressions.

We already know that a Mathematica expression often looks different to human eyes than its internal form (`FullForm`). However, the situation is made more complicated, by the fact that traditional mathematical notation is not unambiguous. Because of this and for reasons of history *Mathematica* has several “forms” of input and output. The first versions of *Mathematica* has only two forms: `InputForm`, which looked like a standard programming language (e.g. Fortran) way of writing mathematical formulas and `OutputForm`, which is a little more like usual mathematics and has become completely obsolete (it retained only for reasons of compatibility with very early *Mathematica* notebooks). Since then they have both been replaced by `StandardForm` and `TraditionalForm`. `StandardForm` retains the basic principles of `InputForm` but allows more usual mathematical expressions. `TraditionalForm` looks almost like the usual mathematical notation. One can convert between these forms using the `Convert To` sub menu in the `Cell` menu. One can also set the default forms for the `Input` and `Output` in the `Preferences` menu.

▣ Basic principles of `InputForm` (and `StandardForm`)

1. All built in functions start with a capital letter.
2. Square brackets `[]` are used as function brackets.
3. (`InputForm`) The basic arithmetical operations are denoted by `+` (addition), `*` or space (multiplication) / (division), `^` (power).

4. There are the following inclusions $\text{InputForm} \subset \text{StandardForm} \subset \text{TraditionalForm}$ but not in the opposite direction.

▣ Links

<http://reference.wolfram.com/Mathematica/guide/Expressions.html>

<http://reference.wolfram.com/mathematica/tutorial/FormsOfInputAndOutput.html>

■ 2. Working with Lists

One of the most common expressions in Mathematica are lists.

```
Solve[x3 == 1, x]
```

```
{ {x → 1}, {x → -(-1)1/3}, {x → (-1)2/3}}
```

```
% // N
```

```
{ {x → 1.}, {x → -0.5 - 0.866025 i}, {x → -0.5 + 0.866025 i}}
```

```
CoefficientList[a x2 + b x + c, x]
```

```
{c, b, a}
```

```
Options[Plot]
```

```
{AlignmentPoint → Center, AspectRatio →  $\frac{1}{\text{GoldenRatio}}$ , Axes → True,
  AxesLabel → None, AxesOrigin → Automatic, AxesStyle → {}, Background → None,
  BaselinePosition → Automatic, BaseStyle → {}, ClippingStyle → None,
  ColorFunction → Automatic, ColorFunctionScaling → True, ColorOutput → Automatic,
  ContentSelectable → Automatic, CoordinatesToolOptions → Automatic,
  DisplayFunction → $DisplayFunction, Epilog → {}, Evaluated → Automatic,
  EvaluationMonitor → None, Exclusions → Automatic, ExclusionsStyle → None,
  Filling → None, FillingStyle → Automatic, FormatType → TraditionalForm,
  Frame → False, FrameLabel → None, FrameStyle → {}, FrameTicks → Automatic,
  FrameTicksStyle → {}, GridLines → None, GridLinesStyle → {}, ImageMargins → 0.,
  ImagePadding → All, ImageSize → Automatic, ImageSizeRaw → Automatic, LabelStyle → {},
  MaxRecursion → Automatic, Mesh → None, MeshFunctions → {#1 &}, MeshShading → None,
  MeshStyle → Automatic, Method → Automatic, PerformanceGoal → $PerformanceGoal,
  PlotLabel → None, PlotPoints → Automatic, PlotRange → {Full, Automatic},
  PlotRangeClipping → True, PlotRangePadding → Automatic, PlotRegion → Automatic,
  PlotStyle → Automatic, PreserveImageOptions → Automatic, Prolog → {},
  RegionFunction → (True &), RotateLabel → True, Ticks → Automatic,
  TicksStyle → {}, WorkingPrecision → MachinePrecision}
```

Let us also recall that the matrix is entered by using lists :

```
{{1, 2, 3}, {2, 3, 4}, {34, 5, 3}} // MatrixForm
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 34 & 5 & 3 \end{pmatrix}$$

Let us learn how to generate lists and what basic operations one can perform with them. Another useful command is Table

```
Table[i^2 + 2, {i, -1, 2}]
```

```
{3, 2, 3, 6}
```

```
% // TableForm
```

```
3  
2  
3  
6
```

One can generate not only numbers but also other expressions :

```
Array[a, 3]
```

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

Some commonly used objects are already defined in Mathematica. For instance, the identity matrix :

```
IdentityMatrix[3] // MatrixForm
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

For the matrices *Mathematica* has a lot of build - in operations

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

```
{4, -1}
```

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

```
{{2, 3}, {-1, 1}}
```

Basic operations for the lists include the following :

```
{1, 2, 3} + {1, 2, 3}
```

```
{2, 4, 6}
```

```
{1, 2, 3} + 1
```

```
{2, 3, 4}
```

A scalar product is given by a dot

```
{a, b, c} . {s, d, f}
```

```
b d + c f + a s
```

However, one needs to be careful with length of the objects.

```
{1, 2, 3} + {1, 2, 3, 4}
```

Thread::tdlen : Objects of unequal length in {1, 2, 3} + {1, 2, 3, 4} cannot be combined. >>

```
{1, 2, 3} + {1, 2, 3, 4}
```

Other useful operations include

```
Prepend[{a, b, c}, d]
```

```
{d, a, b, c}
```

```
Append[{a, b, c}, d]
```

```
{a, b, c, d}
```

```
Union[{a, b, c}, {a, b, d}]
```

```
{a, b, c, d}
```

```
Join[{a, b, c}, {a, b, d}]
```

```
{a, b, c, a, b, d}
```

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

```
{a, b}
```

Also have a look at commands Insert, Delete and many others in the help. The name of the command suggests unambiguously what it performs with a given list.

To get an element of the list one indicates its position.

```
{a, b, c}[[1]]
```

```
a
```



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

```
d
```

Here - 1 means the first element counted from the end.

```
{a, b, c, d}][[2]]
```

```
b
```

If you do not know how many elements are in the list, you can always verify this by using Length

```
Length[{a, b, v}]
```

```
3
```

```
Length[{a, b, {v, w}}]
```

```
3
```

A similar command for the dimension of the list is Dimensions

```
Dimensions[{a, b, {v, w}}]
```

```
{3}
```

```
Dimensions[{{a, b}, {v, w}}]
```

```
{2, 2}
```

This counts the elements of the first level in the list.

In applications one often encounters the problem to verify whether a given element is in the list and if so, one might require further its position.

```
Position[{{a, b, c}, {a, f, g}}, a]
```

```
{{1, 1}, {2, 1}}
```

Here Position takes account of the nesting of lists.

Since the lists can be nested, it is useful to know that they can always be flattened.

```
? Flatten
```

Flatten[list] flattens out nested lists.

Flatten[list, n] flattens to level n .

Flatten[list, n, h] flattens subexpressions with head h .

Flatten[list, {{s₁₁, s₁₂, É }, {s₂₁, s₂₂, É }, É]]

flattens list by combining all levels s_{ij} to make each level i in the result. >>

```
{{a, b, c}, {a, f, g}} // Flatten
```

```
a, b, c, a, f, g
```

To get rid of repeated elements one uses Union

```
% // Union
```

```
a, b, c, f, g
```

From a given list one can get a list of permutations and other lists of a given length with all elements of the original list

```
Permutations[{a, b, c}]
```

```
{{a, b, c}, {a, c, b}, {b, a, c}, {b, c, a}, {c, a, b}, {c, b, a}}
```

```
Tuples[{0, 1}, 3]
```

```
{{0, 0, 0}, {0, 0, 1}, {0, 1, 0}, {0, 1, 1}, {1, 0, 0}, {1, 0, 1}, {1, 1, 0}, {1, 1, 1}}
```

```
Accumulate[{a, b, c, d, e, f}]
```

```
a, a+b, a+b+c, a+b+c+d, a+b+c+d+e, a+b+c+d+e+f
```

■ Apply and Map

```
? Apply
```

Apply[f, expr] or f @@ expr replaces the head of expr by f.

Apply[f, expr, levelspec] replaces heads in parts of expr specified by levelspec. >>

Let us form a new expression from the list and the other way round.

```
FullForm[{a, b, c}]
```

```
List[a, b, c]
```

```
Times@@{a, b, c}
```

```
a b c
```

```
FullForm[a b c]
```

```
Times[a, b, c]
```

```
List@@(a b c)
```

```
{a, b, c}
```

Another example is

```
Plus@@{a, b, c}
```

```
a + b + c
```

A more complicated example is to generate a list of coefficients (maybe useful for polynomial expressions)

```
Subscript[A, #] & /@ Table[i, {i, 1, 10}]
```

```
{A1, A2, A3, A4, A5, A6, A7, A8, A9, A10}
```

Here /@ means Map.

```
? Map
```

Map[f, expr] or f /@ expr applies f to each element on the first level in expr.

Map[f, expr, levelspec] applies f to parts of expr specified by levelspec. >>

Here there is a trivial example of forming a list.

```
Function[x, x^2] /@ Table[i, {i, 1, 10}]
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

```
k = Table[i, {i, 1, 10}]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

Therefore, the command /@ works as follows. It applies the function $x \mapsto x^2$ to every element of the list k. (Here we meet an example of a pure function, the concept which will be discussed below.)

Evaluation

A very important concept in *Mathematica* is that of evaluation. In *Mathematica* evaluation always takes place after you write some input and press Shift + Enter. The process of evaluation is quite complicated, and follows a definite sequence of steps. Understanding this process is important in advanced *Mathematica* programming and we will return to this in the future. Often the evaluation process takes place even if nothing seems to happen. For example:

```
FullForm[Hold[2/3]]
```

```
Hold[Times[2, Power[3, -1]]]
```

$$\frac{2}{3}$$
$$\frac{2}{3}$$

```
FullForm[2 / 3]
```

```
Rational[2, 3]
```

```
2 + 3 I
```

```
2 + 3 i
```

```
FullForm[Hold[2 + 3 I]] // InputForm
```

```
FullForm[Hold[2 + 3*I]]
```

```
FullForm[2 + 3 i]
```

```
Complex[2, 3]
```

```
AtomQ[Complex[2, 3]]
```

```
True
```

```
Head[Unevaluated[2 + 3 I]]
```

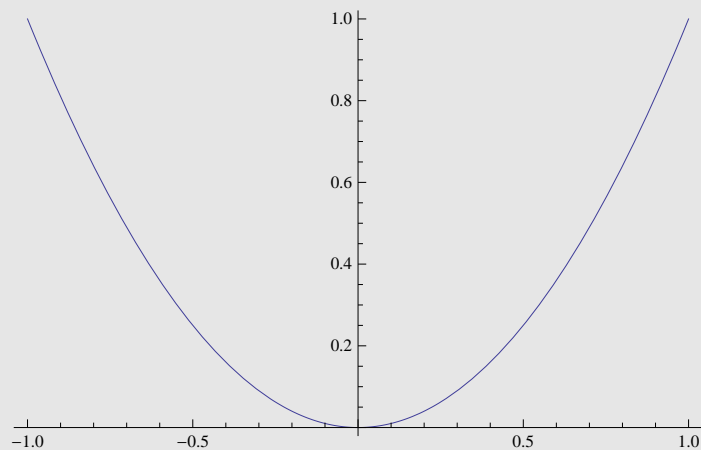
```
Plus
```

```
Head[2 + 3 I]
```

```
Complex
```

An interesting special case are graphics.

```
gr = Plot[x^2, {x, -1, 1}]
```



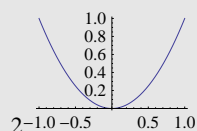
```
Short[InputForm[gr], 5]
```

```
Graphics[
  {{{}}, {{}}, {Hue[0.67, 0.6, 0.6], Line[{{-0.9999999591836735, 0.9999999183673486},
    << 272 >>, {0.9999999591836735, << 1 >>}}]}], {{<< 6 >>}}
```

We see that a plot of a function is also a *Mathematica* Graphics object. One can therefore use the *Mathematica* programming language to control every detail of a graphic. Graphic programming in *Mathematica* is a whole big subject, but we will see a few examples later on.

It is possible to think of *Mathematica* as an algebraic object, somewhat like a ring, with partial addition and multiplication. This means that you can perform algebraic operations which are purely formal, for example, you can raise a number to the power of a graphic:

```
2gr
```



In some situations arithmetical operations on objects of different kind are defined, for example, it is possible to add a number (or a symbol) to a list:

```
{1, 2, 3} + 1
```

```
{2, 3, 4}
```

However, in certain cases, trying to perform such an operation on objects of different kind will cause a error message:

```
{1, 2, 3} + {1, 3}
```

Thread::tdlen : Objects of unequal length in {1, 2, 3} + {1, 3} cannot be combined. >>

```
{1, 3} + {1, 2, 3}
```

Here is an example of abstract algebraic manipulation performed on strings:

```
Distribute[("cat" + "dog") * "mouse"]
```

```
cat mouse + dog mouse
```

■ The evaluation loop.

When you enter an input expression *Mathematica*'s Kernel evaluates in a very definite order. Understanding this order is important for *Mathematica* programming. The evaluation order will be described carefully later once we learn about rules and patterns. However, the basic idea is this: *Mathematica* evaluates each part of the expression by turn, starting with the Head. It applies all rules it knows for the expression, first user defined then built in ones, until it can no longer find a rule. Then it stops and "returns" the result. (Sometimes this evaluation will not stop and we get into an infinite loop. Actually *Mathematica* will almost always detect such situations and will stop, unless we change the defaults to make it run for ever).

Programming using Patterns and Rules

Mathematica allows many different styles of programming. There is one style that, although not unique to *Mathematica*, distinguishes it from most other similar systems. This is the possibility of using "patterns" and "re-write rules" or just "rules".

The basic concepts in this kind of programming are "rule" and "pattern". Rules can be local and global.

■ Local Rules

A local rule always has the form

```
? Rule
```

$lhs \rightarrow rhs$ or $lhs \Rightarrow rhs$ represents a rule that transforms lhs to rhs . >>

or

```
? RuleDelayed
```

$lhs := rhs$ or $lhs \Rightarrow rhs$ represents a rule that transforms lhs to rhs , evaluating rhs only after the rule is used. >>

Note that:

FullForm[lhs → rhs]

Rule[lhs, rhs]

FullForm[lhs :> rhs]

RuleDelayed[lhs, rhs]

The difference between Rule and RuleDelayed will be explained below. Most often Rule is used together with the function ReplaceAll (see also Replace):

? ReplaceAll

expr /. rules applies a rule or list of rules in an attempt to transform each sub part of an expression expr. **More...**

Here are some examples of using rules (in some of these examples the output appears in Traditional-Form).

$x^2 + \sin[xy] + 3 /. \{x \rightarrow \pi, y \rightarrow 2\pi\}$

$3 + \pi^2 + \sin(2\pi^2)$

Now we use a more general rule. This time we use a "pattern"

? _

_ or Blank[] is a pattern object that can stand for any Mathematica expression.

_h or Blank[h] can stand for any expression with head h. >>

$x^2 + \sin[xy] + 3 /. _ \rightarrow \pi$

π

The reason for the above result is that ReplaceAll starts looking for a match starting at the top level of the expression and when it finds a match it stops looking for more. If we want to find a match at a different level we can use the function Replace with a level specification. For example, here we replace everything on level 3 of the expression with π .

Replace $[x^2 + \sin[xy] + 3, _ \rightarrow \pi, \{3\}]$

$x^2 + 3 + \sin(\pi^2)$

Level $[x^2 + \sin[xy] + 3, \{3\}]$

$\{x, y\}$

Replace $[x^2 + \sin[xy] + 3, _ \rightarrow \pi, \{2\}]$

$3 + \pi^\pi$

$$x^2 + \text{Sin}[x y] + 3 /. \text{Sin}[x_] \rightarrow \text{Sin}[x^2]$$

$$\sin(x^2 y^2) + x^2 + 3$$

$$x^2 + \text{Sin}[x y] + 3 /. _? (\# > 2 \&) \rightarrow \pi$$

$$x^2 + \sin(x y) + \pi$$

$$x^2 + \text{Sin}[x y] + 3 /. a_ /; a > 2 \rightarrow \text{Pi}$$

$$x^2 + \sin(x y) + \pi$$

$$x^2 + \text{Sin}[x y] + 3 /. _? (\text{AtomQ}[\#] \&) \rightarrow \pi$$

$$\pi(\pi, \pi(\pi, \pi), \pi(\pi(\pi, \pi)))$$

$$x^2 + \text{Sin}[x y] + 3 /. x_ \text{Times} \rightarrow \pi/4$$

$$x^2 + \frac{1}{\sqrt{2}} + 3$$

$$\text{FullForm}[x^2 + \text{Sin}[x y] + 3]$$

$$\text{Plus}[3, \text{Power}[x, 2], \text{Sin}[\text{Times}[x, y]]]$$

$$x^2 + \text{Sin}[x y] + 3 /. _ \text{Power} \rightarrow \pi/4$$

$$\sin(x y) + \frac{\pi}{4} + 3$$

These examples illustrate some of the very many ways of forming patterns in *Mathematica*. The most basic pattern is `x_` which stands for anything that is (locally) assigned the name `x`.

Here is an example where `Rule` and `RuleDelayed` give different answers:

$$a = 0;$$

$$\text{Sin}[2] /. a_ \text{Integer} \rightarrow a^2$$

$$0$$

$$\text{Sin}[2] /. a_ \text{Integer} \rightarrow a^2$$

$$\sin(4)$$

Before using a rule it is a good idea to look at the `FullForm` of an expression. Here are some possible “traps”:


```
Clear[a]
```

$$\sqrt{2} + \frac{1}{\sqrt{2}} /. \sqrt{2} \rightarrow a$$

$$a + \frac{1}{\sqrt{2}}$$

```
FullForm[ $\sqrt{2} + \frac{1}{\sqrt{2}}$ ]
```

```
Plus[Power[2, Rational[-1, 2]], Power[2, Rational[1, 2]]]
```

```
Unevaluated[ $\sqrt{2} + \frac{1}{\sqrt{2}}$ ] /. HoldPattern[ $\sqrt{2}$ ] → a
```

$$a + \frac{1}{a}$$

```
 $\sqrt{2} + \frac{1}{\sqrt{2}} /. 2^{\text{Rational}[x_, y_]} \rightarrow a^x$ 
```

$$a + \frac{1}{a}$$

```
 $\sqrt{2} + \frac{1}{\sqrt{2}} /. \left\{ \sqrt{2} \rightarrow a, \frac{1}{\sqrt{2}} \rightarrow 1/a \right\}$ 
```

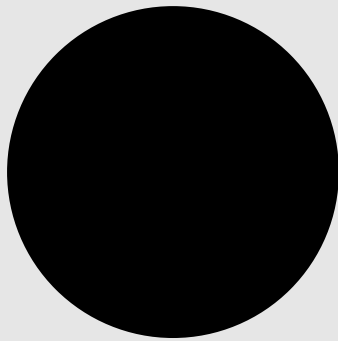
$$a + \frac{1}{a}$$

Rule based programming is very convenient when dealing with graphics.

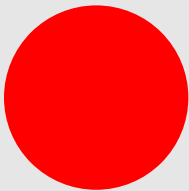
```
gr = Disk[{0, 0}, 1]
```

```
Disk[{0, 0}, 1]
```

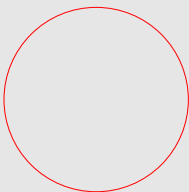
Graphics[gr, ImageSize → Small]



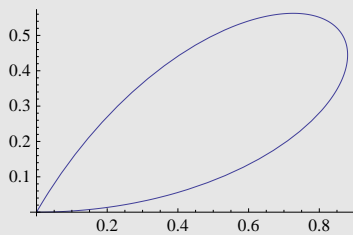
Graphics[{Red, gr}, ImageSize → Tiny]



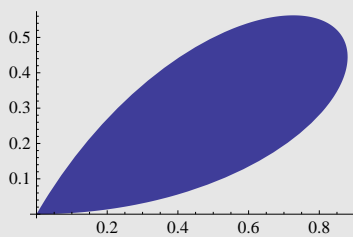
Graphics[{Red, gr}, ImageSize → {100, 100}] /. Disk → Circle



gr2 = PolarPlot[Sin[3 θ], { θ , 0, $\frac{\pi}{3}$ }, ImageSize → Small]



gr2 /. Line → Polygon



Many *Mathematica* functions return a list of rules as the output.

```
rules = Solve[x3 + 3 x + 4 == 0, x]
```

```
{ {x → -1}, {x →  $\frac{1}{2}(1 - i\sqrt{15})$ }, {x →  $\frac{1}{2}(1 + i\sqrt{15})$ }}
```

Note that this is actually a list of lists, each containing one rule.

This is very convenient, because we can use `ReplaceAll` to substitute these rules into other formulas. For example:

```
x /. rules
```

```
{ -1,  $\frac{1}{2}(1 - i\sqrt{15})$ ,  $\frac{1}{2}(1 + i\sqrt{15})$  }
```

```
x3 + 3 x + 4 /. rules // Simplify
```

```
{0, 0, 0}
```

```
Simplify[%]
```

```
{0, 0, 0}
```

Here is a similar example with `FindRoot` instead of `Solve`

```
FindRoot[x e-x == 0.2, {x, 0.1}]
```

```
{x → 0.259171}
```

```
x Exp[-x] == 0.2 /. %
```

```
True
```

▣ Links

[http : // reference.wolfram.com/mathematica/tutorial/PatternsAndTransformationRules.html](http://reference.wolfram.com/mathematica/tutorial/PatternsAndTransformationRules.html)

■ Global Rules ("Functions")

Here is one way to define a "function" in *Mathematica*:

```
Clear[f]
```

```
f[x_] := x2
```

f[1]

- 1

Clear[a]**f[a]** a^2 **f[3]**

9

?f

Global`f

f[x_] := x²**DownValues[f]**{HoldPattern[f(x_)] \rightarrow x²}

Although people often call f defined in this way a function, actually it is only a "global rule". More precisely, when a definition of this kind is evaluated, Mathematica creates a rule for the symbol f , which it uses every time when f is used. The rule is stored as a DownValue of f :

DownValues[f]{HoldPattern[f(x_)] \rightarrow x²}{HoldPattern[f(x_)] \rightarrow x²}

Here $x_$ is a "pattern", which stands for "anything", with a temporary name "x". The rule says "change $f(\text{anything})$ to anything²". HoldPattern prevents evaluation of $f(x_)$ (which would otherwise be replaced by $x_$ ² but $f[x_]$ is treated as a pattern for matching purposes.

So what happens when we evaluate definitions of this kind is this: *Mathematica* makes certain rules, stores them, and then applies them in a certain order. Here is an example:

Clear["Global`*"]**f[x_Real] := 3****f[1] := 2****f[x_Symbol] := x²**

```
f[x_Integer] := 5
```

```
DownValues[f]
```

```
{HoldPattern[f(1)] :> 2, HoldPattern[f(x_Real)] :> 3, HoldPattern[f(x_Symbol)] :> x2, HoldPattern[f(x_Integer)] :> 5}
```

```
DownValues[f] = Rest[DownValues[f]]
```

```
{HoldPattern[f(x_Real)] :> 3, HoldPattern[f(x_Symbol)] :> x2, HoldPattern[f(x_Integer)] :> 5}
```

```
DownValues[f]
```

```
{HoldPattern[f(x_Real)] :> 3, HoldPattern[f(x_Symbol)] :> x2, HoldPattern[f(x_Integer)] :> 5, HoldPattern[f(x_)] :> 0}
```

```
f[2 / 3]
```

```
0
```

```
f[x_] := 0
```

```
f["cat"]
```

```
f(cat)
```

```
f[1]
```

```
5
```

```
f[x_] := 0
```

```
f["dog"]
```

```
0
```

```
Clear[f]
```

```
DownValues[f]
```

```
{}
```

```
f[2]
```

```
5
```

$f[7]$

5

 $f[1.1]$

3

 $f[a]$ a^2 **$f["cat"]$** $f(\text{cat})$ **$\text{Map}[f, \{.5, 1, a\}]$** $\{3, 2, a^2\}$

The order in which rules are applied by *Mathematica* is roughly determined by two facts; more specific rules are applied before more general rules, and rules of equal generality are applied in the order they are entered.

In addition to `DownValues` there are also `OwnValues` and `UpValues` (and some other `Values`) created as follows:

`Clear[a]` **$a = 1; \text{OwnValues}[a]$** $\{\text{HoldPattern}[a] \rightarrow 1\}$ **`ClearAll[b];`** **$b /: \text{Sin}[b] = 2;$** **$b /: \text{Cos}[b] = 2;$** **$\cos^2(b) + \sin^2(b)$**

8

`UpValues[b]` $\{\text{HoldPattern}[\cos(b)] \rightarrow 2, \text{HoldPattern}[\sin(b)] \rightarrow 2\}$

When an expression is evaluated, *Mathematica* applies the rules contained in `UpValues`, `DownValues`, and so on in a certain order, after which it applies the built-in rules. It keeps evaluating the resulting

expression until it stops changing. Note also that certain built in rules are applied by *Mathematica* automatically on evaluation but others require using a special function such as `Simplify` or `FullSimplify`. For example the transformation

```
Clear[a]
```

$$a^n a^m$$

$$a^{m+n}$$

while

$$\cos(\alpha)^2 + \sin(\alpha)^2$$

$$\sin^2(\alpha) + \cos^2(\alpha)$$

does not automatically simplify to 1 but

```
Simplify[cos^2(b) + sin^2(b)]
```

```
1
```

Some simplifications only work with specific assumptions:

```
Simplify[Sqrt[x^2]]
```

$$\sqrt{x^2}$$

```
Assuming[x >= 0, Simplify[Sqrt[x^2]]]
```

```
x
```

```
Assuming[x <= 0, Simplify[Sqrt[x^2]]]
```

```
-x
```

Mathematica generally tries to apply any transformations it knows to an expression until it no longer changes. However, this is not the case when we use `ReplaceAll`. `ReplaceAll` looks for patterns in all the parts of an expression, but only looks for one match in each part. So, if we have only more than one rule, we may not obtain all the transformations we wish to get:

```
rules = {Log[x_ y_] -> Log[x] + Log[y], Log[x_^k_] -> k Log[x]};
```

```
Log[ $\sqrt{a (b c^d)^e}$ ] /.rules
```

$$\frac{1}{2} \log(a (b c^d)^e)$$

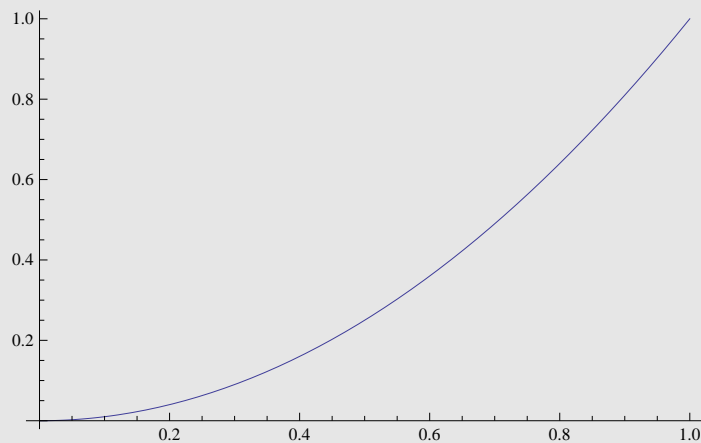
In order to obtain all transformations we should use `ReplaceRepeated` (`//.`) instead of `ReplaceAll` (`/.).`

```
Log[ $\sqrt{a (b c^d)^e}$ ] //.rules
```

$$\frac{1}{2} (\log(a) + e (\log(b) + d \log(c)))$$

Another important thing: Options of *Mathematica*'s functions are given as Rules.

```
Plot[x2, {x, 0, 1}]
```

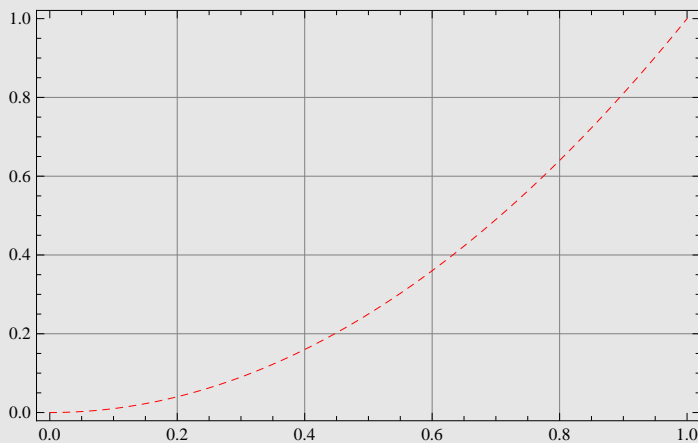


```
Options[Plot]
```

```
{AlignmentPoint → Center, AspectRatio →  $\frac{1}{\phi}$ , Axes → True, AxesLabel → None, AxesOrigin → Automatic,
  AxesStyle → {}, Background → None, BaselinePosition → Automatic, BaseStyle → {}, ClippingStyle → None,
  ColorFunction → Automatic, ColorFunctionScaling → True, ColorOutput → Automatic,
  ContentSelectable → Automatic, DisplayFunction → $DisplayFunction, Epilog → {}, Evaluated → Automatic,
  EvaluationMonitor → None, Exclusions → Automatic, ExclusionsStyle → None, Filling → None,
  FillingStyle → Automatic, FormatType → TraditionalForm, Frame → False, FrameLabel → None,
  FrameStyle → {}, FrameTicks → Automatic, FrameTicksStyle → {}, GridLines → None, GridLinesStyle → {},
  ImageMargins → 0., ImagePadding → All, ImageSize → Automatic, LabelStyle → {}, MaxRecursion → Automatic,
  Mesh → None, MeshFunctions → {#1 &}, MeshShading → None, MeshStyle → Automatic,
  Method → Automatic, PerformanceGoal → $PerformanceGoal, PlotLabel → None, PlotPoints → Automatic,
  PlotRange → {Full, Automatic}, PlotRangeClipping → True, PlotRangePadding → Automatic, PlotRegion → Automatic,
  PlotStyle → Automatic, PreserveImageOptions → Automatic, Prolog → {}, RegionFunction → (True &),
  RotateLabel → True, Ticks → Automatic, TicksStyle → {}, WorkingPrecision → MachinePrecision}
```



```
Plot[x^2, {x, 0, 1}, Axes → False, Frame → True, GridLines → Automatic, PlotStyle → {Red, Dashing[0.01]}]
```



■ The difference between := and =

The difference between := and = is exactly the same as that between \Rightarrow and \rightarrow . Note these FullForms:

```
FullForm[Hold[a = 3]]
```

```
Hold[Set[a, 3]]
```

```
FullForm[Hold[a := 3]]
```

```
Hold[SetDelayed[a, 3]]
```

Consider the following two definitions:

```
f[p_] := Expand[p]
```

```
g[p_] = Expand[p];
```

```
?=
```

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

```
?:=
```

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

If we apply them to an expression like $(a + b)^3$ we will get quite different results:

$$f[(a+b)^3]$$

$$a^3 + 3 b a^2 + 3 b^2 a + b^3$$

$$g[(a+b)^3]$$

$$(a+b)^3$$

The reason is that `=` evaluates the right hand side before assigning the evaluated value to the left hand side, while `:=` assigns the unevaluated right hand side to the left hand side.

▣ Links

<http://reference.wolfram.com/mathematica/tutorial/ManipulatingValueLists.html>

<http://reference.wolfram.com/mathematica/tutorial/ManipulatingOptions.html>

Functions and Functional Programming

■ Pure Functions

In addition to functions defined by means of global rules *Mathematica* also has "genuine functions", defined as follows:

$$\text{Function}[x, x^3][c]$$

$$c^3$$

Note that such a function does not need to have a name (so it is called an anonymous function), although we can of course give it a name:

$$f = \text{Function}[x, x^3];$$

$$f[3]$$

$$27$$

$$\text{Clear}[f]$$

$$\text{OwnValues}[f]$$

$$\{\text{HoldPattern}[f] \rightarrow \text{Function}[x, x^3]\}$$

We can also, of course, in the same way, construct functions of several variables.

```
Function[{x, y},  $x^3 + y^2$ ][4, 2]
```

```
68
```

There are two problems with this approach. First, it is inconvenient to use letters for variable names. This problem is solved by using the notation #1, #2 ,... for the first, second, third etc., arguments. Thus:

```
Function[#12 + #22][1, 5]
```

```
26
```

Lastly, the word Function can be replaced by the shorthand & after the end of the function, as in

```
#12 + #22 &[2, 3]
```

```
13
```

■ Predicates (Boolean Functions)

A common class of functions are functions whose value are the Boolean constants True and False. Such functions are called predicates. Most built in *Mathematica* predicates have names that end in Q:

```
PrimeQ[25]
```

```
False
```

```
PrimeQ[18]
```

```
False
```

```
EvenQ[7]
```

```
False
```

Here are two ways of defining a predicate that test is a number is larger than 5:

```
LargerThanFive[n_] :=  $n > 5$ 
```

```
LargerThanFive[1]
```

```
False
```

```
LargerThanFive[p]
```

```
 $p > 5$ 
```

Here is the same thing done using a pure function

```
# > 5 &[7]
```

```
True
```

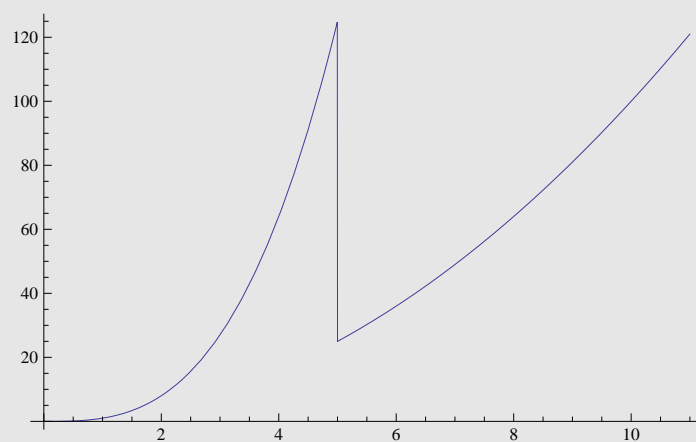
Such pure functions can be used in patterns:

```
Clear[f]
```

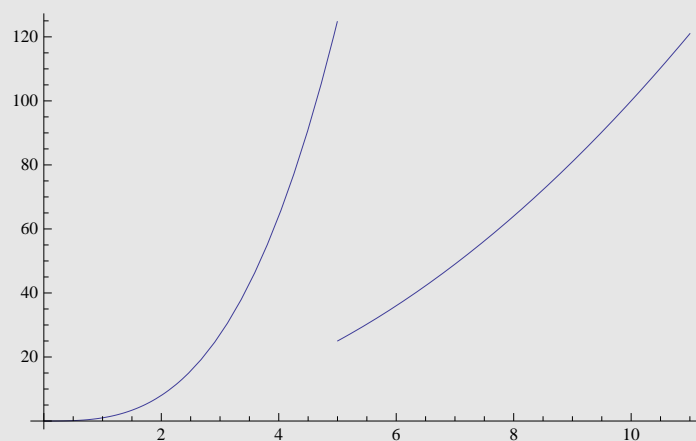
```
f[x_? (# > 5 &)] := x2
```

```
f[x_? (# ≤ 5 &)] := x3
```

```
Plot[f[x], {x, 0, 11}]
```



```
Plot[f[x], {x, 0, 11}, Exclusions → {5}]
```



■ Functions that take Functions as arguments

In functional programming a very important role is played by functions that take functions as arguments. The most important of these are Map and Apply:

```
Clear[f]
```

Map[f , { a , b , c , d }]

$\{f(a), f(b), f(c), f(d)\}$

Map[f , $x^2 y^3$]

$f(x^2) f(y^3)$

Map[f , $x^2 y^3$, {1}]

$f(x^2) f(y^3)$

Map[f , $x^2 y^3$, {2}]

$f(x)^{f(2)} f(y)^{f(3)}$

Apply[f , $g[x, y]$]

$f(x, y)$

Apply[Plus, $x * y$]

$x + y$

Apply[Times, $x + y$]

$x y$

Apply[Times, Unevaluated[2 + 3]]

6

Apply[f , $g[h[x], k[y]]$, {1}]

$g(f(x), f(y))$

Short notation:

Map[f , expr]

$f \text{ /@ } \text{expr}$

Apply[f , expr]

$f \text{ @@ } \text{expr}$

■ Attributes and Listability

The behavior of *Mathematica* functions and global rules is affected by so called Attributes. Each built-in function has some attributes, for example

Attributes[Sin]

{Listable, NumericFunction, Protected}

Attributes[Plus]

{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}

The most important attribute of functions is the attribute Listable. Let's explain briefly what it does.

Clear[f]

ls = Range[10]

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Map[f, ls]

{f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8), f(9), f(10)}

If we give f the Attribute Listable we will not need to use Map.

SetAttributes[f, Listable]

f[ls]

{f(1), f(2), f(3), f(4), f(5), f(6), f(7), f(8), f(9), f(10)}

In addition

f[{a, b}, {c, d}]

{f(a, c), f(b, d)}

f[a, {b, c}]

{f(a, b), f(a, c)}

The attribute Listable of Plus is the reason for the following behavior:

{1, 2, 3} + {4, 5, 6}

{5, 7, 9}

```
1 + {2, 3, 4, 5}
```

```
{3, 4, 5, 6}
```

The attributes Orderless, Flat and OneIdentity are interesting, but complicated. Let's see an illustration

```
ClearAll[f]
```

```
f[a, b] /. f[b, x_] -> g
```

```
f(a, b)
```

```
SetAttributes[f, Orderless]
```

```
f[a, b] /. f[b, x_] -> g
```

```
g
```

```
f[a, b, c] /. f[a, f[b, c]] -> g
```

```
f(a, b, c)
```

```
SetAttributes[f, Flat]
```

```
f[a, b, c] /. f[a, f[b, c]] -> g
```

```
g
```

```
ClearAll[f]
```

Another group of important attributes are HoldFirst, HoldAll, HoldRest

```
Attributes[Set]
```

```
{HoldFirst, Protected, SequenceHold}
```

```
Attributes[SetDelayed]
```

```
{HoldAll, Protected, SequenceHold}
```

```
ClearAll[f]
```

```
SetAttributes[f, HoldFirst]
```

```
f[x_, y_] := (x = y^2)
```

```
x = 3;
```

```
f[x, 2];
```

```
x
```

```
4
```

▣ Links

<http://reference.wolfram.com/mathematica/tutorial/PureFunctions.html>

<http://reference.wolfram.com/mathematica/tutorial/ApplyingFunctionsToListsAndOtherExpressions.html>

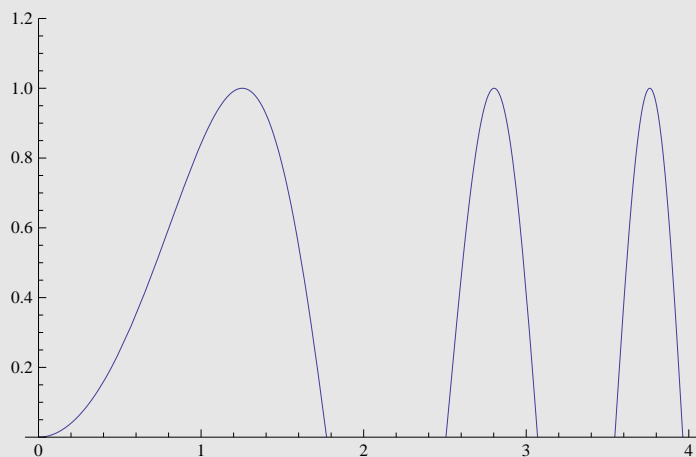
<http://reference.wolfram.com/mathematica/tutorial/Attributes.html>

<http://reference.wolfram.com/mathematica/tutorial/SelectingPartsOfExpressionsWithFunctions.html>

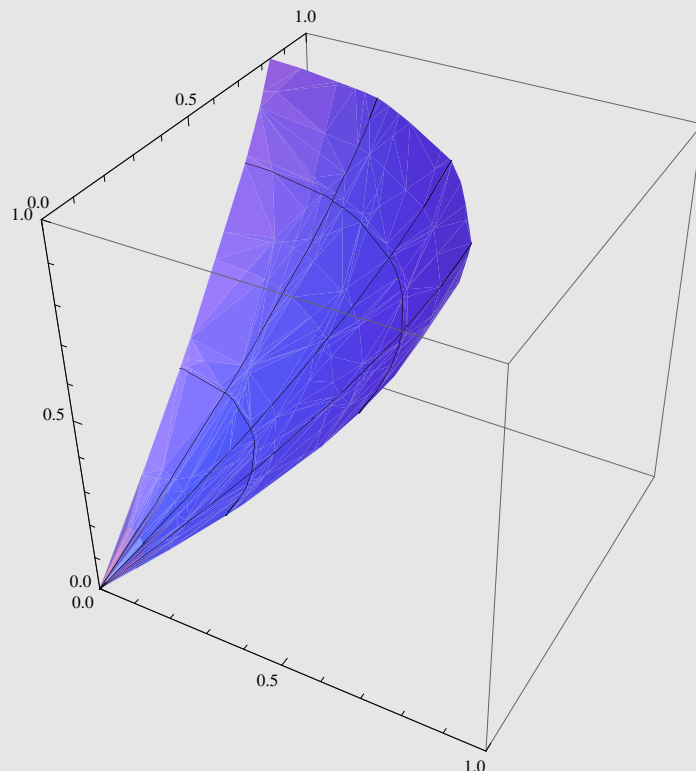
3. An Overview of Graphics

One can use Mathematica to make 2 D and 3 D graphics. It is perhaps the most straightforward and, at the same time, the most complicated section. It is straightforward since all the commands and properties one can find in the Help Browser. However, the complicated part is to find the property one needs among hundreds of similar ones. Below there are only examples of some of the most commonly used graphic features. Moreover, the older versions of *Mathematica* might have different names and commands. The following pictures are drawn in *Mathematica 7*. The pictures are mostly self-explanatory and for the usage of unknown functions the reader is referred to the documentation center.

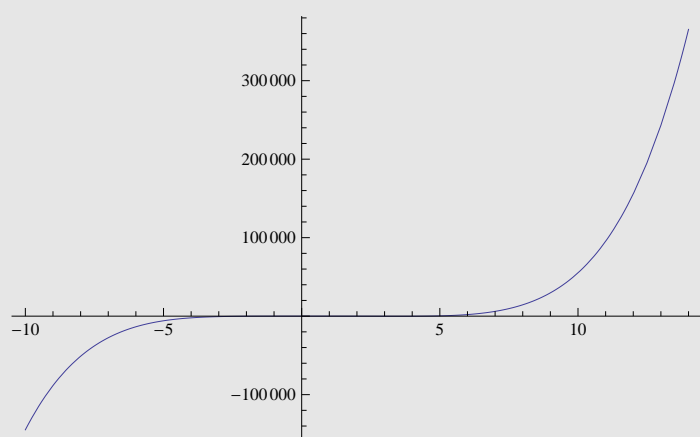
```
Plot[Sin[x2], {x, 0, 4}, PlotRange -> {0, 1.2}]
```



```
ParametricPlot3D[{Sin[z] Sin[t], Sin[z] Cos[t], z},  
{z, -π, π}, {t, 0, 2 π}, PlotRange -> {{0, 1}, {0, 1}, {0, 1}}]
```

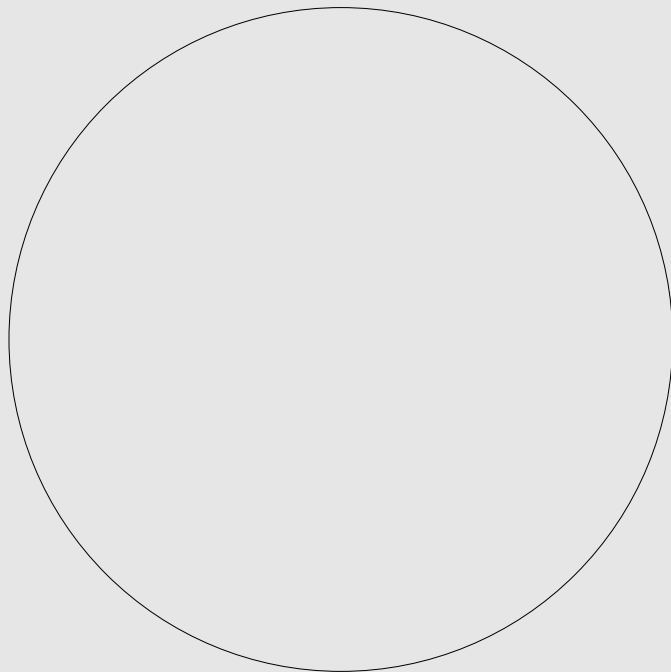


```
Plot[x5 - 4.5 x4 + 2.1 x2 - 7, {x, -10, 14}, PlotRange -> All]
```



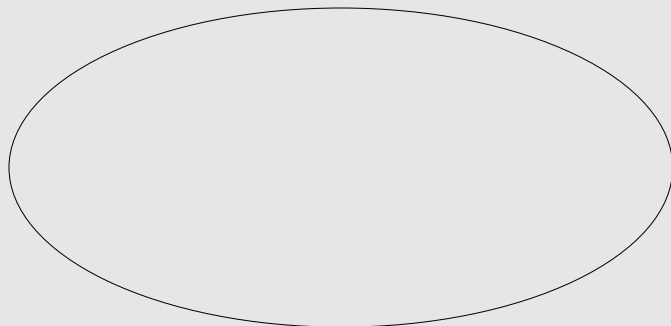
```
Graphics[Circle[{0, 0}, 1], PlotLabel -> "circle"]
```

circle

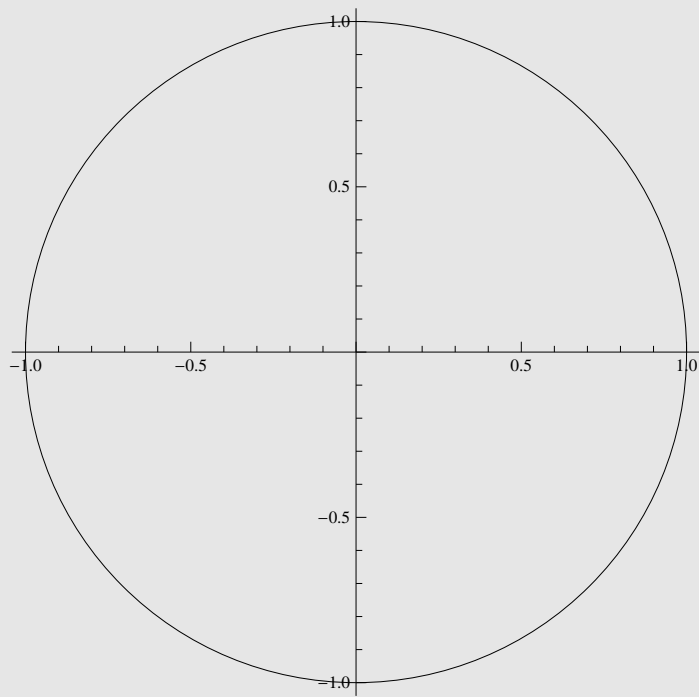


```
Graphics[Circle[{0, 0}, 1], AspectRatio -> 1 / 2, PlotLabel -> "ellipse"]
```

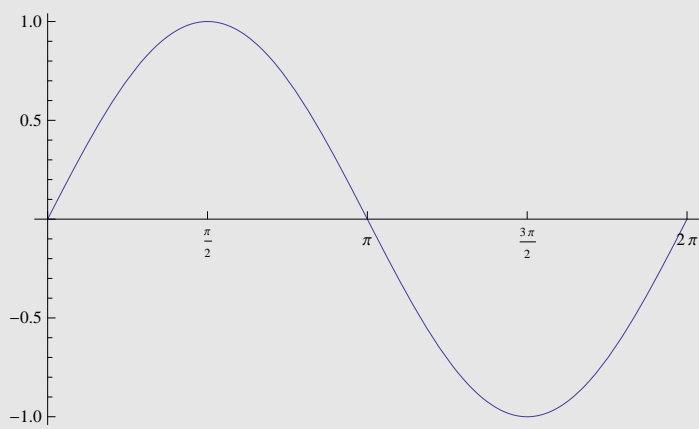
ellipse



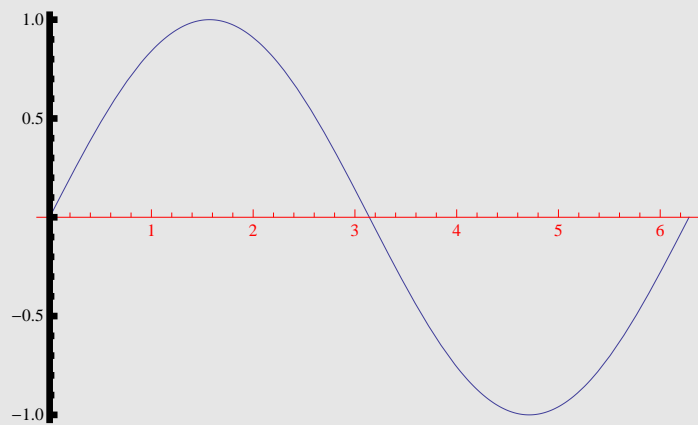
```
Graphics[Circle[{0, 0}, 1], Axes → Automatic]
```



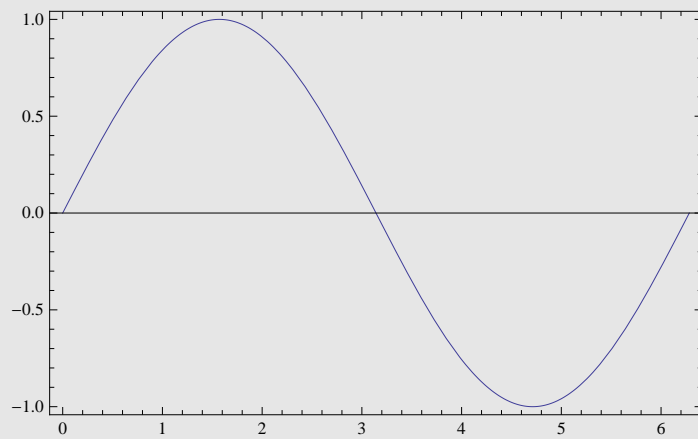
```
Plot[Sin[x], {x, 0, 2 π}, Ticks → {{0,  $\frac{\pi}{2}$ ,  $\pi$ ,  $\frac{3\pi}{2}$ , 2 π}, Automatic}]
```



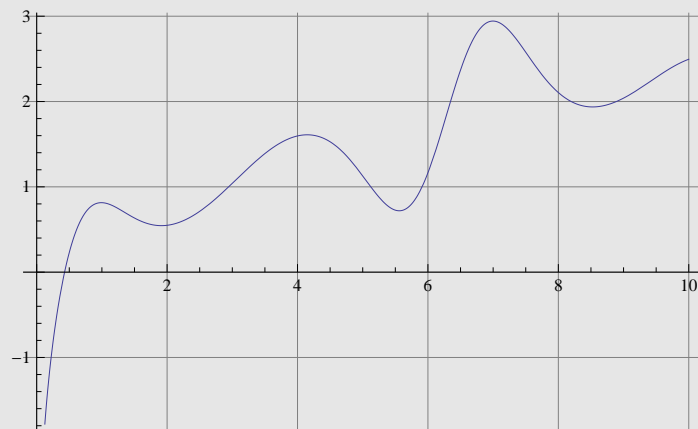
```
Plot[Sin[x], {x, 0, 2  $\pi$ }, AxesStyle -> {RGBColor[1, 0, 0], Thickness[0.01]}]
```



```
Plot[Sin[x], {x, 0, 2  $\pi$ }, Frame -> True]
```



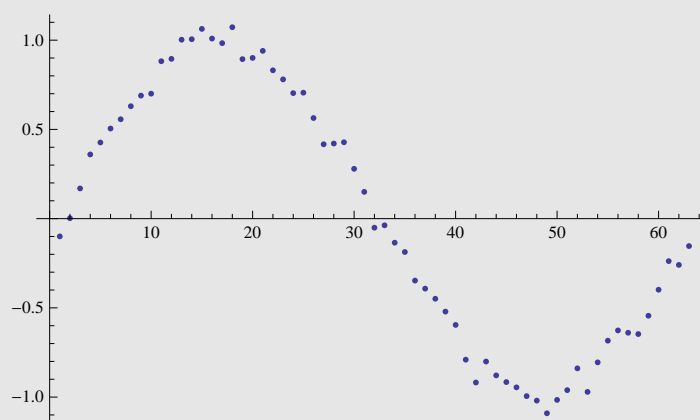
```
Plot[Log[x] + Sin[x +  $\sqrt{2}$  Sin[x]], {x, 0, 10}, GridLines -> Automatic]
```



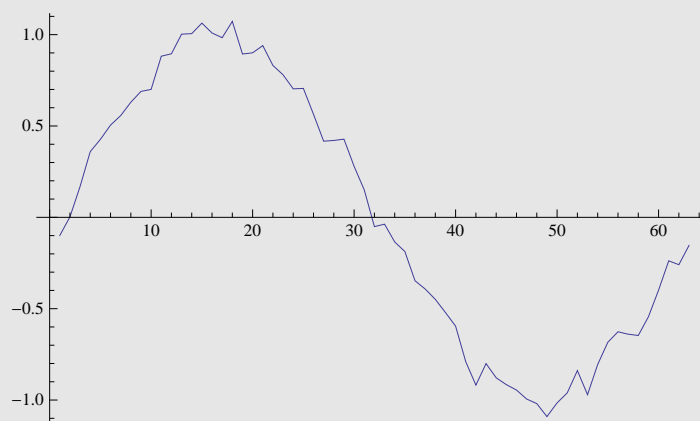
```
data = Table[Sin[x] + Random[Real, {-0.1, 0.1}], {x, 0, 2  $\pi$ , 0.1}]
```

```
{-0.0994549, 0.0025176, 0.169147, 0.35959, 0.426721, 0.504849, 0.5569,  
0.629977, 0.689323, 0.700252, 0.881904, 0.895217, 1.0026, 1.00545, 1.06306,  
1.009, 0.983381, 1.07258, 0.893839, 0.900578, 0.940057, 0.831201, 0.780229,  
0.703249, 0.705678, 0.563779, 0.416756, 0.420854, 0.4279, 0.279133,  
0.150118, -0.0507052, -0.0374291, -0.134788, -0.186977, -0.347078,  
-0.392135, -0.448773, -0.520907, -0.595565, -0.790225, -0.918132,  
-0.800616, -0.878242, -0.915784, -0.945376, -0.994464, -1.01954, -1.09056,  
-1.01561, -0.960952, -0.838909, -0.970764, -0.805304, -0.68379, -0.626348,  
-0.639521, -0.64668, -0.544192, -0.398389, -0.238055, -0.25922, -0.15363}
```

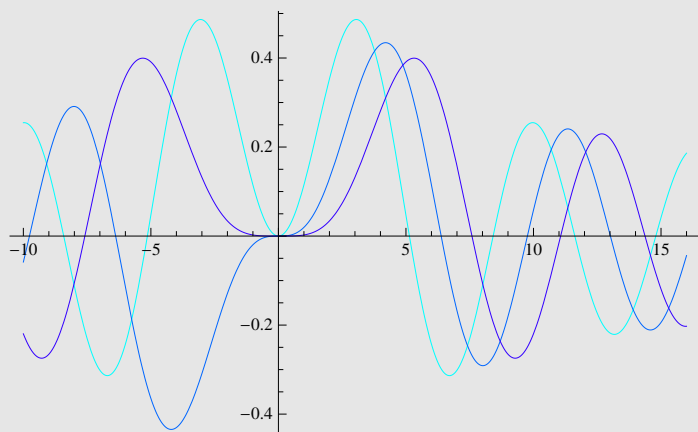
```
ListPlot[data]
```



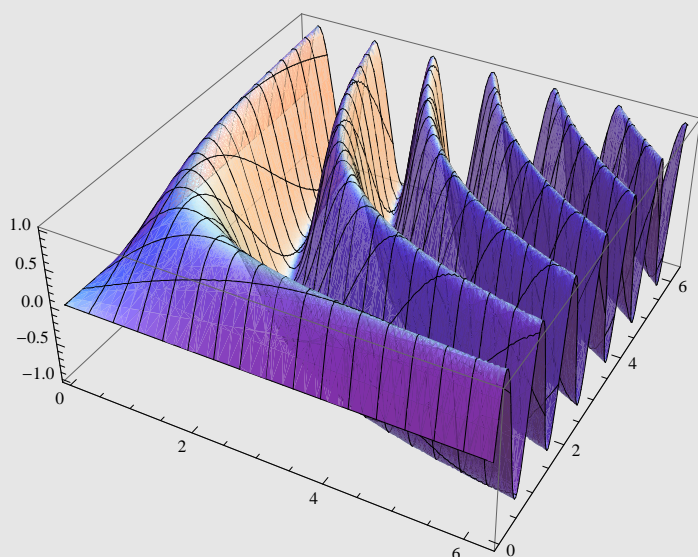
```
ListLinePlot[data]
```



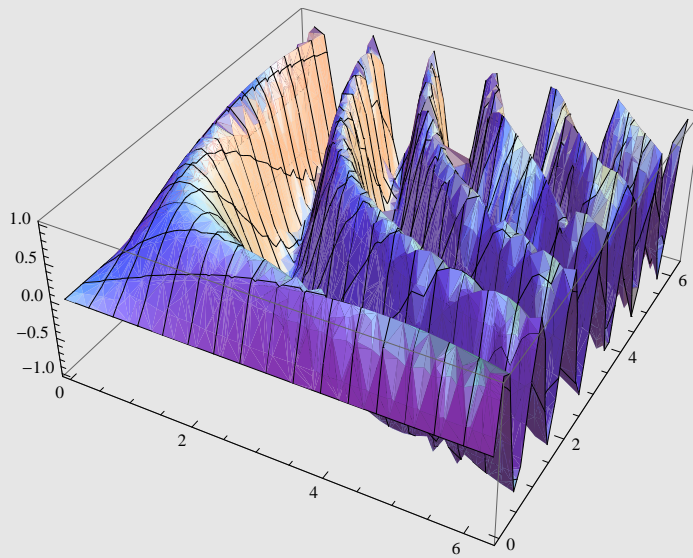
```
Plot[{BesselJ[2, z], BesselJ[3, z], BesselJ[4, z]},  
  {z, -10, 16}, PlotStyle -> {Hue[0.5], Hue[0.6], Hue[0.7]}]
```



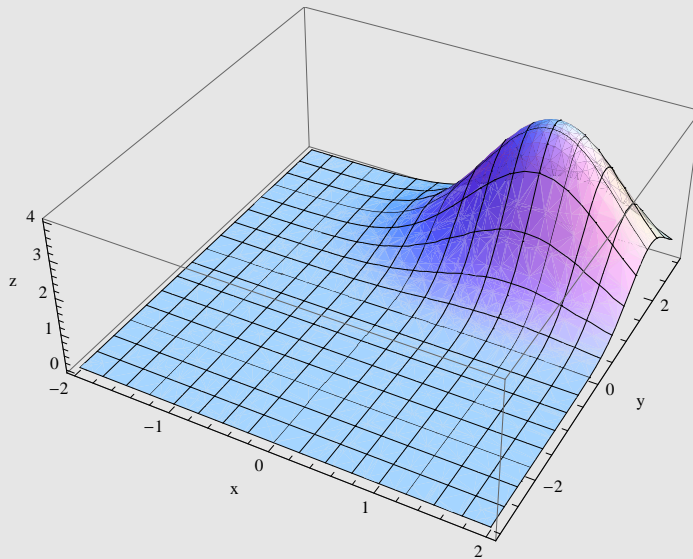
```
Plot3D[Sin[x y], {x, 0, 2 π}, {y, 0, 2 π}, PlotPoints -> 40]
```



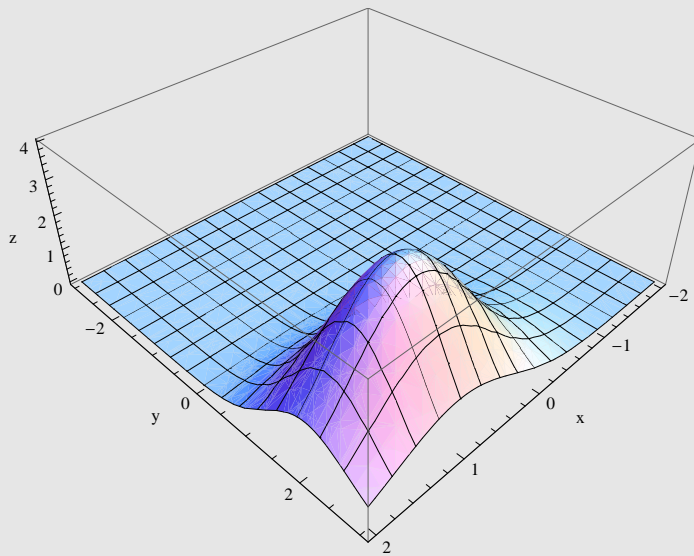
```
Plot3D[Sin[x y], {x, 0, 2  $\pi$ }, {y, 0, 2  $\pi$ }, PlotPoints  $\rightarrow$  10]
```



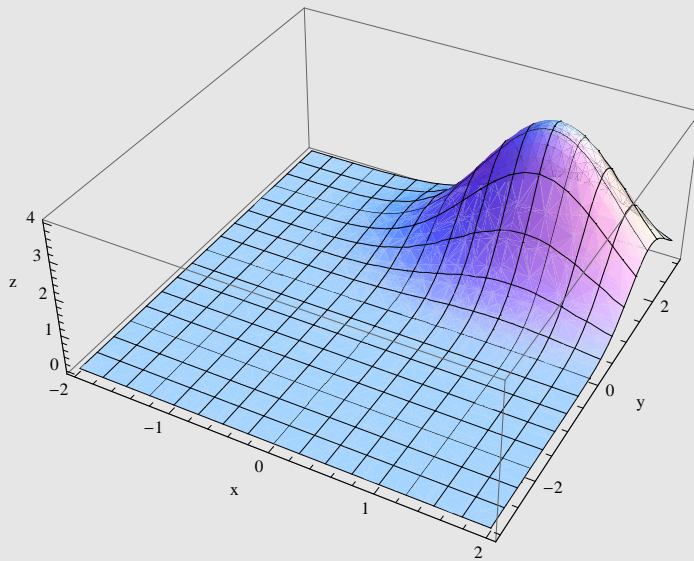
```
Plot3D[4 e-(x-1)2-(y-2)2, {x, -2, 2}, {y, -3, 3},  
PlotRange -> All, AxesLabel -> {"x", "y", "z"}]
```



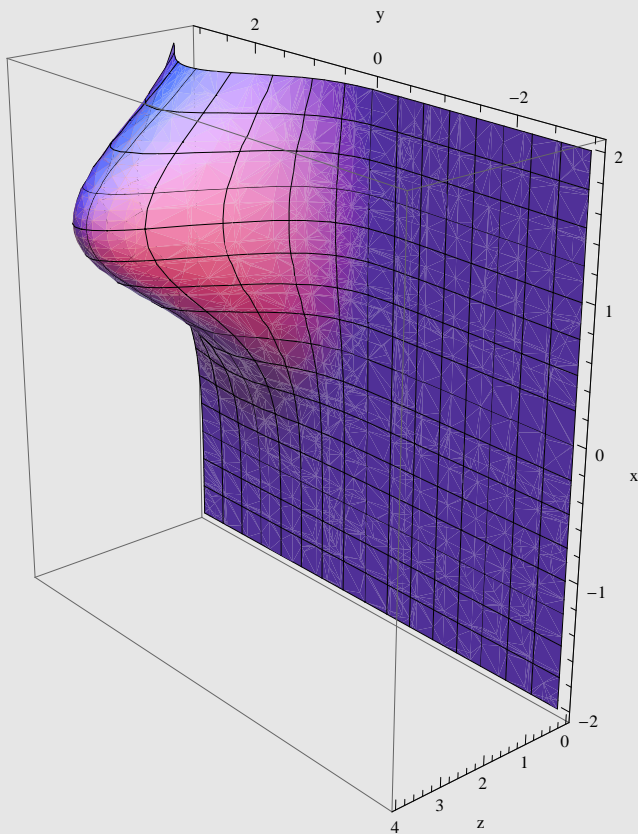
```
Show[%, ViewPoint -> {1.2, 1.2, 1.2}]
```



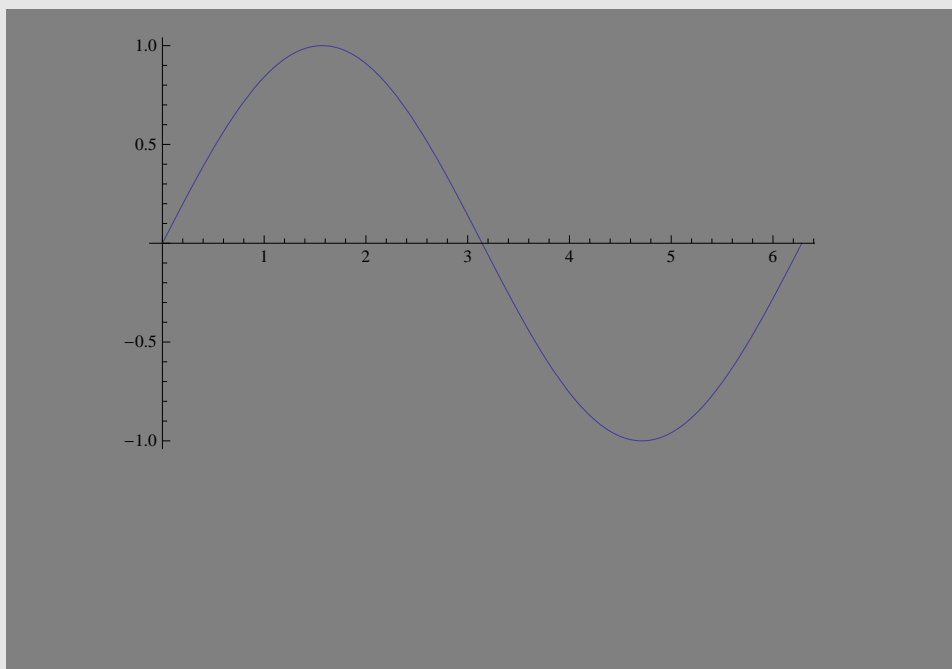
```
Plot3D[4 e-(x-1)2-(y-2)2, {x, -2, 2}, {y, -3, 3},  
PlotRange -> All, AxesLabel -> {"x", "y", "z"}]
```




```
Show[%, ViewVertical -> {1, 0, 0}]
```

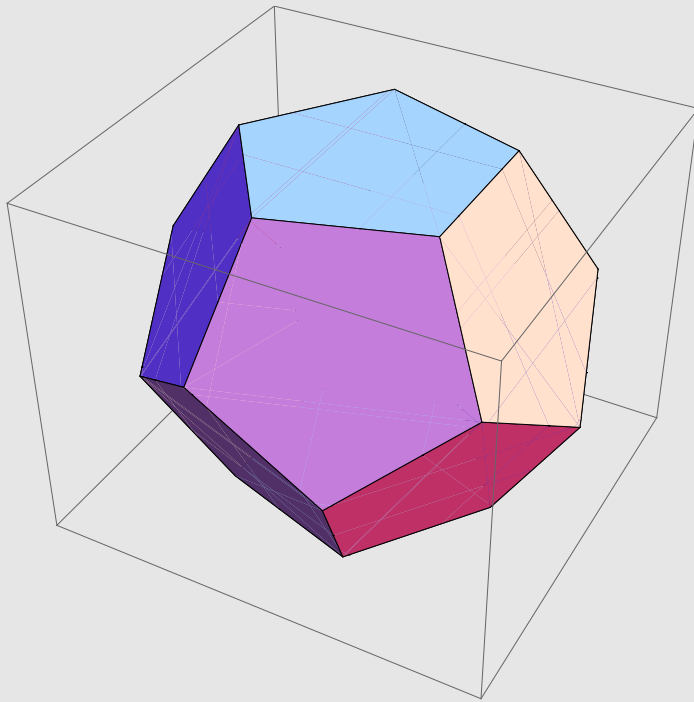


```
Plot[Sin[x], {x, 0, 2  $\pi$ }, Background -> GrayLevel[0.5]]
```

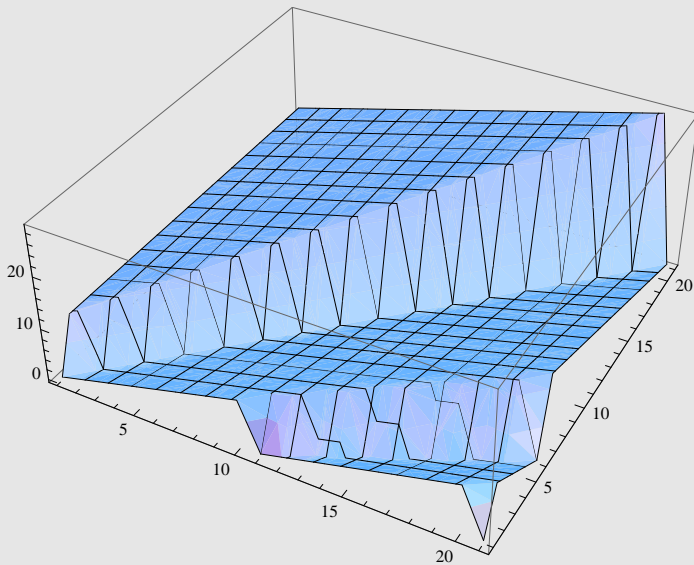


```
Quit[]
```

```
PolyhedronData["Dodecahedron"]
```



```
ListPlot3D[Table[Mod[y, x], {x, 10, 30}, {y, 10, 30}]]
```



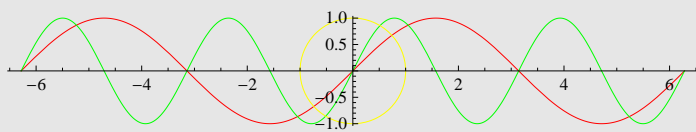
The function `Show` is used to combine several graphics together.

```
plot1 = Plot[Sin[x], {x, -2 Pi, 2 Pi}, PlotStyle -> Red];
```

```
plot2 = Plot[Sin[2 x], {x, -2 Pi, 2 Pi}, PlotStyle -> Green];
```

```
plot3 = Graphics[{Yellow, Circle[{0, 0}, 1]}];
```

```
Show[plot1, plot2, plot3, AspectRatio -> Automatic]
```



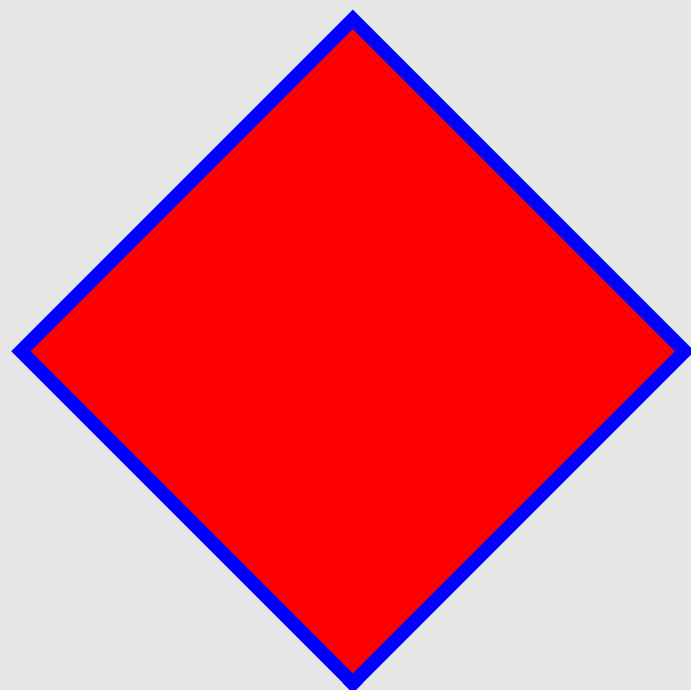
Graphics[primitives, options] represents a two-dimensional graphical image (circle, disc, point, line, polygon, ...).

```
vertices = {{0, -1}, {1, 0}, {0, 1}, {-1, 0}, {0, -1}};
```

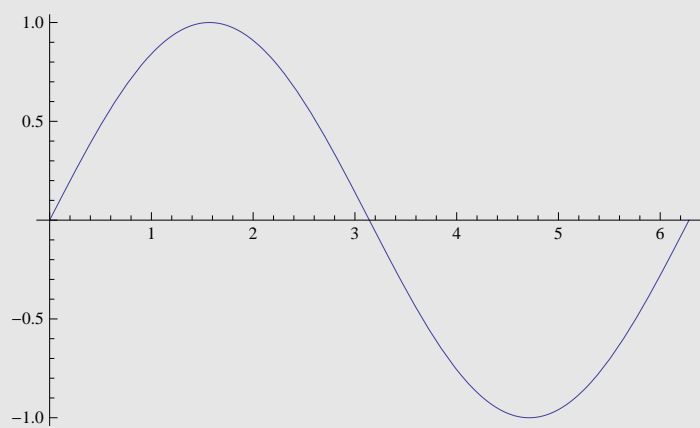
```
p = Graphics[{RGBColor[1, 0, 0], Polygon[vertices]}];
```

```
l = Graphics[{Thickness[.02], RGBColor[0, 0, 1], Line[vertices]}];
```

```
Show[p, l]
```

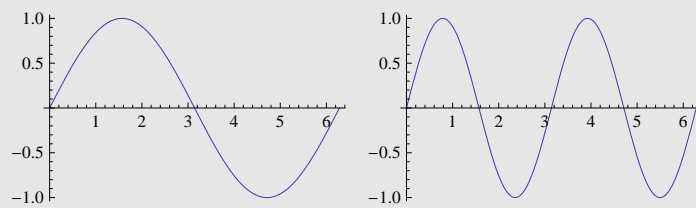


```
p1 = Plot[Sin[x], {x, 0, 2 π}]
```

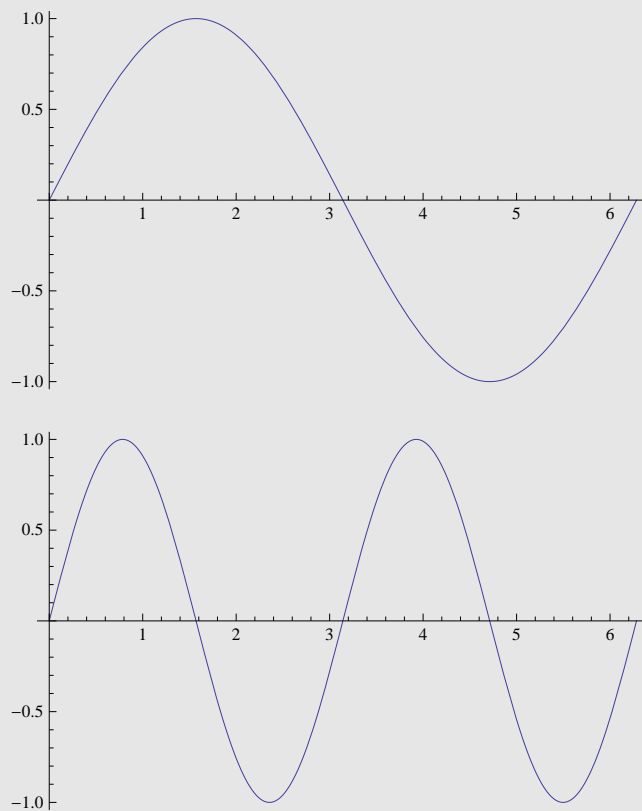


```
p2 = Plot[Sin[2 x], {x, 0, 2  $\pi$ ];
```

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



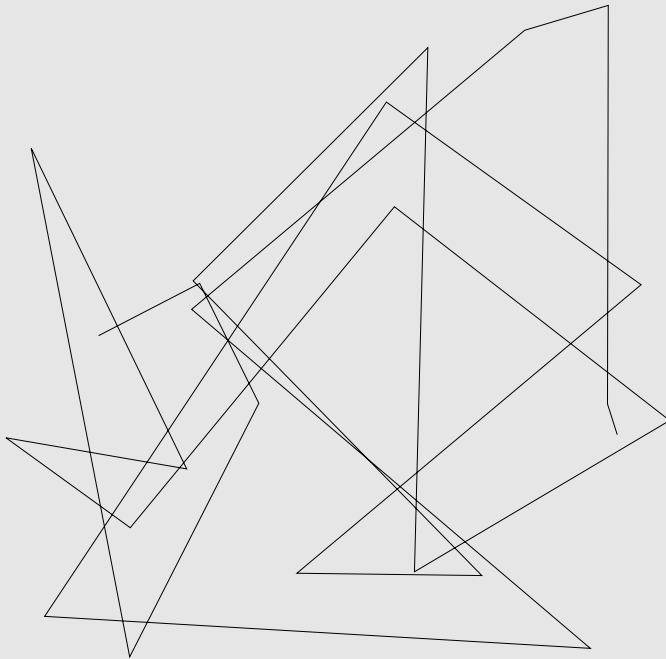
```
GraphicsGrid[{{p1}, {p2}}]
```



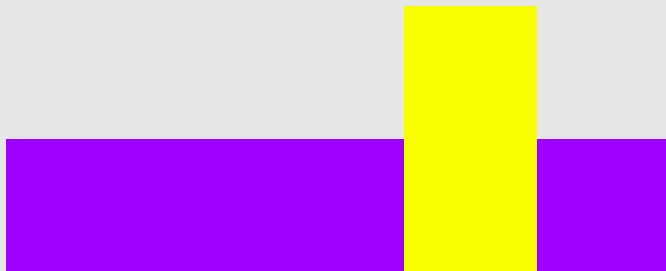
```
RandomReal[{0, 1}, {24, 2}]
```

```
{ {0.455485, 0.477362}, {0.70587, 0.75487}, {0.554577, 0.621808},  
  {0.801264, 0.342213}, {0.348274, 0.598254}, {0.378546, 0.520606},  
  {0.598194, 0.411677}, {0.856736, 0.864341}, {0.807134, 0.0230063},  
  {0.878315, 0.454539}, {0.874411, 0.0559939}, {0.32683, 0.456437},  
  {0.781892, 0.647471}, {0.975297, 0.293457}, {0.453405, 0.246344},  
  {0.452779, 0.784384}, {0.500323, 0.796357}, {0.218455, 0.890308},  
  {0.128262, 0.63954}, {0.927231, 0.22219}, {0.287996, 0.846428},  
  {0.823864, 0.134597}, {0.773817, 0.869142}, {0.264906, 0.357928} }
```

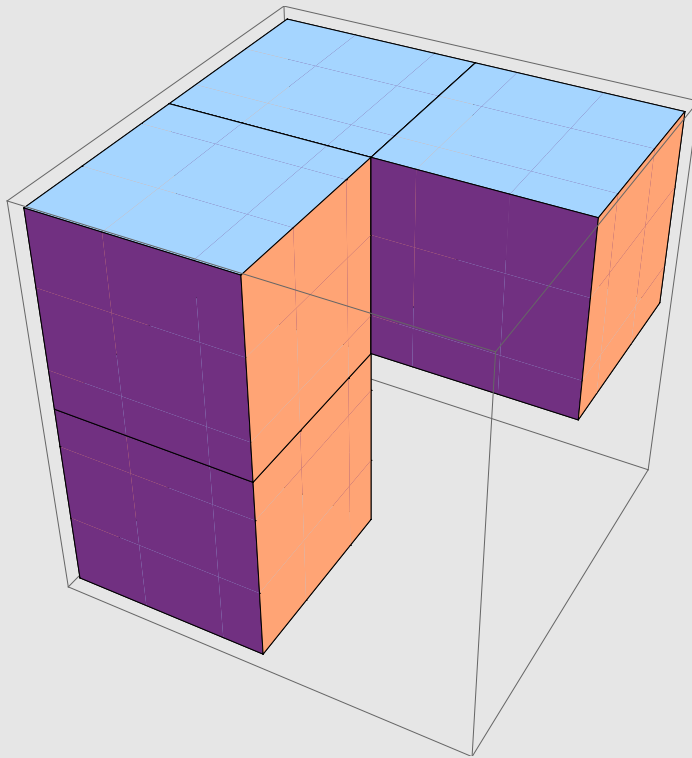
```
Graphics[Line[RandomReal[{0, 1}, {24, 2}]]]
```



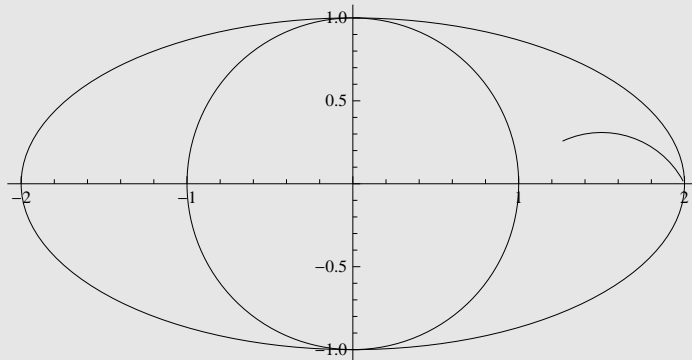
```
Graphics[{Hue[.77], Rectangle[{0, 0}, {5, 1}], Hue[.17], Rectangle[{3, 0}, {4, 2}]}]
```



```
Graphics3D[{Cuboid[{0, 0, 0}], Cuboid[{0, 0, 1}], Cuboid[{0, 1, 1}], Cuboid[{1, 1, 1}]}]
```



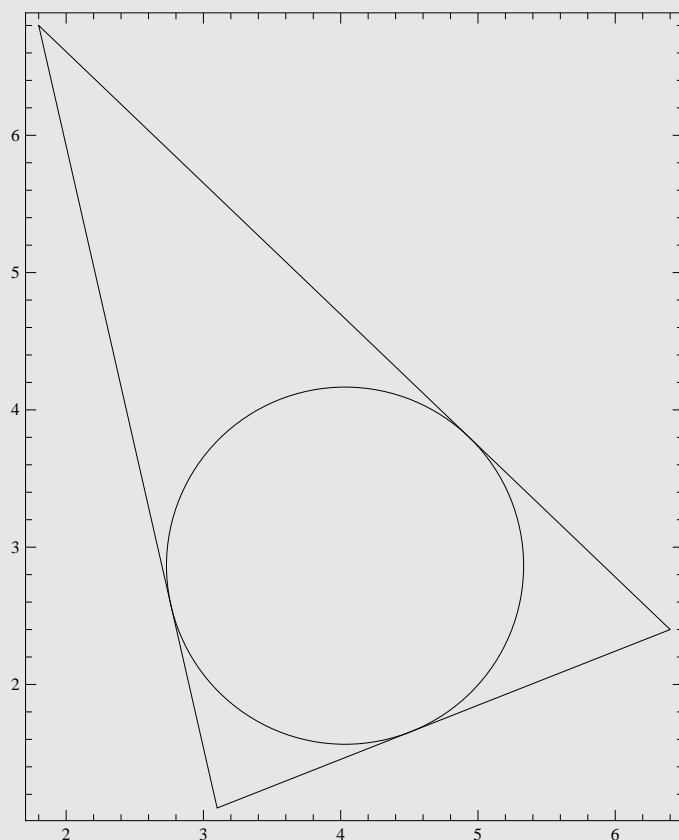
```
Graphics[{Circle[{0, 0}, 1], Circle[{0, 0}, {2, 1}], Circle[{3/2, -1/4}, sqrt(5)/4, {1/2, 2}]}],  
AspectRatio -> Automatic, Axes -> Automatic]
```



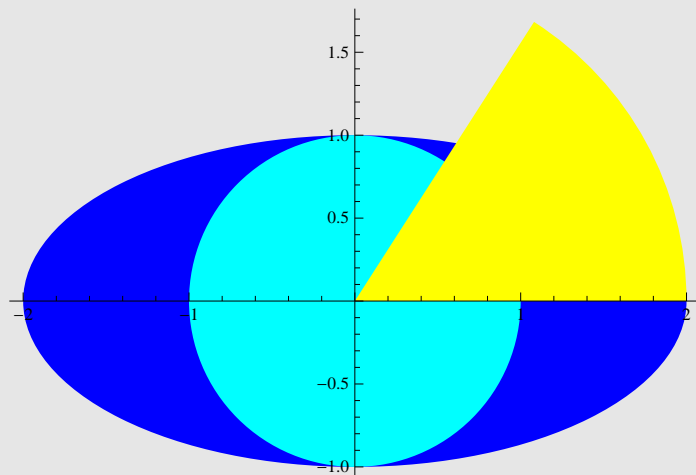
```
InscribedCircleData[pA : {_, _}, pB : {_, _}, pC : {_, _}] :=
Module[{AB, BC, AC, a, b, c, s, pP, pQ, AP, BQ, p, q, ps, qs, pqs, incenter, inradius},
  AB = pB - pA; BC = pC - pB; AC = pC - pA; a =  $\sqrt{BC \cdot BC}$ ; b =  $\sqrt{AC \cdot AC}$ ; c =  $\sqrt{AB \cdot AB}$ ;
  AP = pB + p BC - pA; BQ = pA + q AC - pB; ps = Solve[ $\frac{AP \cdot AB}{c} == \frac{AP \cdot AC}{b}$ , p][[1, 1]];
  qs = Solve[ $\frac{BQ \cdot BC}{a} == \frac{BQ \cdot (-AB)}{c}$ , q][[1, 1]]; pP = pB + p BC /. ps;
  pQ = pA + q AC /. qs; pqs = Solve[pA + p (pP - pA) == pB + q (pQ - pB), {p, q}][[1]];
  incenter = pA + p (pP - pA) /. pqs; s =  $\frac{1}{2} (a + b + c)$ ;
  inradius =  $\sqrt{\frac{(s - a) (s - b) (s - c)}{s}}$ ; {incenter, inradius}]
```

```
InscribedCircle[pA : {_, _}, pB : {_, _}, pC : {_, _}] := Graphics[
  {Line[{pA, pB, pC, pA}], Circle[Sequence @@ InscribedCircleData[pA, pB, pC]]},
  AspectRatio -> Automatic, PlotRange -> All, Frame -> True]
```

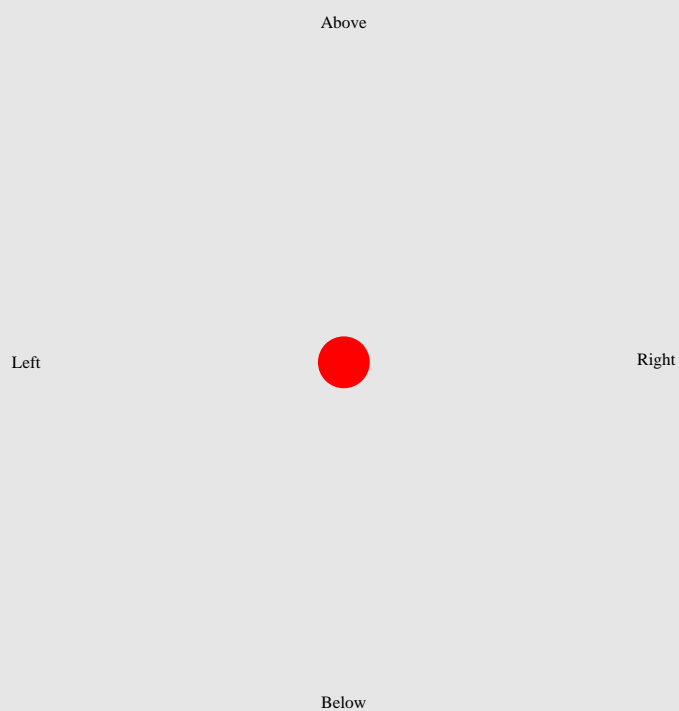
```
InscribedCircle[{1.8, 6.8}, {3.1, 1.1}, {6.4, 2.4}]
```



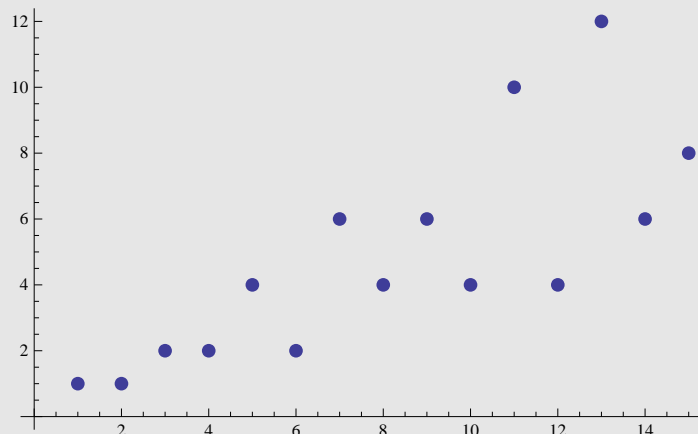
```
Graphics[{{RGBColor[0, 0, 1], Disk[{0, 0}, {2, 1}]},
  {RGBColor[0, 1, 1], Disk[{0, 0}, 1]}, {RGBColor[1, 1, 0], Disk[{0, 0}, 2, {0, 1}]}},
  AspectRatio -> Automatic, Axes -> Automatic]
```



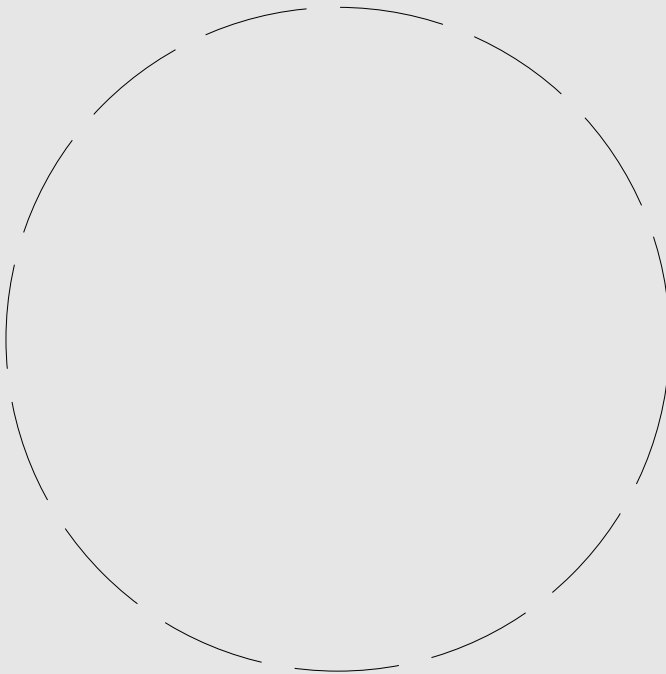
```
Graphics[{{Text["Left", {-1, 0}, {-1, 0}], Text["Right", {1, 0}, {1, 0}],
  Text["Above", {0, 1}, {0, -1}], Text["Below", {0, -1}, {0, 1}],
  {PointSize[.075], RGBColor[1, 0, 0], Point[{0, 0}]}}, PlotRange -> All]
```



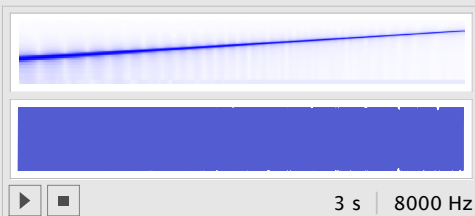

```
ListPlot[Table[{x, EulerPhi[x]}, {x, 15}], PlotStyle -> PointSize[0.02]]
```



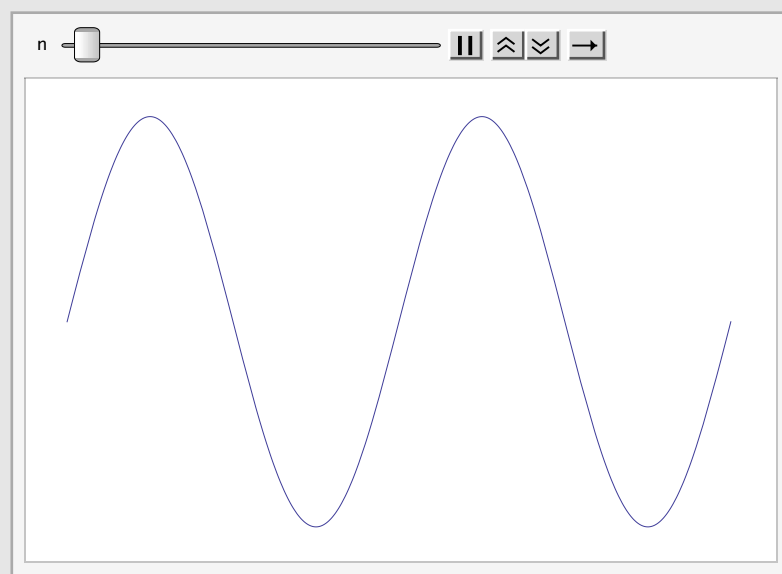
```
Graphics[{Dashing[{0.15, 0.05}], Circle[{0, 0}, 1]}, AspectRatio -> Automatic]
```



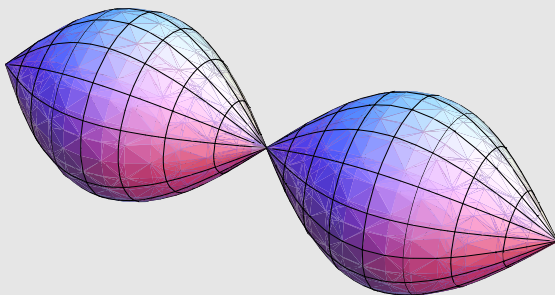
```
Play[Sin[2^t], {t, 11, 14}]
```



```
Animate[Plot[Sin[n x], {x, 0, 2 Pi}, Axes -> False], {n, 1, 3, 1}]
```

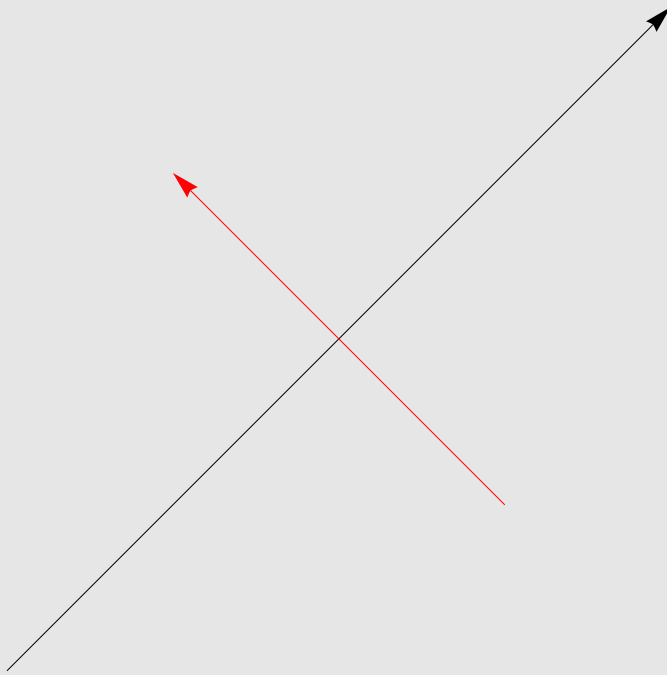


```
g = ParametricPlot3D[  
  {x, Cos[t] Sin[x], Sin[t] Sin[x]},  
  {x, -Pi, Pi}, {t, 0, 2Pi},  
  Axes -> False, Boxed -> False]
```

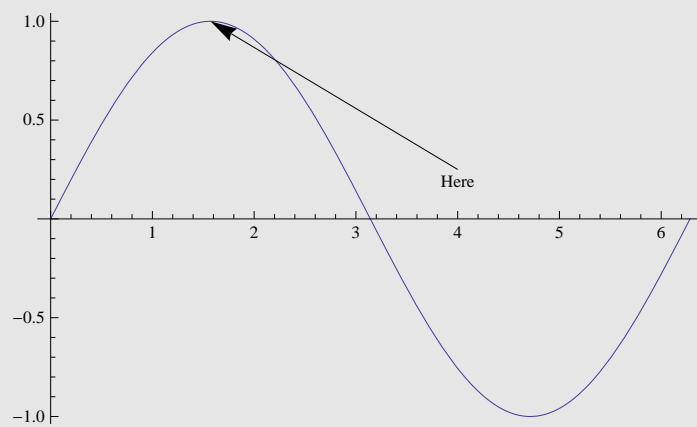


```
Quit[]
```

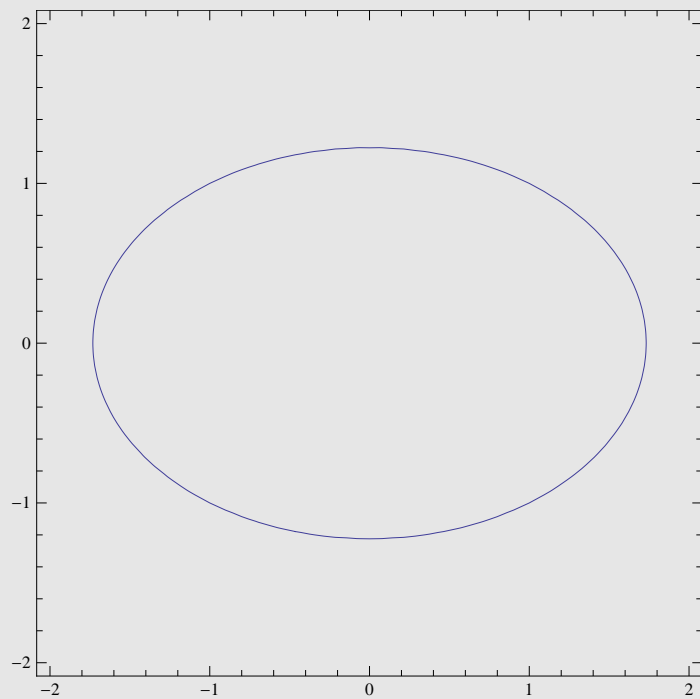
```
Graphics[{Arrow[{{0, 0},{1, 1}}],
  Hue[0], Arrow[{{.75, .25},{.25, .75}}]}]
```



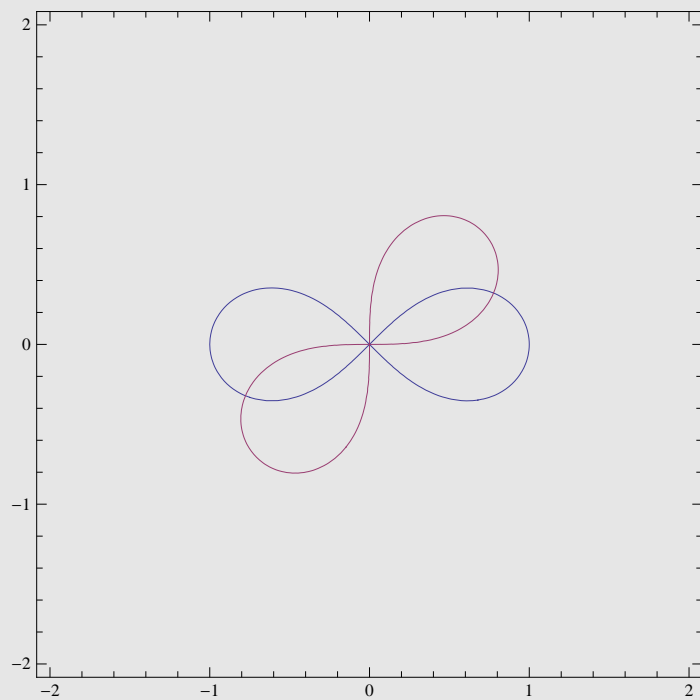
```
Plot[Sin[x], {x, 0, 2Pi},
  Epilog -> {Arrow[{{4, .25}, {Pi/2, 1}}],
  Text["Here", {4, .15}, {0, -1}]}]
```



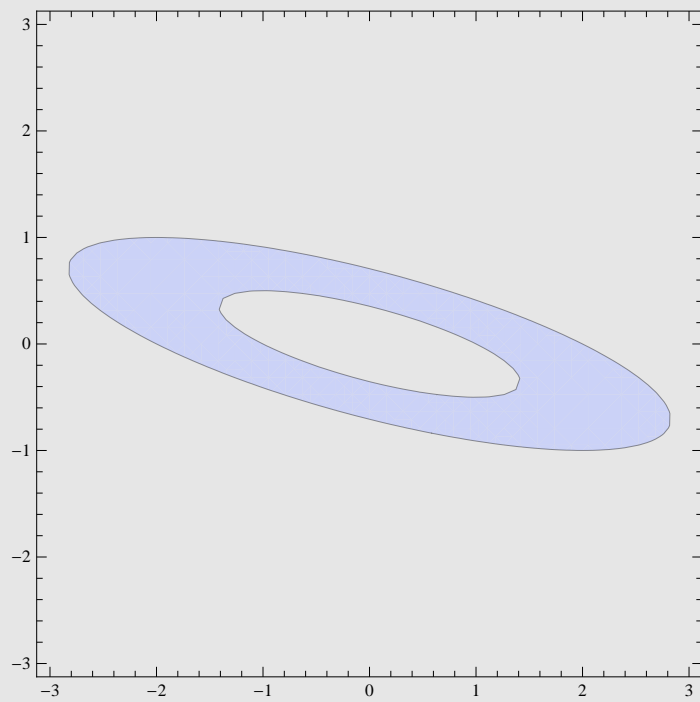
```
ContourPlot[x^2 + 2 y^2 == 3, {x, -2, 2}, {y, -2, 2}]
```



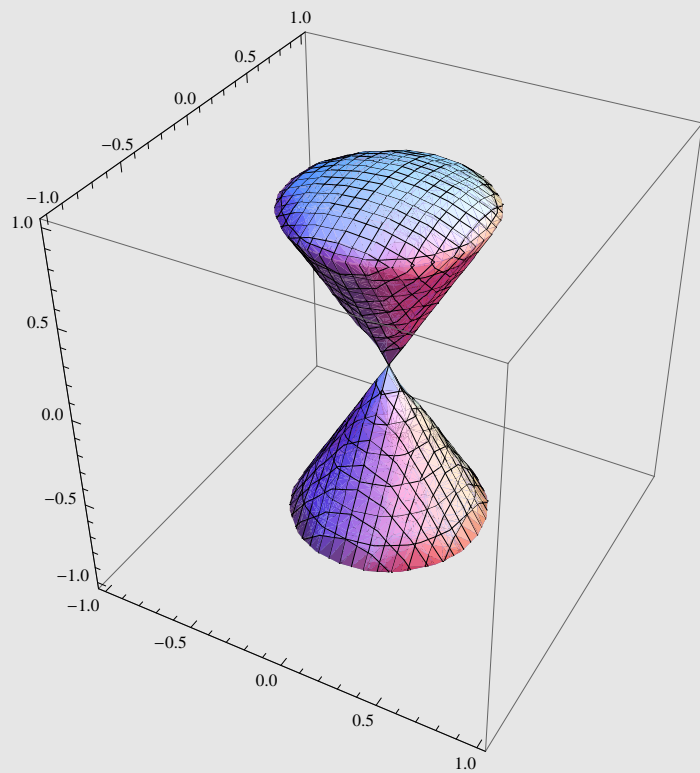
```
ContourPlot[{(x^2 + y^2)^2 == (x^2 - y^2),  
(x^2 + y^2)^2 == 2 x y}, {x, -2, 2}, {y, -2, 2}]
```



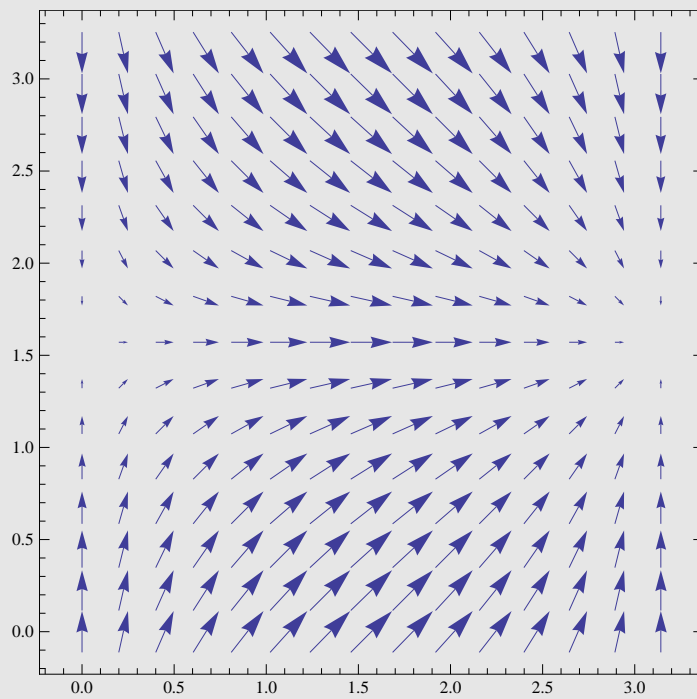
```
RegionPlot[ $1 \leq (x + 2y)^2 + 4y^2 \leq 4$ , {x, -3, 3}, {y, -3, 3}]
```



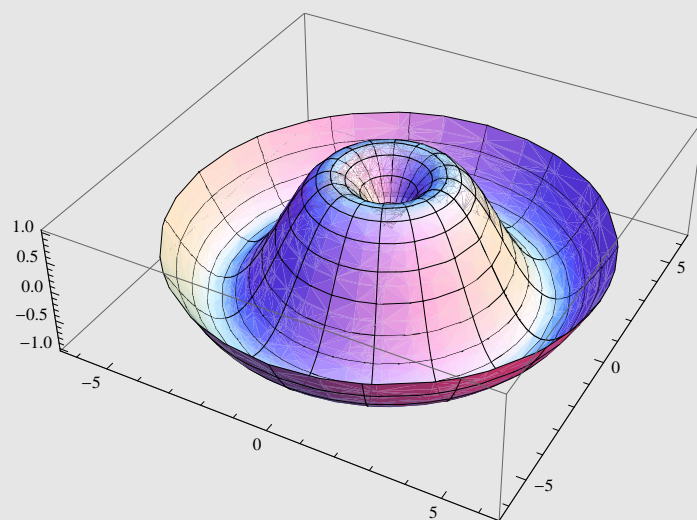
```
RegionPlot3D[ $x^2 + y^2 + z^2 \leq 1 \wedge 3x^2 + 3y^2 \leq z^2$ , {x, -1, 1}, {y, -1, 1}, {z, -1, 1}]
```



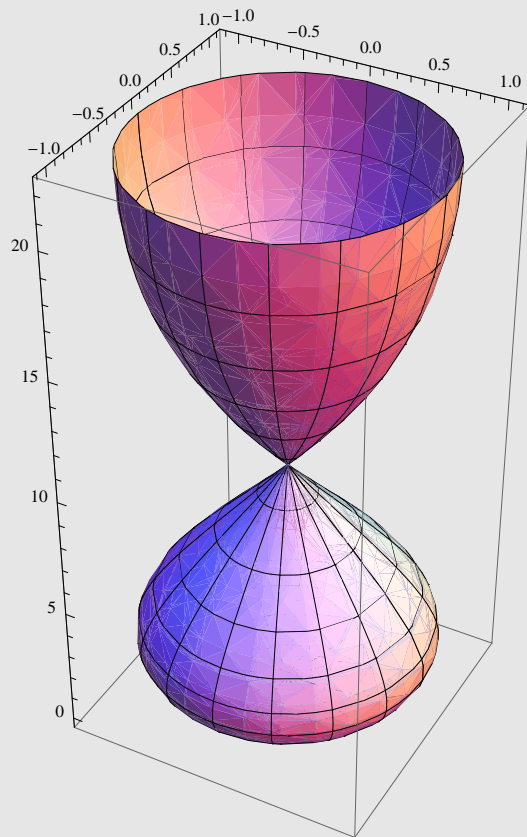
```
VectorPlot[{Sin[x], Cos[y]}, {x, 0, Pi}, {y, 0, Pi}]
```



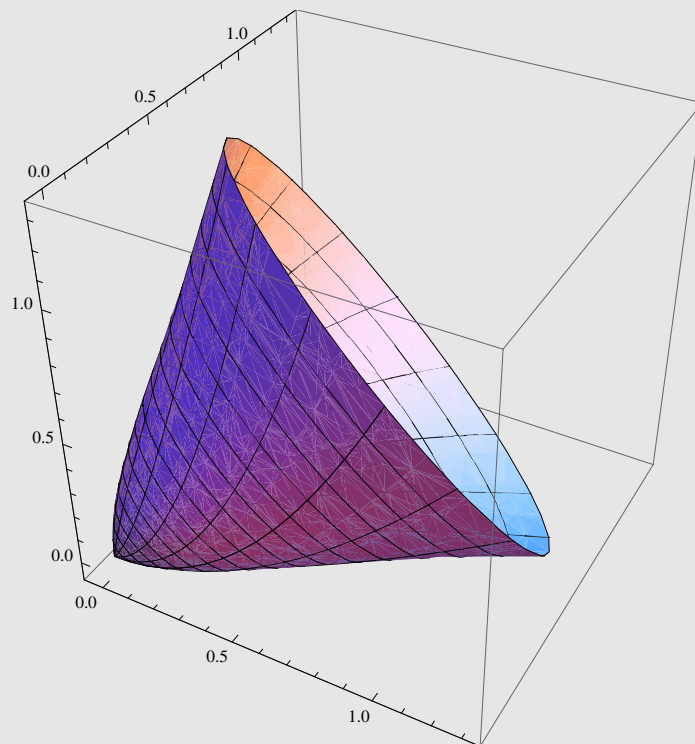
```
RevolutionPlot3D[  
  Sin[x], {x, 0, 2 Pi}]
```



```
RevolutionPlot3D[{1.1 Sin[u], u^2},  
  {u, 0, 3 Pi/2}, BoxRatios -> {1, 1, 2}]
```



```
RevolutionPlot3D[x^2, {x, 0, 1},  
  RevolutionAxis -> {1, 1, 1}]
```



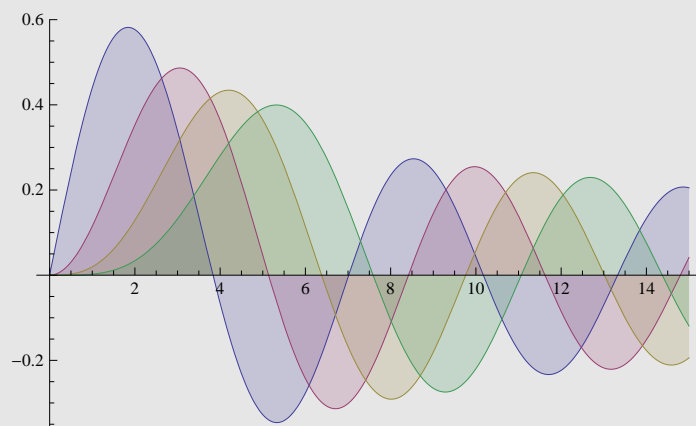
```
FinancialData["GE", "Price"]
```

20.595

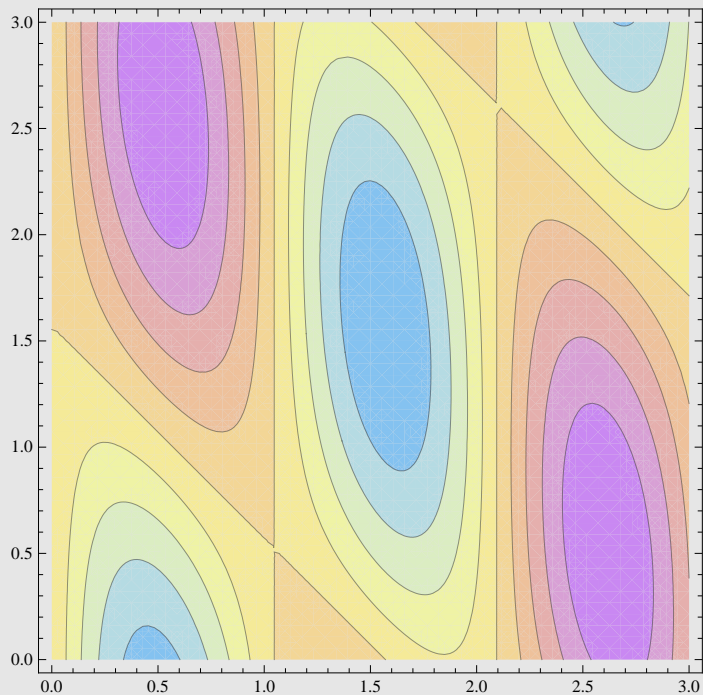
```
DateListLogPlot[FinancialData["^DJI", All], Joined → True, Filling → Bottom]
```



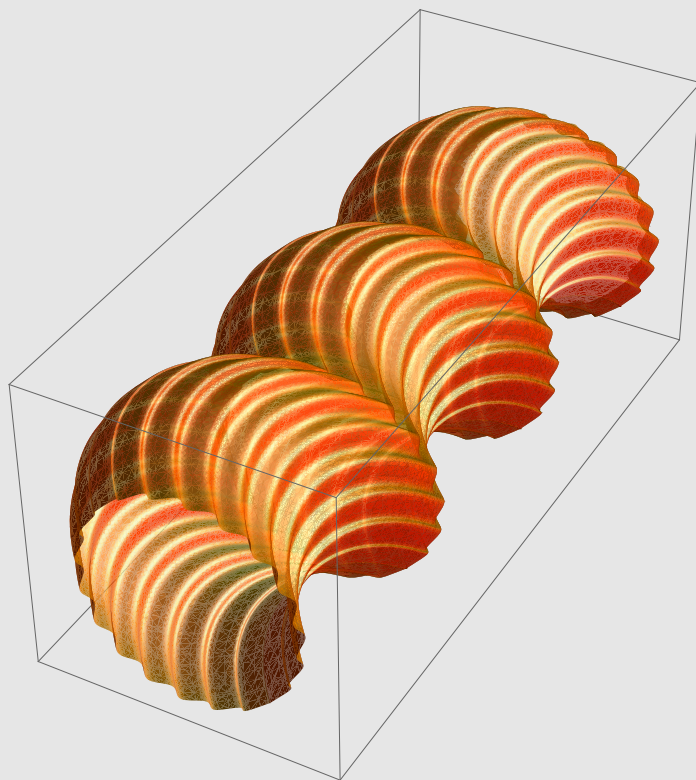
```
Plot[Table[BesselJ[n, x], {n, 4}], {x, 0, 15}, Filling → Axis, Evaluated -> True]
```




```
ContourPlot[Sin[3 x] Cos[x + y], {x, 0, 3}, {y, 0, 3},
  ContourLabels -> Automatic, ColorFunction -> "Pastel"]
```

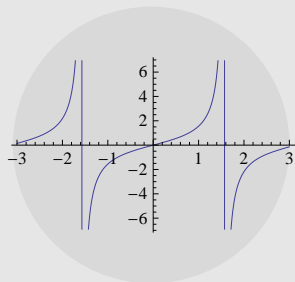


```
ParametricPlot3D[{Cos[v] + 0.3 Sin[3 u] + 0.04 Sin[20 v], u,
  Sin[v] + 0.3 Cos[3 u] + 0.04 Sin[20 v]}, {u, -π, π}, {v, -π, π}, PlotPoints -> 100,
  PlotStyle -> {Orange, Specularity[White, 10]}, Axes -> None, Mesh -> None]
```



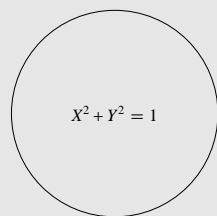
One can insert a plot into a disk:

```
Graphics[{LightGray, Disk[], Inset[Plot[Tan[x], {x, -3, 3}]]}]
```



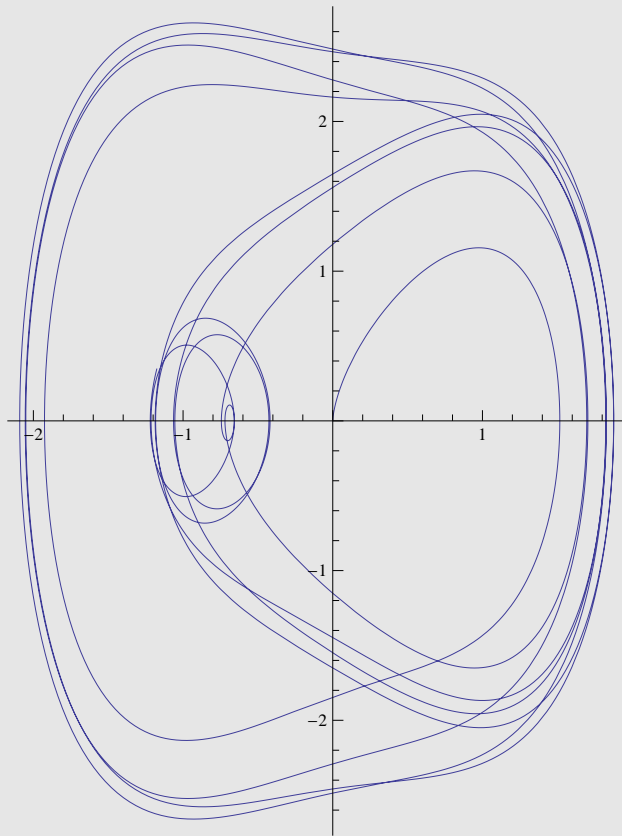
One can insert an expression in a graphic:

```
Graphics[{Circle[], Inset[X^2 + Y^2 == 1, {0, 0}]]}
```

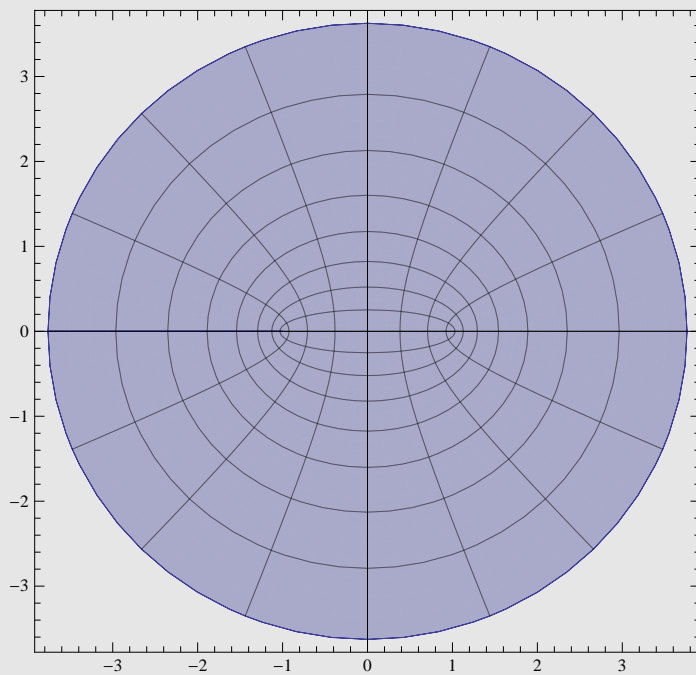


```
solution2 = NDSolve[{x''[t] + x[t]^3 == Sin[t], x[0] == x'[0] == 0}, x, {t, 0, 50}]
{{x -> InterpolatingFunction[{{0., 50.}}, <>]}}
```

```
ParametricPlot[{x[t], x'[t]} /. solution2, {t, 0, 50}]
```



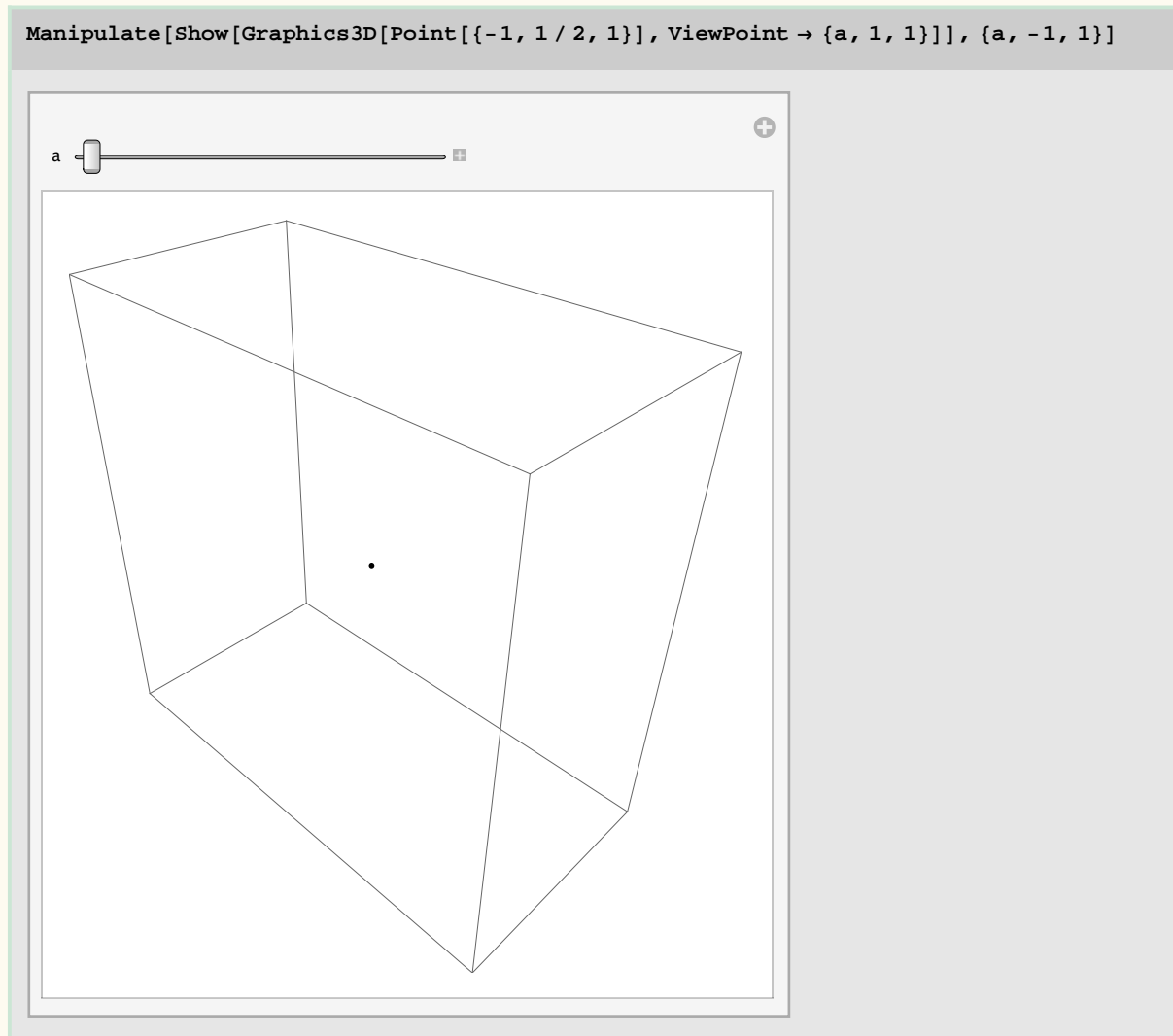
```
Block[{f = Cos[x + I y]},  
ParametricPlot[Evaluate[{Re[f], Im[f]}], {x, -Pi, Pi}, {y, -2, 2}]]
```



4. Dynamic Interactivity

Mathematica has several dynamic elements. It is very useful for visualization of the results if the problem contains some parameters and one wants to study it under the change of those parameters. You can study the following examples.

Example 1.



Example 2.

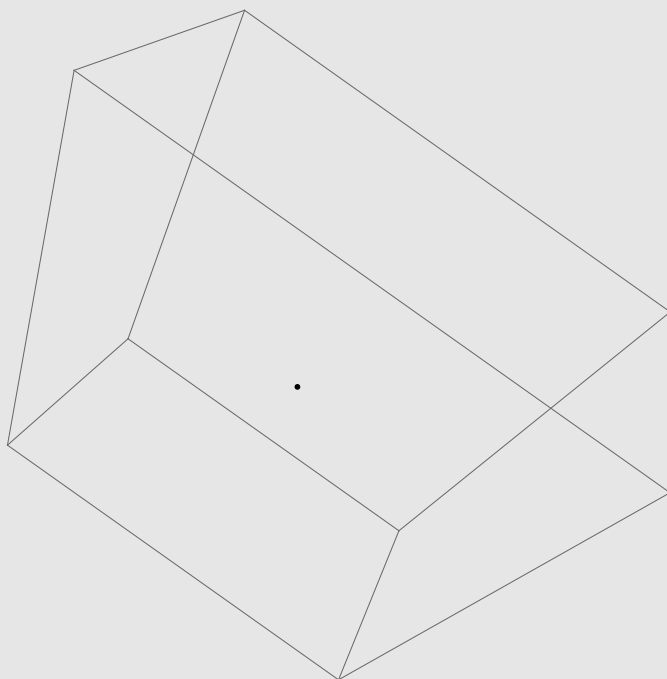
b
b

Dynamic[b]
0.

```
Slider[Dynamic[b]]
```



```
Show[Graphics3D[Point[{1, 1/2, 1}], ViewPoint -> {Dynamic[b], 1, 1}]]
```



Other examples.

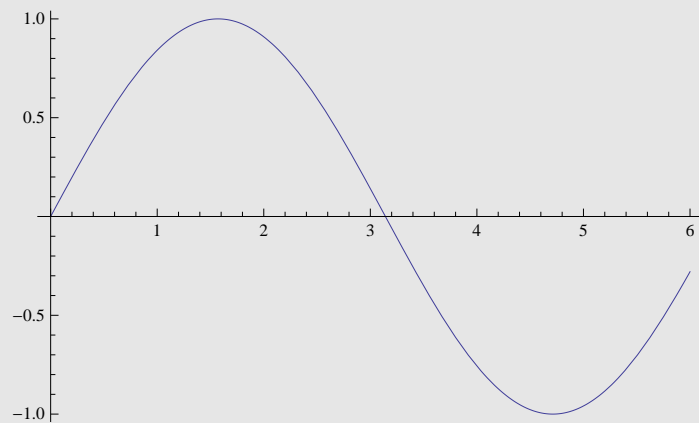
```
Dynamic[n]
```

0.

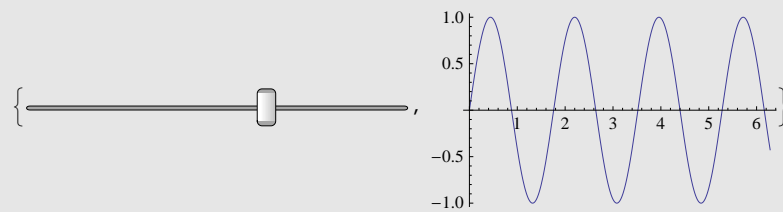
```
Slider[Dynamic[n]]
```



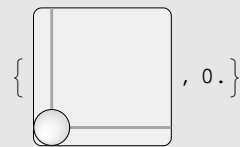
```
Dynamic[Plot[Sin[(n + 1) x], {x, 0, 6}]]
```



```
DynamicModule[{x}, {Slider[Dynamic[x], {1, 5}], Dynamic[Plot[Sin[x i], {i, 0, 2 Pi}]]}]
```



```
{Slider2D[Dynamic[x]], Dynamic[x]}
```



```
{InputField[Dynamic[x]], Dynamic[x]}
```

```
{0., 0.}
```

```
{0., 0.}
```

```
{Slider[Dynamic[1 - y, (y = 1 - #) &]], Dynamic[y]}
```

```
{Slider[Dynamic[1 - y, (y = 1 - #) &]], y}
```

```
{Slider[Dynamic[x]], Slider[Dynamic[1 - x, (x = 1 - #) &]]}
```

```
{Slider[Dynamic[x]], Slider[Dynamic[1 - x, (x = 1 - #) &]]}
```

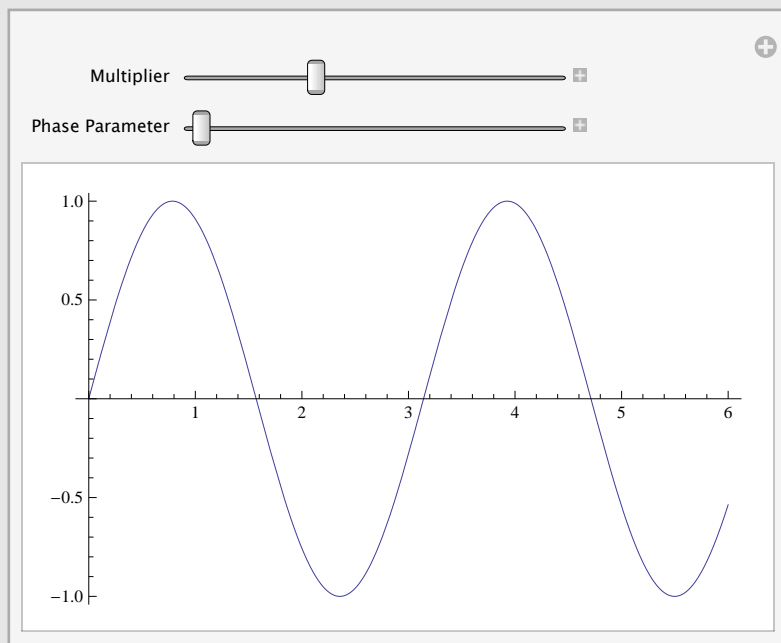
```
Clear[x, n]
```

```
Manipulate[Factor[x^n - 1], {n, 10, 100, 1}]
```



```
Clear[a, b, c, x, y]
```

```
Manipulate[Plot[Sin[a x + b], {x, 0, 6}],  
  {{a, 2, "Multiplier"}, 1, 4}, {{b, 0, "Phase Parameter"}, 0, 10}]
```



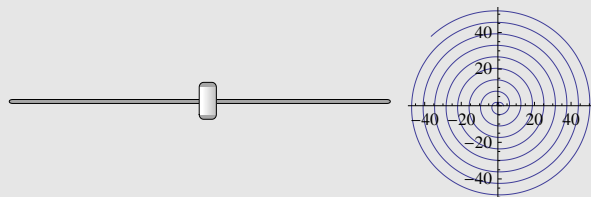
```
Grid[{{a, b, c}, {x, y, z}}]
```

```
a b c  
x y z
```

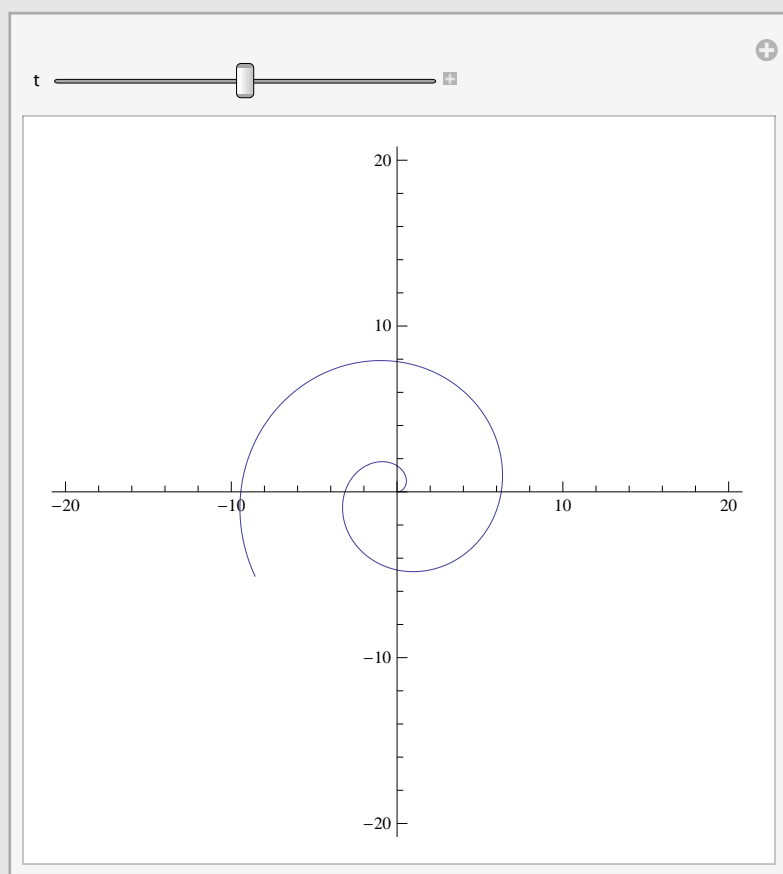
```
Grid[{{a, b, c}, {x, y^2, z^3}}, Frame -> All]
```

a	b	c
x	y ²	z ³

```
DynamicModule[{ $\theta = 0$ }, Grid[{{Slider[Dynamic[ $\theta$ ], {0, 100}],  
Dynamic@PolarPlot[t, {t, -Pi,  $\theta$ ], ImageSize -> Tiny}}]}
```



```
Manipulate[PolarPlot[ $\theta$ , { $\theta$ , 0, t}, PlotRange -> 20], {t, 1, 6 Pi}]
```



```
DynamicModule[{col = Green}, EventHandler[  
Style["text", FontColor -> Dynamic[col]], {"MouseClicked" -> (col = Red)}]]
```

text

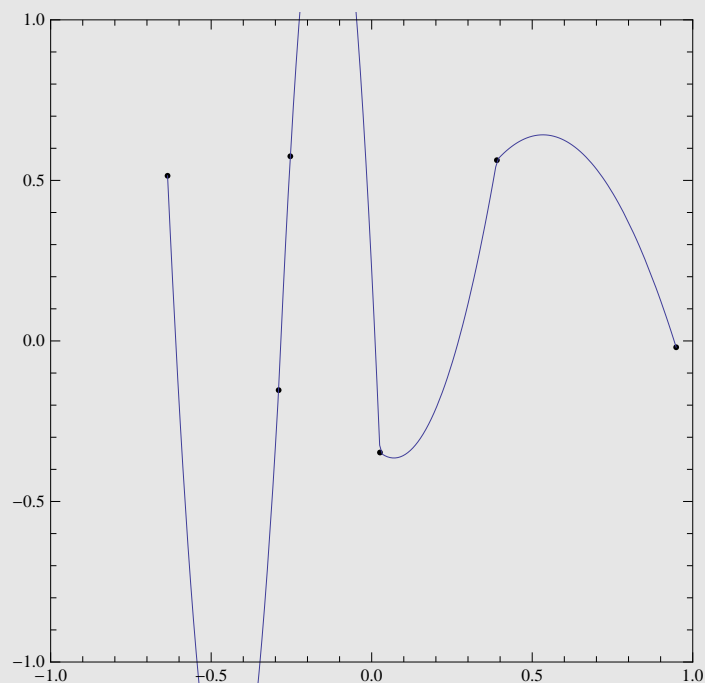
```
DynamicModule[{col = Green}, EventHandler[Style["text", FontColor -> Dynamic[col]],  
{"MouseClicked" -> (col = col /. {Red -> Green, Green -> Red})}]]
```

text

```
interpolationCurve[p_, n_] :=  
Module[{x, f = Interpolation[p, InterpolationOrder -> n]},  
First@Plot[Evaluate@f[x], {x, Min[p[[All, 1]]], Max[p[[All, 1]]]}];
```



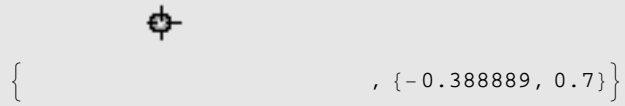
```
DynamicModule[{n = 2, p = {}, c = {}},
  EventHandler[Dynamic@Graphics[{Point[p], c}, PlotRange -> 1, Frame -> True],
    "MouseDown" :>
      (p = Union[Sort@Append[p, MousePosition["Graphics"]],
        SameTest -> (First[#1] == First[#2] &)]);
      If[Length[p] ≥ n + 1, c = interpolationCurve[p, n]])]
```



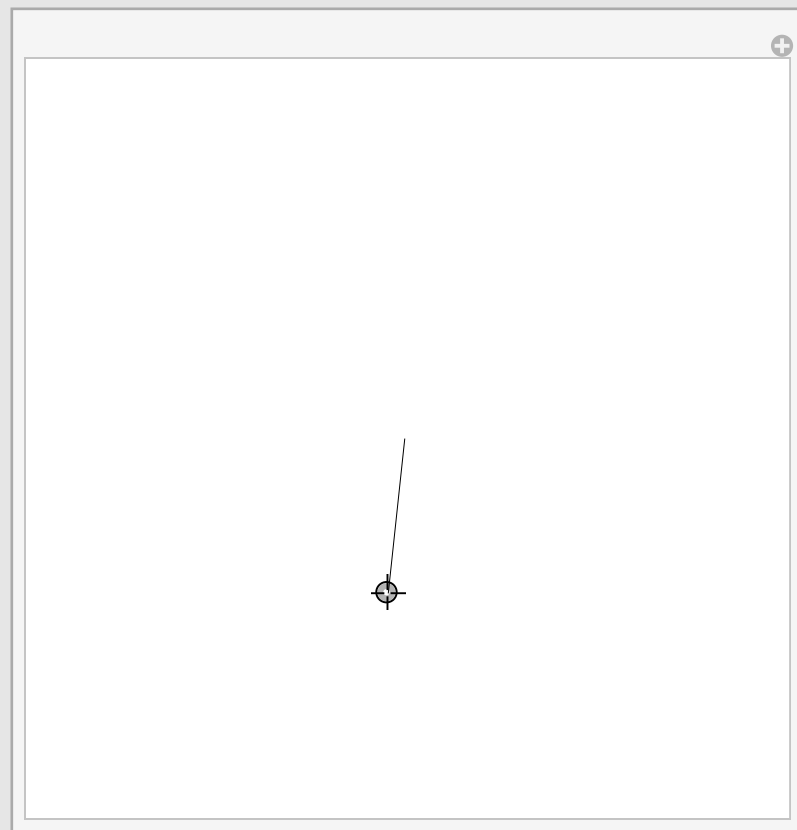
```
Graphics[Locator[{0, 0}], PlotRange -> 2]
```



```
DynamicModule[{p = {0.5, 0.5}},  
  {Graphics[Locator[Dynamic[p]], PlotRange -> 2], Dynamic[p]}]
```



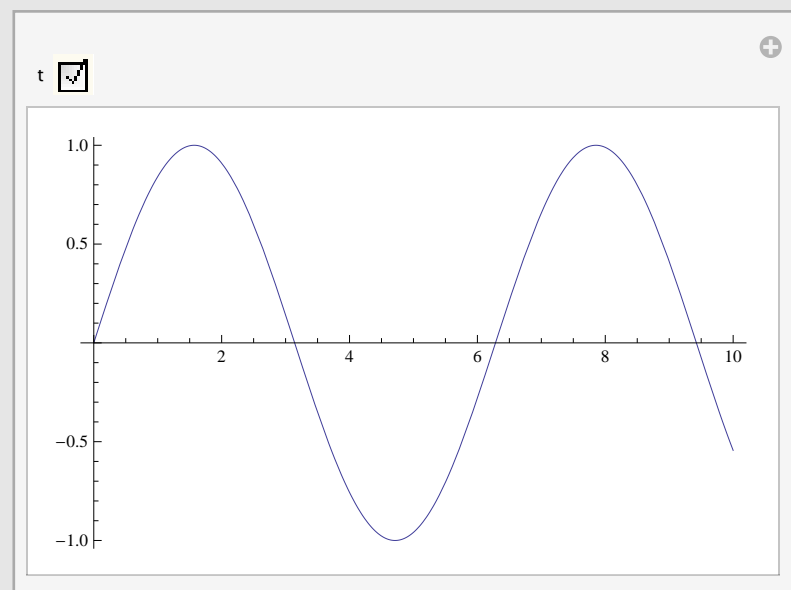
```
Manipulate[Graphics[Line[{0, 0}, p]], PlotRange -> 2], {{p, {1, 1}}, Locator}]
```



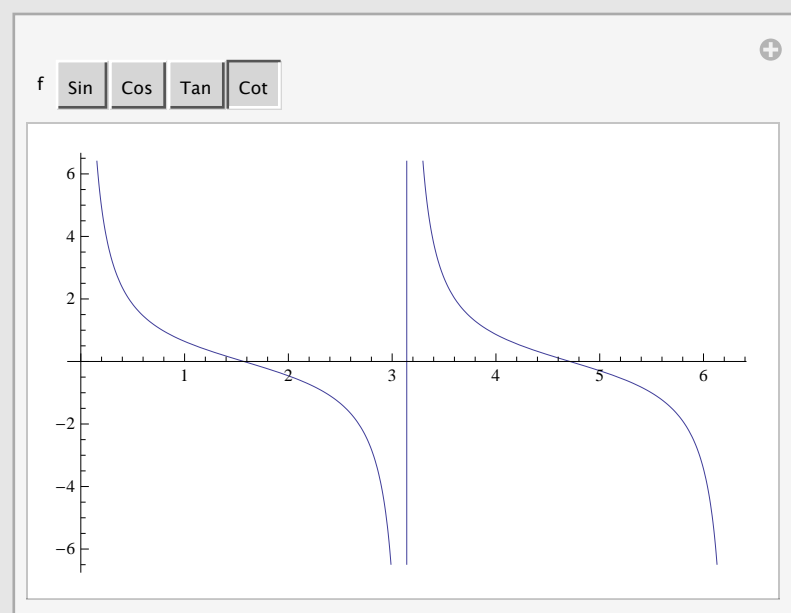
■ Manipulate options

```
Quit[]
```

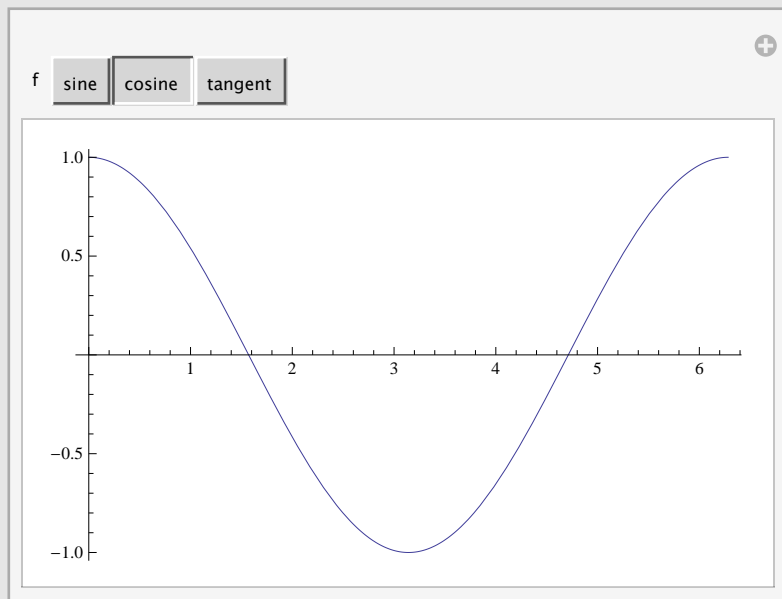
```
Manipulate[Plot[If[t, Sin[x], Cos[x]], {x, 0, 10}], {t, {True, False}}]
```



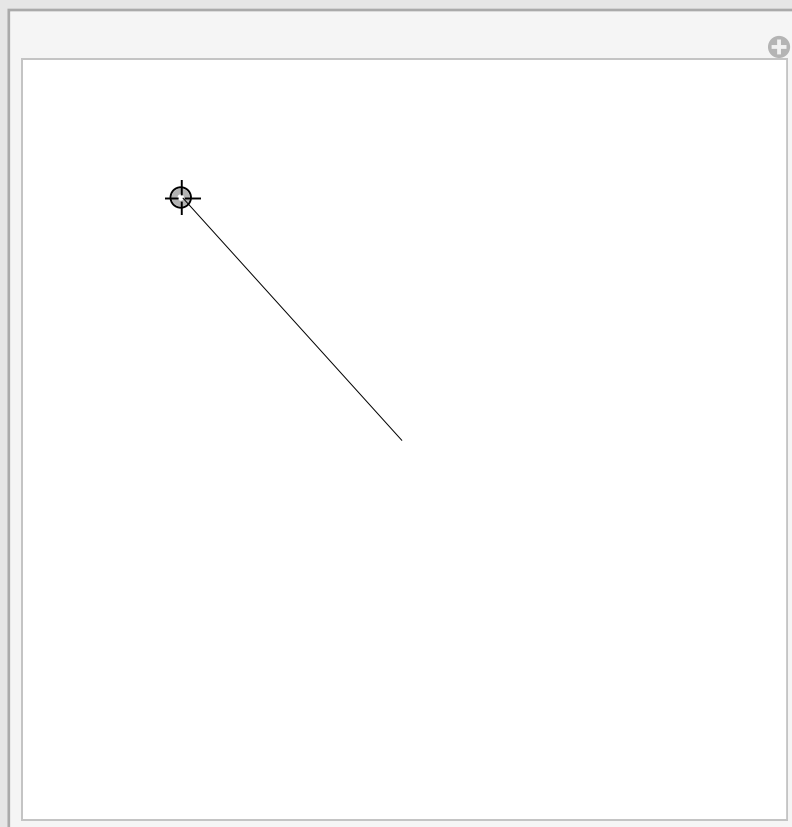
```
Manipulate[Plot[f[x], {x, 0, 2 Pi}], {f, {Sin, Cos, Tan, Cot}}]
```



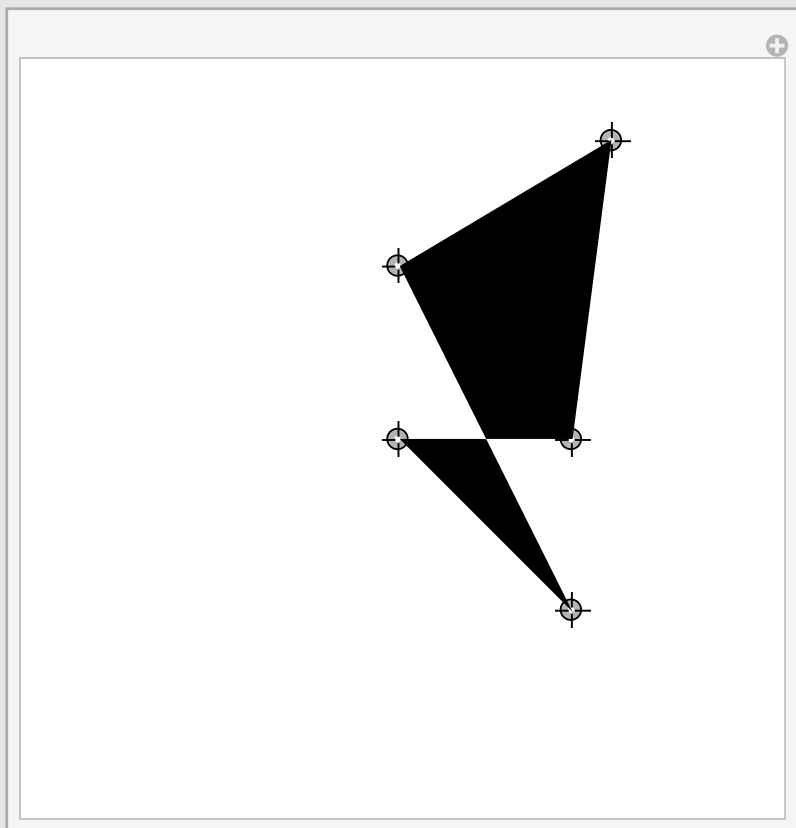
```
Manipulate[Plot[f[x], {x, 0, 2 Pi}],  
  {f, {Sin → "sine", Cos → "cosine", Tan → "tangent"}}]
```



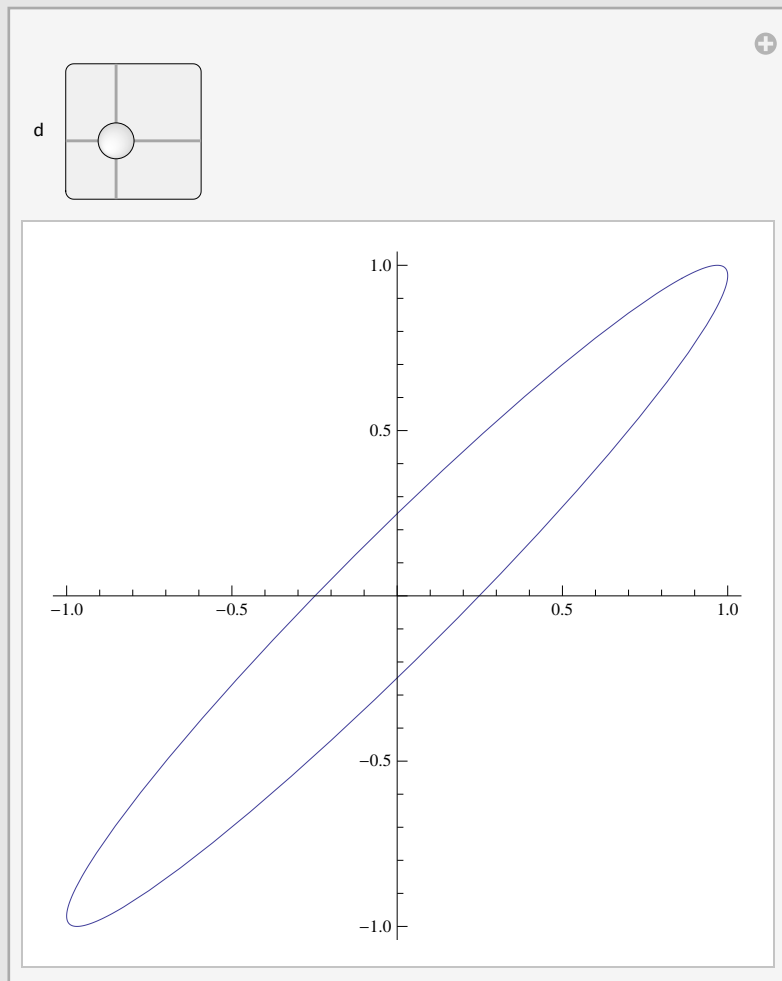
```
Manipulate[Graphics[Line[{{0, 0}, p}], PlotRange → 2], {{p, {1, 1}}, Locator}]
```



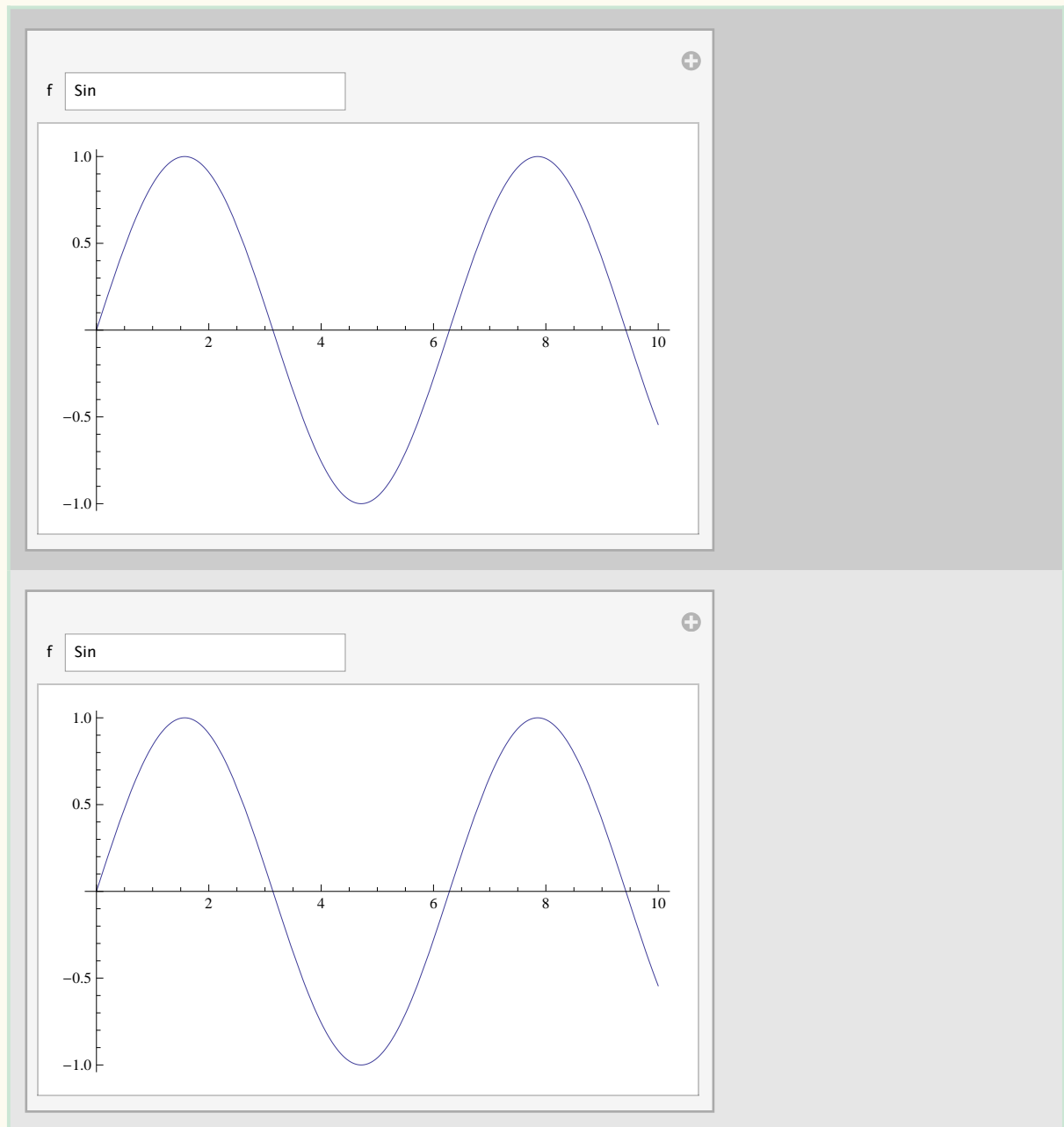
```
Manipulate[Graphics[Polygon[pt], PlotRange -> 2],  
  {{pt, {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {1, -1}}}, Locator}]
```



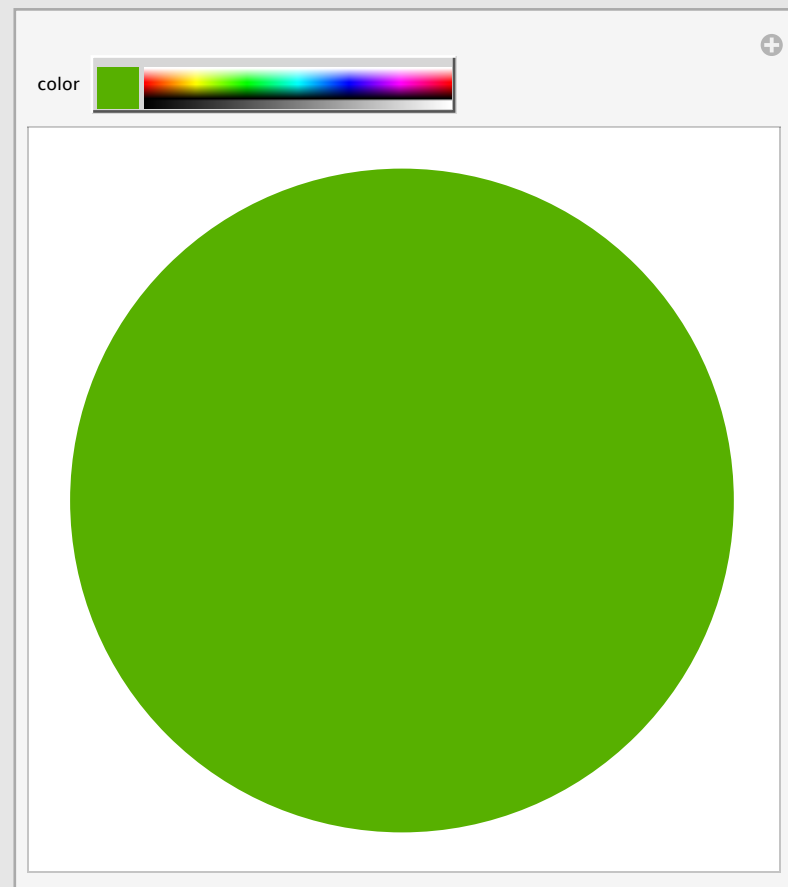
```
Manipulate[ParametricPlot[{Sin[t + d[[1]]], Sin[t + d[[2]]]}, {t, 0, 2 Pi}],  
  {d, {0, 0}, {Pi, Pi}}]
```



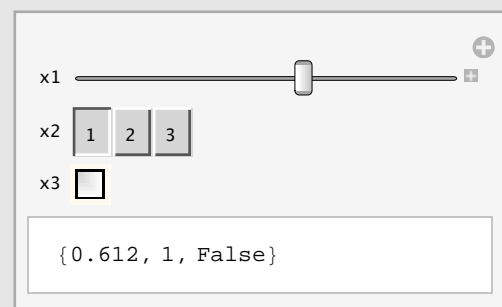
```
Manipulate[Plot[f[x], {x, 0, 10}], {f, Tan}]
```



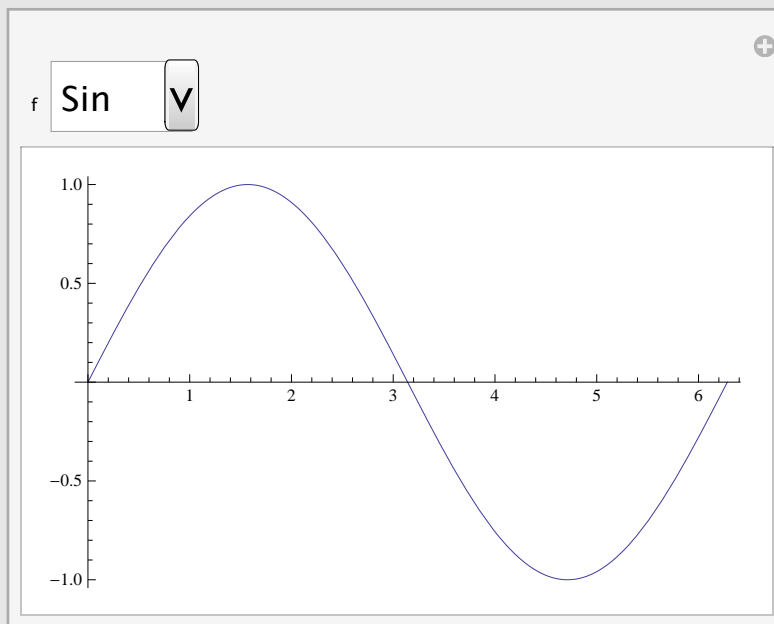
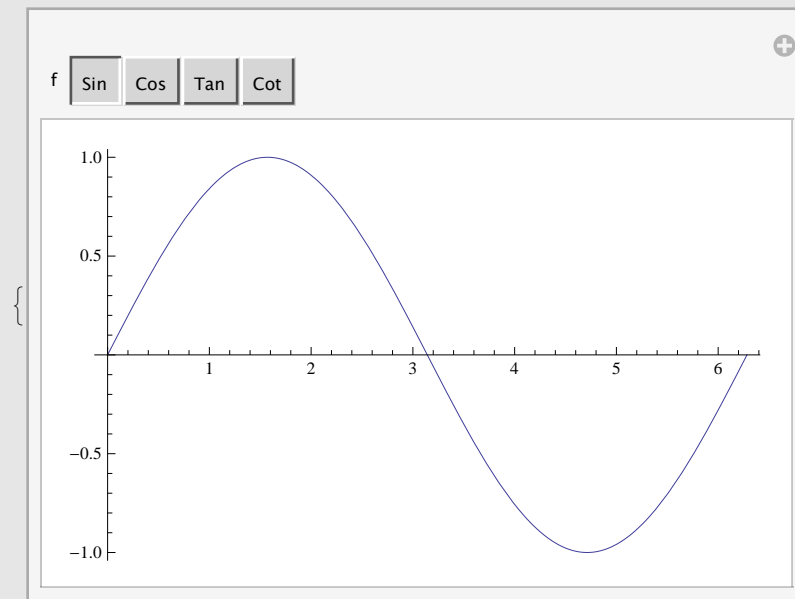
```
Manipulate[Graphics[{color, Disk[]}], {color, Purple}]
```



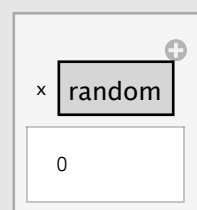
```
Manipulate[{x1, x2, x3}, {x1, 0, 1}, {x2, {1, 2, 3}}, {x3, {True, False}}]
```




```
{Manipulate[Plot[f[x], {x, 0, 2 Pi}], {f, {Sin, Cos, Tan, Cot}}], Manipulate[
  Plot[f[x], {x, 0, 2 Pi}], {f, {Sin, Cos, Tan, Cot}}, ControlType -> PopupMenu]}
```



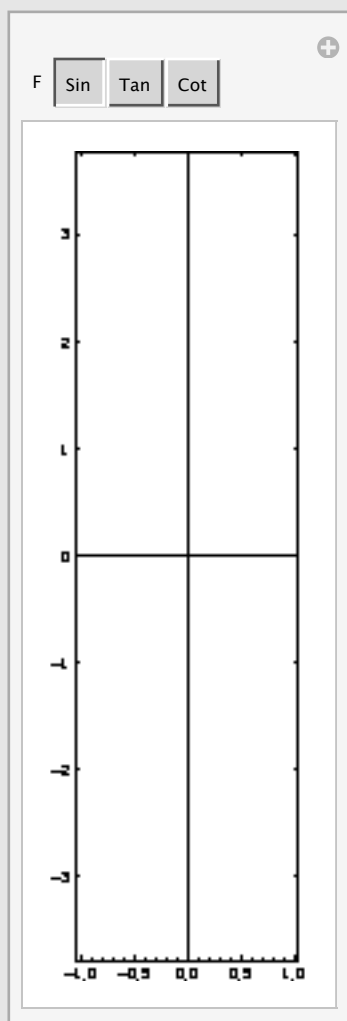
```
Manipulate[x, {{x, 0}, Button["random", x = RandomReal[]] &}]
```



```
Manipulate[x, {{x, 1}, 0, 5}, {x, Range[5]}]
```

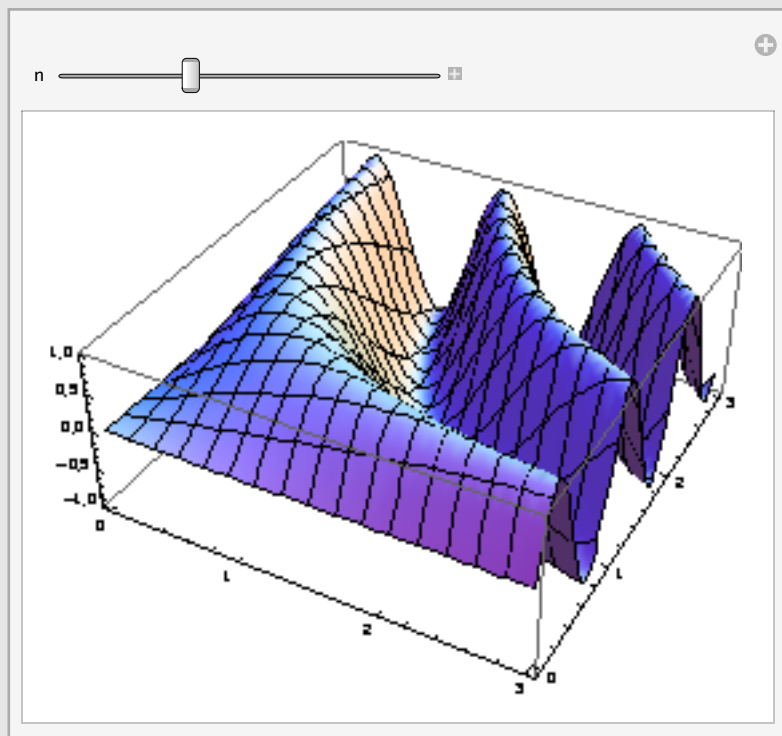


```
Manipulate[Block[{f = F[x + I y]}, ParametricPlot[  
  Evaluate[{Re[f], Im[f]}], {x, -Pi, Pi}, {y, -2, 2}], {F, {Sin, Tan, Cot}}]
```

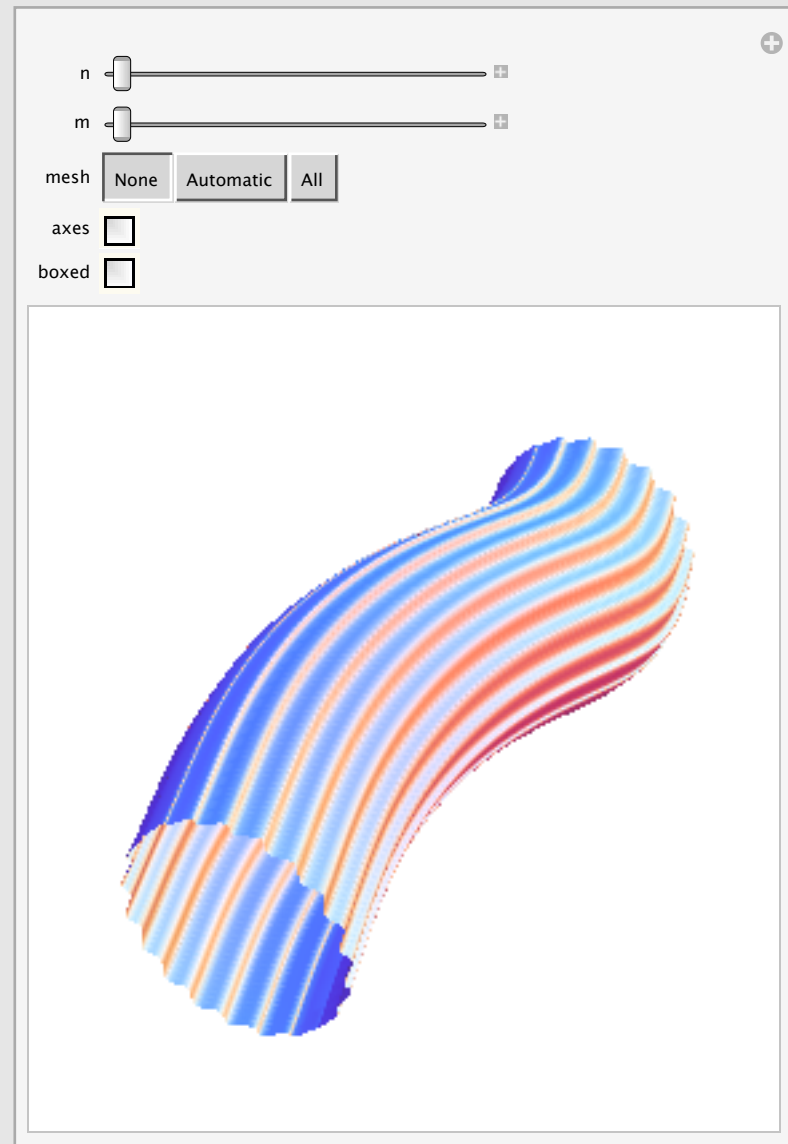


```
Quit[]
```

```
Manipulate[Plot3D[Sin[n x y], {x, 0, 3}, {y, 0, 3}], {n, 1, 4, 0.5}]
```



```
Manipulate[ParametricPlot3D[  
  {Cos[v] + 0.3 Sin[n u] + 0.04 Sin[20 v], u, Sin[v] + 0.3 Cos[m u] + 0.04 Sin[20 v]},  
  {u, - $\pi$ ,  $\pi$ }, {v, - $\pi$ ,  $\pi$ }, PlotPoints  $\rightarrow$  50, Mesh  $\rightarrow$  mesh, Axes  $\rightarrow$  axes, Boxed  $\rightarrow$  boxed],  
  {n, 1, 5}, {m, 1, 5},  
  {mesh, {None, Automatic, All}},  
  {axes, {False, True}},  
  {boxed, {False, True}}]
```

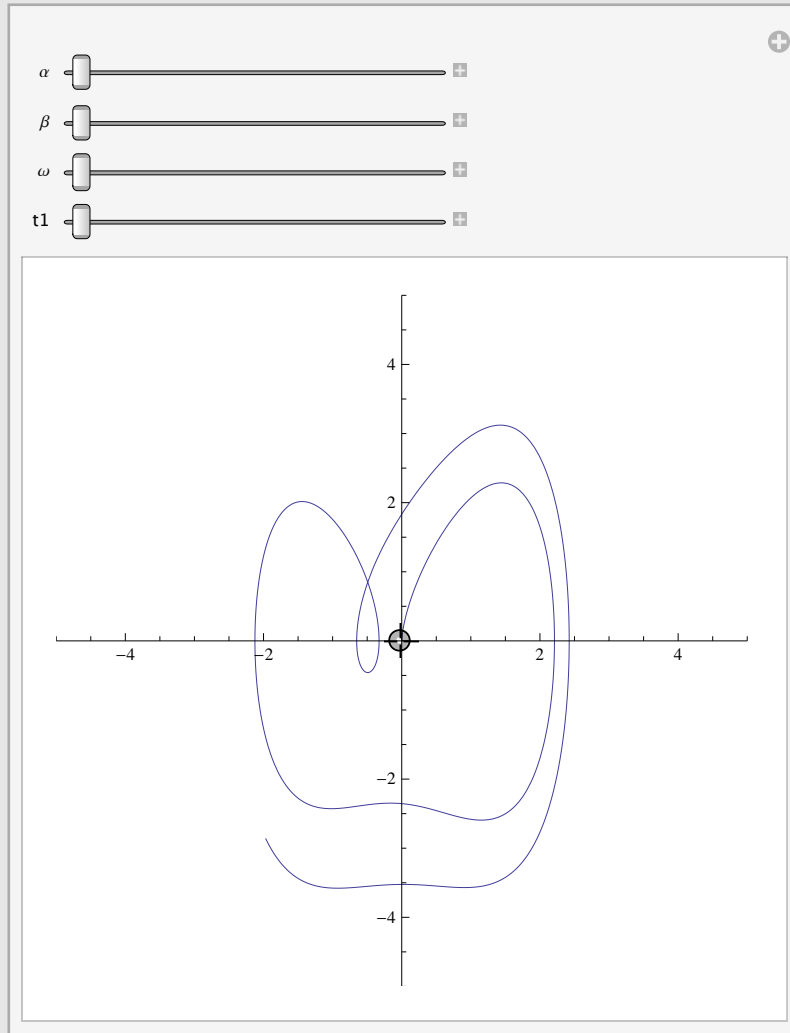


```
Quit[]
```

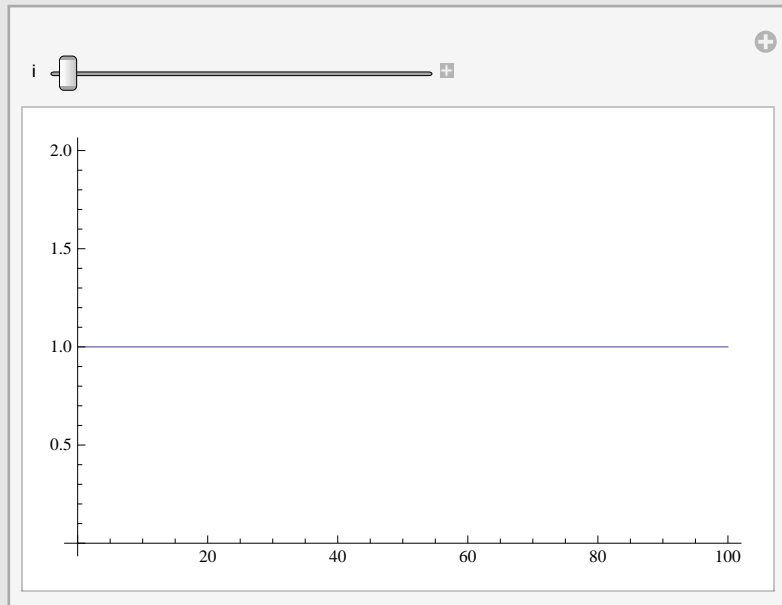
```

Manipulate[Module[{x}, With[
  {sol = NDSolve[{x'[t] + x[t]^3 ==  $\alpha$  Sin[ $\omega$  t +  $\beta$ ], x[0] == pt[[1]], x'[0] == pt[[2]]}, x,
    {t, 0, 50}], ParametricPlot[{x[t], x'[t]} /. sol, {t, 0, t1}, PlotRange -> 5]}],
  { $\alpha$ , -3, 3}, { $\beta$ , - $\pi$ ,  $\pi$ }, { $\omega$ , 1, 5}, {t1, 10, 50}, {{pt, {0, 0}}, Locator},
  SaveDefinitions -> True]

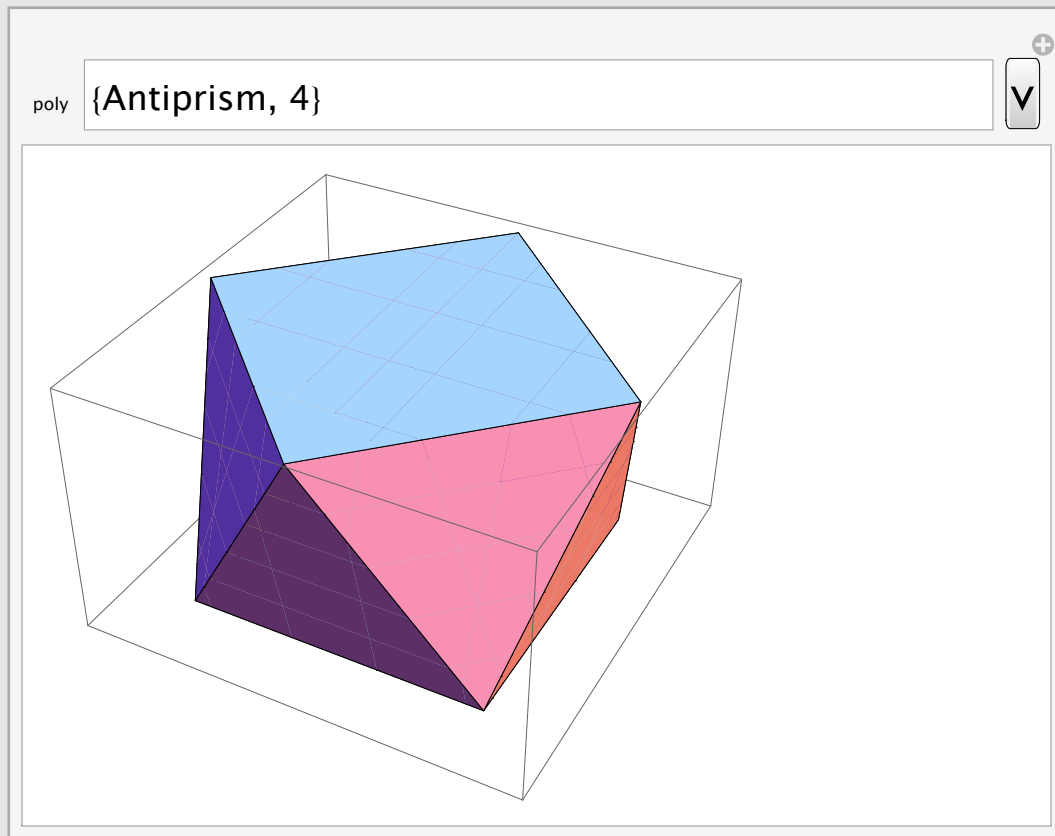
```




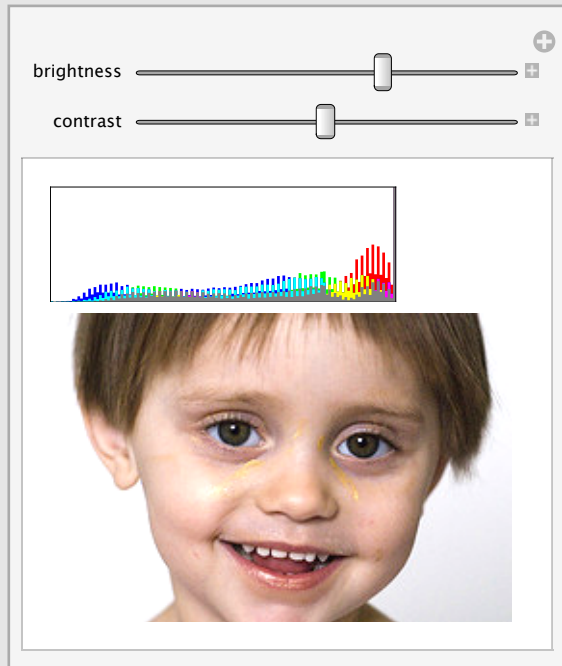
```
Manipulate[  
  Plot[Evaluate[Sum[ $\frac{1}{(k+1)^i}$ , {k, 0, n}]], {n, 0, 100}, PlotRange -> All], {i, 1, 10, 1}]
```



```
Manipulate[PolyhedronData[poly], {poly, PolyhedronData[]}]
```



```
Manipulate[With[
  {img = ImageAdjust[, {contrast, brightness}]},
  Column[{ImageHistogram[img], Image[img, ImageSize -> All]}],
  {{brightness, 0}, -1, 1}, {{contrast, 0}, -1, 1}]
```



5. More Advanced Topics.

Some Other topics

■ Dynamic Programming

Recursive programs are programs that "refer to themselves". Such programs can be very slow because they use a lot of "stack memory". One way to speed them up is by means of "dynamic programming" or "functions that remember their values". The best way to understand this method is by means of an example.

□ Fast Fibonacci numbers.

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

F_n

```
FunctionExpand[Fibonacci[n]]
```

$$\frac{1}{\sqrt{5}} \left(\left(\frac{1}{2} (1 + \sqrt{5}) \right)^n - \left(\frac{2}{1 + \sqrt{5}} \right)^n \cos(\pi n) \right)$$

Let's compare the following two definitions of the Fibonacci numbers. First we start with a usual recursive definition.

```
Fib1[0] = 0; Fib1[1] = 1;
```

```
Fib1[n_] := Fib1[n - 1] + Fib1[n - 2]
```

```
Timing[Fib1[30]]
```

```
{2.26725, 832040}
```

This is very slow and already the 30th Fibonacci number takes a noticeable time to compute. Next, we try "dynamic programming". The definition looks a little strange; note the use of := and = .

```
Clear[Fib2]
```

```
Fib2[0] = 0; Fib2[1] = 1;
```



```
Fib2[n_] := Fib2[n] = Fib2[n - 1] + Fib2[n - 2]
```

```
Timing[Fib2[30]]
```

```
{0.000267, 832.040}
```

Computing even the 50th Fibonacci number takes virtually no time.

We can see the difference between the two definitions by using the function Trace:

```
Trace[Fib1[5]]
```

```
{Fib1[5], Fib1[5 - 1] + Fib1[5 - 2], {{5 - 1, 4}, Fib1[4],
  Fib1[4 - 1] + Fib1[4 - 2], {{4 - 1, 3}, Fib1[3], Fib1[3 - 1] + Fib1[3 - 2],
    {{3 - 1, 2}, Fib1[2], Fib1[2 - 1] + Fib1[2 - 2], {{2 - 1, 1}, Fib1[1], 1},
      {{2 - 2, 0}, Fib1[0], 0}, 1 + 0, 1}, {{3 - 2, 1}, Fib1[1], 1}, 1 + 1, 2},
    {{4 - 2, 2}, Fib1[2], Fib1[2 - 1] + Fib1[2 - 2], {{2 - 1, 1}, Fib1[1], 1},
      {{2 - 2, 0}, Fib1[0], 0}, 1 + 0, 1}, 2 + 1, 3},
  {{5 - 2, 3}, Fib1[3], Fib1[3 - 1] + Fib1[3 - 2],
    {{3 - 1, 2}, Fib1[2], Fib1[2 - 1] + Fib1[2 - 2], {{2 - 1, 1}, Fib1[1], 1},
      {{2 - 2, 0}, Fib1[0], 0}, 1 + 0, 1}, {{3 - 2, 1}, Fib1[1], 1}, 1 + 1, 2}, 3 + 2, 5}
```

```
Trace[Fib2[5]]
```

```
{Fib2[5], Fib2[5] = Fib2[5 - 1] + Fib2[5 - 2],
  {{5 - 1, 4}, Fib2[4], Fib2[4] = Fib2[4 - 1] + Fib2[4 - 2],
    {{4 - 1, 3}, Fib2[3], Fib2[3] = Fib2[3 - 1] + Fib2[3 - 2],
      {{3 - 1, 2}, Fib2[2], Fib2[2] = Fib2[2 - 1] + Fib2[2 - 2],
        {{2 - 1, 1}, Fib2[1], 1}, {{2 - 2, 0}, Fib2[0], 0}, 1 + 0, 1}, Fib2[2] = 1, 1},
        {{3 - 2, 1}, Fib2[1], 1}, 1 + 1, 2}, Fib2[3] = 2, 2},
        {{4 - 2, 2}, Fib2[2], 1}, 2 + 1, 3}, Fib2[4] = 3, 3},
        {{5 - 2, 3}, Fib2[3], 2}, 3 + 2, 5}, Fib2[5] = 5, 5}
```

The first function repeatedly performs the same computation (see Fib1[3] and Fib2[3] in the output of Trace above).

We can also check what Mathematica knows about the functions Fib1 and Fib2

```
?Fib1
```

```
Global`Fib1
```

```
Fib1[0] = 0
```

```
Fib1[1] = 1
```

```
Fib1[n_] := Fib1[n - 1] + Fib1[n - 2]
```

```
DownValues[Fib1]
```

```
{HoldPattern[Fib1[0]] => 0, HoldPattern[Fib1[1]] => 1,
  HoldPattern[Fib1[n_]] => Fib1[n - 1] + Fib1[n - 2]}
```

```
?Fib2
```

```
Global`Fib2
```

```
Fib2[0] = 0
```

```
Fib2[1] = 1
```

```
Fib2[2] = 1
```

```
Fib2[3] = 2
```

```
Fib2[4] = 3
```

```
Fib2[5] = 5
```

```
Fib2[n_] := Fib2[n] = Fib2[n - 1] + Fib2[n - 2]
```

The point is that the last definition makes Fib2 remember each value that it has once computed so it never has to compute it again. The result is much better performance at the cost of some memory consumption, of course. This can be recovered by using

```
Clear[Fib2]
```

```
Fibonacci[30] // Timing
```

```
{0.000012, 832040}
```

$$e1 = \frac{1}{2}(\sqrt{5} + 1);$$

$$e2 = \frac{1}{2}(1 - \sqrt{5});$$

$$b1 = \frac{1}{10}(\sqrt{5} + 5);$$

$$b2 = \frac{1}{10}(5 - \sqrt{5});$$

```
Fib3(n_) := Expand[b1 e1n-1 + b2 e2n-1]
```

```
Fib3[30]
```

```
832040
```

Imperative (Procedural) and Functional programming

So far we have considered two programming styles that one can use in *Mathematica* - rule based and functional. There is also another style, called procedural or imperative. This is the style used by most traditional programming languages, such as C. The characteristic of this style is the use of explicit assignments to variables, and of loops that change the state of a variable. Here is an example of a procedural program which changes the state of a variable x by using assignments:

```
Clear[x]
```

```
x = 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1;
```

```
x
```

```
7
```

This can be also written as

```
FullForm[Hold[x = 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1; x = x + 1;]]
```

```
Hold[CompoundExpression[Set[x, 1], Set[x, Plus[x, 1]], Set[x, Plus[x, 1]],  
Set[x, Plus[x, 1]], Set[x, Plus[x, 1]], Set[x, Plus[x, 1]], Set[x, Plus[x, 1]], Null]]
```

Note that *Mathematica* automatically returns the value of the last argument of `CompoundExpression`. In most procedural languages no final value would be returned and you need an explicit `Return` or `Print` statement `Return[x]` or `Print[x]` at the end. In *Mathematica* you never need these statements for this purpose.

Below is another way to do the same thing, using a `Do` loop. Note that the `Do` loop does not return anything, so we need `x` at the end if we wish to return the value of `x`.

```
Clear[x]
```

```
x = 1; Do[x = x + 1, {6}]; x
```

```
7
```

```
x
```

```
7
```

```
x = 1; Do[x++, {6}]; x
```

```
7
```

Mathematica has other procedural loops: `While` and `For`. They should be used sparingly, particularly the last one which is very inefficient. In general you can get much better performance from *Mathematica* by using functional constructions: `Nest`, `Fold` and `FixedPoint` and in version 6, `Accumulate`.

■ Local Variables

Because in imperative programs assignments are used, one has to be careful not to accidentally use or redefine variables to which values may have been assigned earlier. The best way to protect oneself from this possibility is by means of local variables. *Mathematica* has three basic constructions for localizing variables: `Block`, `Module` and `With`.

```
?Block
```

```
Block[{x, y, ...}, expr] specifies that expr is to be evaluated with local values for the symbols x, y, ...  
Block[{x = x0, ...}, expr] defines initial local values for x, .... >>
```

```
x = 3; Block[{x}, x = 1]
```

```
1
```

```
x
```

```
3
```

Although inside Block we set the value of x to 1, outside it remained equal to 3. The same will happen if we use Module

```
?Module
```

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

```
x = 3;
```

```
Module[{x, y, z = 1}, x = 5; y = x + z]
```

```
6
```

```
x
```

```
3
```

Block and Module work in a quite different way. When you localize a variable in Block its value is first stored, then erased, than after Block is exited the old (stored) value is restored. In the case of Module the variables are renamed so that their names do not conflict with any other names. Another construction that localizes variables is With. Note that, unlike in Module and Block, all local variables in With must be initialized so you can't use assignments to local variables in With.

```
ClearAll[f]
```

```
f[x_List] := Module[{u = Length[x], v}, v = u + 1]
```

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

```
4
```

```
g[x_List] := Module[{u = Length[x], v = u + 1}, v]
```

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

```
u + 1
```

```
ClearAll[g]
```

```
g[x_List] := With[{u = Length[x], v = u + 1}, v]
```

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

```
1 + u
```

```
h[x_List] := Module[{u = Length[x], v = u + 1}, v]
```

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

```
1 + u
```

```
g[x_List] := Block[{u = Length[x], v = u + 1}, v]
```

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

```
4
```

Another important example:

```
foo := x
```

```
x = 1; foo
```

```
1
```

```
Block[{x = 2}, foo]
```

```
2
```

Although foo was defined outside Block, its value inside Block is changed. This does not happen when we use Module:

```
foo
```

```
1
```

```
Module[{x = 2}, foo]
```

```
1
```

Thus, Module depends only on the original definition, Block on the evaluation. (Lexical scoping vs dynamic scoping).

■ Loops and Functional Iteration

Programs written in this style change the values of some variable. In order to get the changed value one has to explicitly evaluate the variable. Here is a simple procedural program which uses the Do loop.

?Do

Do[*expr*, {*i*_{max}}] evaluates *expr* *i*_{max} times.
Do[*expr*, {*i*, *i*_{max}}] evaluates *expr* with the variable *i* successively taking on the values 1 through *i*_{max} (in steps of 1).
Do[*expr*, {*i*, *i*_{min}, *i*_{max}}] starts with *i* = *i*_{min}.
Do[*expr*, {*i*, *i*_{min}, *i*_{max}, *di*}] uses steps *di*.
Do[*expr*, {*i*, {*i*₁, *i*₂, ...}}] uses the successive values *i*₁, *i*₂,
Do[*expr*, {*i*, *i*_{min}, *i*_{max}}, {*j*, *j*_{min}, *j*_{max}}, ...] evaluates *expr* looping over different values of *j*, etc. for each *i*. >>

Timing[*x* = 1; **Do**[*x* ++, {20 000}]; *x*]

{0.011248, 20 001}

We can do the same thing by using the functional style. When programming in this style we use functions which return values rather than change states of variables. Instead of loops we use "higher functions", that is functions whose arguments are functions. One of such functions is Nest.

?Nest

Nest[*f*, *expr*, *n*] gives an expression with *f* applied *n* times to *expr*. >>

Nest[*f*, *a*, 4]

f[f[f[f[a]]]]

There is also a related function NestList

?NestList

NestList[*f*, *expr*, *n*] gives a list of the results of applying *f* to *expr* 0 through *n* times. >>

NestList[*f*, *a*, 4]

{a, f[a], f[f[a]], f[f[f[a]]], f[f[f[f[a]]]]}

Instead of using the Do loop above we can obtain the same result using the functional approach as follows:

Nest[# + 1 &, 1, 20 000] // **Timing**

{0.000682, 20 001}

The program runs much faster.

```
FoldList[f, a, {b, c, d, e}
```

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

Here *f* has to be a function of two arguments. Here is an example which shows the working of FoldList:

```
FoldList[Plus, 0, {a, b, c, d}
```

```
{0, a, a + b, a + b + c, a + b + c + d}
```

```
FoldList[Times, 1, {a, b, c, d}
```

```
{1, a, a b, a b c, a b c d}
```

Finally there is one new and very useful function that appeared in *Mathematica* 6.

```
? Accumulate
```

```
Accumulate[list] gives a list of the successive accumulated totals of elements in list. >>
```

```
Accumulate[{a, b, c, d, e}
```

```
{a, a + b, a + b + c, a + b + c + d, a + b + c + d + e}
```

The same result can be achieved using FoldList, however, Accumulate, being a more specialised function, is considerably faster.

```
FoldList[Plus, 0, {a, b, c, d, e}
```

```
{0, a, a + b, a + b + c, a + b + c + d, a + b + c + d + e}
```

```
ls = RandomInteger[{1, 100}, {1000}];
```

```
a = (Accumulate[ls]; // Timing)
```

```
{0.00005, Null}
```

```
b = (Rest[FoldList[Plus, 0, ls]]; // Timing)
```

```
{0.000465, Null}
```

```
First[b]/First[a]
```

```
9.3
```

```
Last[a] == Last[b]
```

```
True
```

This much greater speed of a more specialized function compared with a more general one is typical of *Mathematica* programming.

<http://reference.wolfram.com/mathematica/tutorial/ApplyingFunctionsRepeatedly.html>

■ Block and global variables.

One of the most common uses of Block is to change temporarily the value of a global variable. For example, the global variable \$RecursionLimit has by default the value:

```
$RecursionLimit
```

```
256
```

The reason for this is to stop accidental infinite recursion from occurring as a result of programming errors. However, this can sometimes be inconvenient. Here is a familiar example.

```
Clear[Fib]
```

```
Fib[1] = 1; Fib[2] = 1; Fib[n_] := Fib[n] = Fib[n - 1] + Fib[n - 2];
```

```
Fib[3000]
```

```
$RecursionLimit::reclim : Recursion depth of 256 exceeded. >>
```

The value of 256 for \$RecursionLimit prevents the code from working. Using Block we can temporarily change this value:

```
Clear[Fib]
```

```
Fib[1] = 1; Fib[2] = 1; Fib[n_] := Fib[n] = Fib[n - 1] + Fib[n - 2];
```

```
Block[{$RecursionLimit = 10 000}, Fib[3000]]
```

```
410 615 886 307 971 260 333 568 378 719 267 105 220 125 108 637 369 252 408 885 430 926 905 584 274 \
113 403 731 330 491 660 850 044 560 830 036 835 706 942 274 588 569 362 145 476 502 674 373 045 \
446 852 160 486 606 292 497 360 503 469 773 453 733 196 887 405 847 255 290 082 049 086 907 512 \
622 059 054 542 195 889 758 031 109 222 670 849 274 793 859 539 133 318 371 244 795 543 147 611 \
073 276 240 066 737 934 085 191 731 810 993 201 706 776 838 934 766 764 778 739 502 174 470 268 \
627 820 918 553 842 225 858 306 408 301 661 862 900 358 266 857 238 210 235 802 504 351 951 472 \
997 919 676 524 004 784 236 376 453 347 268 364 152 648 346 245 840 573 214 241 419 937 917 242 \
918 602 639 810 097 866 942 392 015 404 620 153 818 671 425 739 835 074 851 396 421 139 982 713 \
640 679 581 178 458 198 658 692 285 968 043 243 656 709 796 000
```


However, note that the global value of `$RecursionLimit` remains unchanged:

```
$RecursionLimit
```

```
256
```

Example 1: Simulating Brownian Motion

▣ One Path

```
BrownianMotion[n_] := Accumulate[Prepend[RandomReal[NormalDistribution[0, Sqrt[1/n]], {n}], 0]]
```

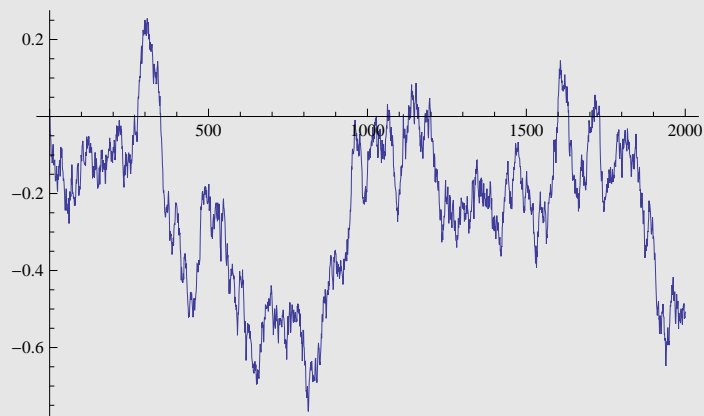
slower alternative

```
BrownianMotion[n_] :=  
  FoldList[Plus, 0, RandomReal[NormalDistribution[0, Sqrt[1/n]], {n}]]
```

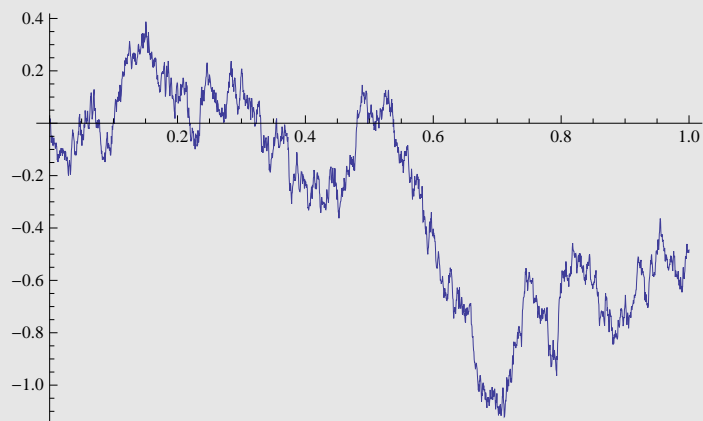
and even slower

```
BrownianMotion[n_] :=  
  NestList[# + RandomReal[NormalDistribution[0, Sqrt[1/n]]] &, 0, n]
```

```
ListLinePlot[BrownianMotion[2000]]
```



```
ListLinePlot[BrownianMotion[2000], DataRange → {0, 1}]
```

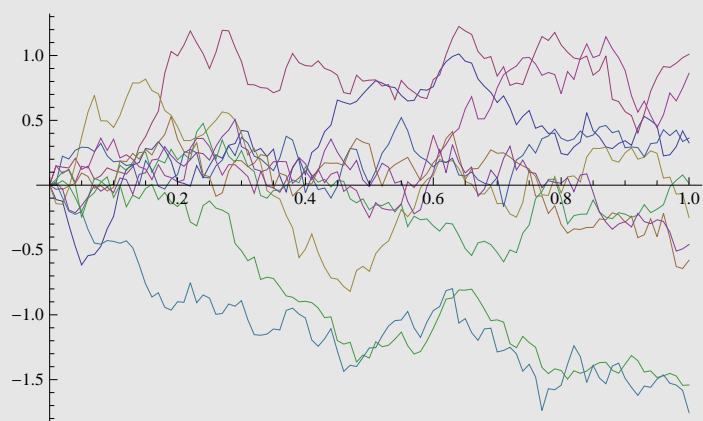


▣ Many Paths

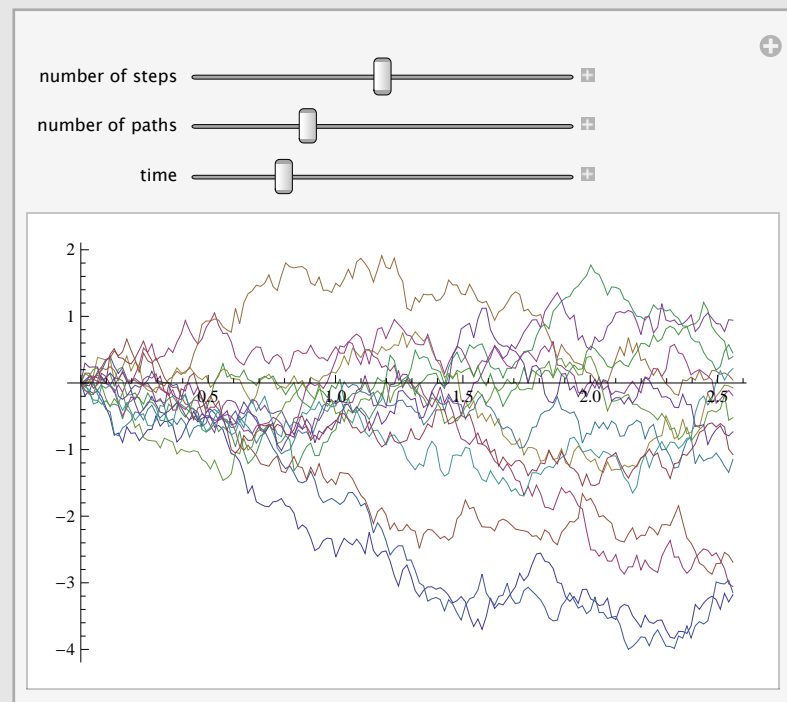
```
Clear[BrownianMotion]
```

```
BrownianMotion[time_, steps_, paths_] := Transpose[Accumulate[Join[{ConstantArray[0, paths]},  
Transpose[RandomReal[NormalDistribution[0, Sqrt[time/steps]], {paths, steps}]]]]]
```

```
ListLinePlot[BrownianMotion[1, 100, 10], DataRange → {0, 1}, PlotRange → All]
```



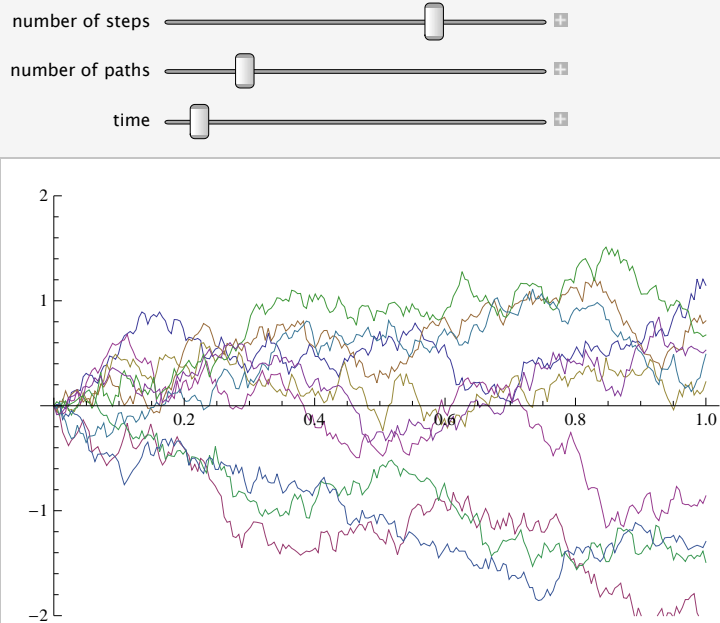
```
Manipulate[ListLinePlot[BrownianMotion[time, steps, paths], DataRange -> {0, time}, PlotRange -> All],
{{steps, 100, "number of steps"}, 10, 300, 1}, {{paths, 10, "number of paths"}, 1, 50, 1},
{{time, 1, "time"}, 0.5, 10}, SaveDefinitions -> True]
```



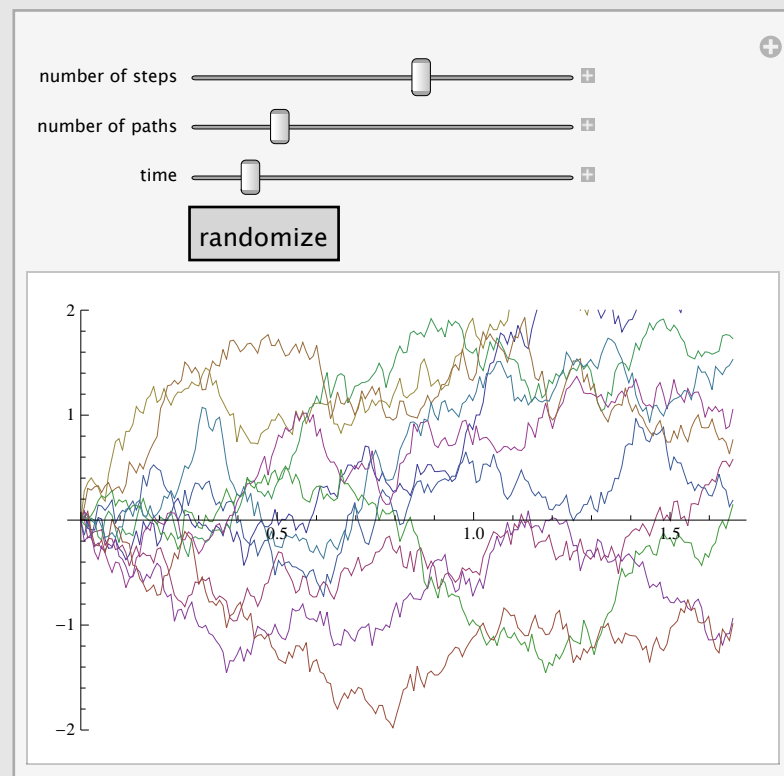
ListLinePlot::lpn : BrownianMotion[2.56, 156, 15] is not a list of numbers or pairs of numbers. >>

Manipulate[

```
BlockRandom[ListLinePlot[BrownianMotion[time, steps, paths], DataRange → {0, time}, PlotRange → {-2, 2}],
{{steps, 100, "number of steps"}, 10, 300, 1}, {{paths, 10, "number of paths"}, 1, 50, 1},
{{time, 1, "time"}, 0.5, 10}, SaveDefinition → True, Initialization →
(BrownianMotion[time_, steps_, paths_] := Transpose[Accumulate[Join[{ConstantArray[0, paths]},
RandomReal[NormalDistribution[0, Sqrt[time/steps]], {steps, paths}]]]])]
```



```
Manipulate[BlockRandom[SeedRandom[r];
  ListLinePlot[BrownianMotion[time, steps, paths], DataRange → {0, time}, PlotRange → {-2, 2}],
  {{steps, 100, "number of steps"}, 10, 300, 1}, {{paths, 10, "number of paths"}, 1, 50, 1},
  {{time, 1, "time"}, 0.5, 10}, {{r, 0, ""}}, Button["randomize", r = RandomInteger[2^64 - 1]] &,
  SaveDefinition → True, Initialization :=
  (BrownianMotion[time_, steps_, paths_] := Transpose[Accumulate[Join[{ConstantArray[0, paths]},
    RandomReal[NormalDistribution[0, Sqrt[time/steps]], {steps, paths}]]]])]
```



<http://reference.wolfram.com/mathematica/tutorial/PseudorandomNumbers.html>

A few words about Dynamic and Manipulate

In version 6 of Mathematica, new features appeared which made it possible to use the Front End in a new way. The main idea is that expressions with head `Dynamic` are updated when their “displayed form” changes. The simplest case is:

■ Dynamic

```
Dynamic[x]
```

```
0.
```

```
x = 5
```

```
5
```


```
DateString[]
```

Wed 16 Nov 2011 14:29:47

```
Dynamic[Refresh[DateString[], UpdateInterval -> Infinity]]
```

Wed 4 Jan 2012 13:06:21

```
{Slider[Dynamic[x], Appearance -> "Labeled"], Dynamic[x^2]}
```

{  0. , 0. }


```
Slider[Dynamic[x]]
```



```
DynamicModule[{x}, {Dynamic[x^2], Slider[Dynamic[x], Appearance -> "Labeled"]}]
```

{0.190969,  0.437 }

```
DynamicModule[{x},  
  {Dynamic[x^2], Slider[Dynamic[x], {0, 10, 1}, Appearance -> "Labeled"]}]
```

{64,  8 }

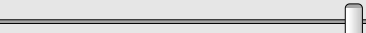
```
DynamicModule[{x}, {Dynamic[x^2], PopupMenu[Dynamic[x], Range[10]]}]
```

{16,  }

```
DynamicModule[{x}, {Dynamic[x^2], SetterBar[Dynamic[x], Range[10]]}]
```

{25,  }

```
DynamicModule[{x = 1},  
  {Dynamic[x], Dynamic[Slider[x, {0, 1}, Appearance -> "Labeled"]]}]
```

{1,  1 }

```
DynamicModule[{x = 1},  
  {Dynamic[x], Slider[Dynamic[x], {0, 1}, Appearance -> "Labeled"]}]
```

{0.817,  0.817 }

```
DynamicModule[{n = 1}, Row[{Dynamic[Plot[x^n, {x, -1, 1}, PlotRange -> All]],
  Slider[Dynamic[n], {0, 5, 1}, Appearance -> "Labeled"]}]]
```



Manipulate

```
Manipulate[x^2, {x, 1, 10, 1, SetterBar}]
```

x

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

49

```
Manipulate[#^2 &@expr, {{expr, 0, "expression"}, 0, 1, 0.1}]
```

expression

1.

```
Manipulate[#^2 &[expr], {expr, Table[i, {i, 0, 1, 0.1}]}
```

expr

0.

0.

```
Manipulate[#^2 &@expr, {{expr, 0, "expression"}, 0, 1, 0.1, Appearance -> "Labeled"}]
```

expression

0.25

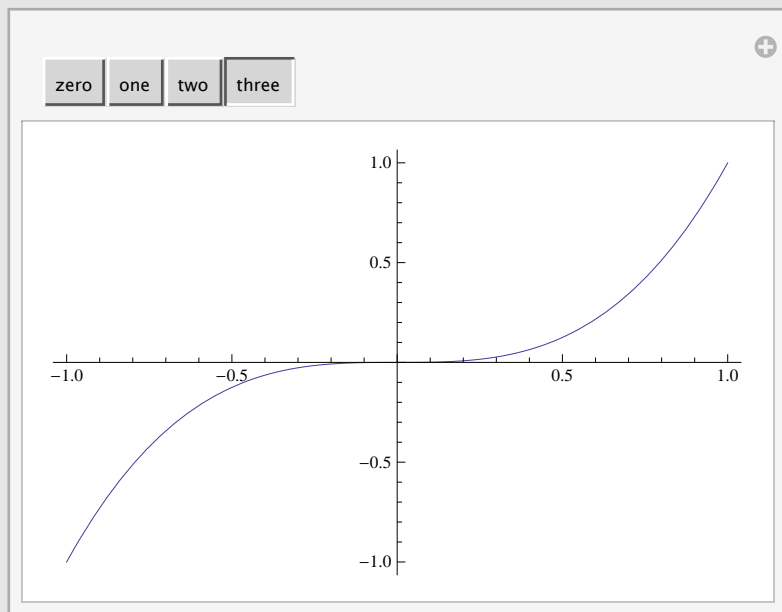
```
Manipulate[#^2 &[expr], {expr, Table[i, {i, 0, 1, 0.1}]}
```

+

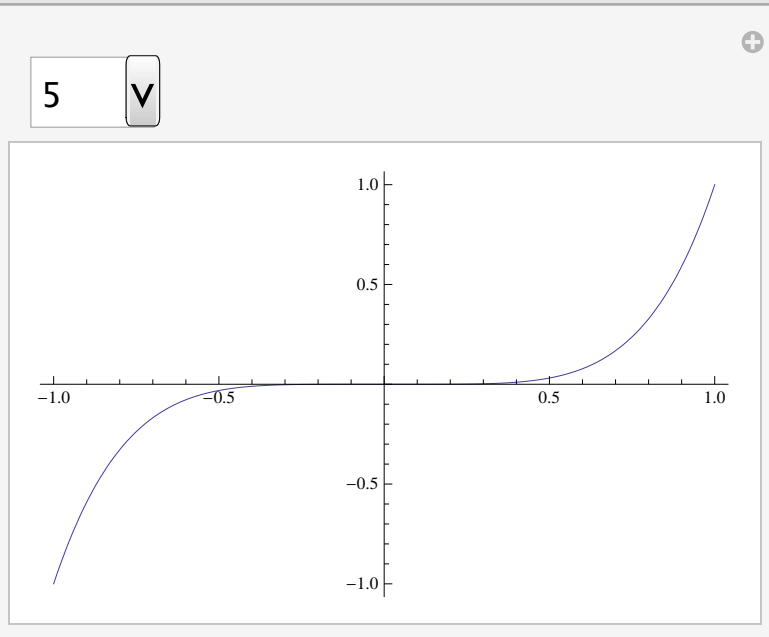
expr 0.5 V

0.25

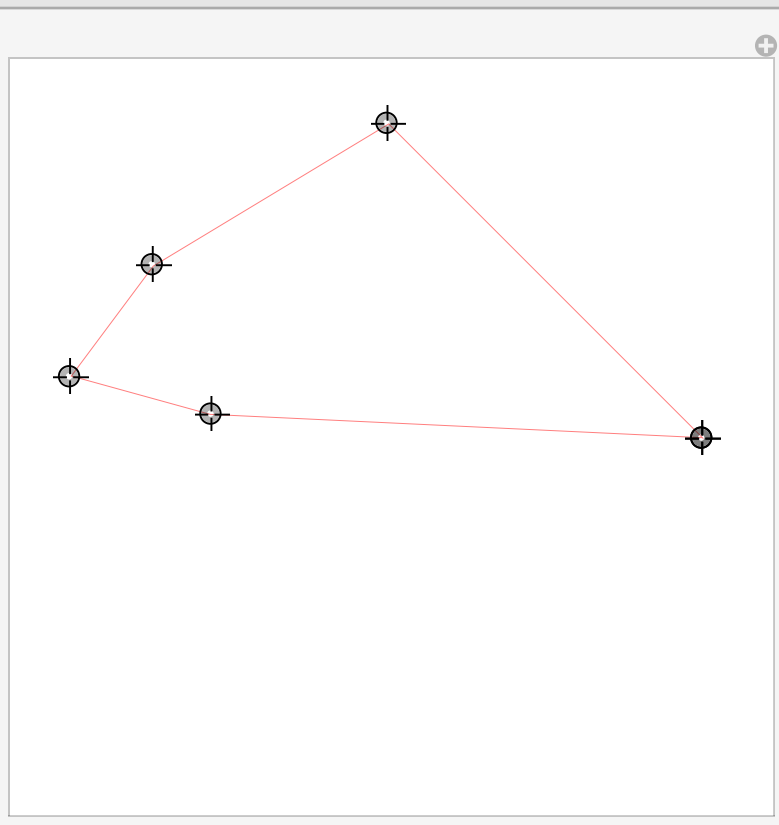
```
Manipulate[Plot[x^n, {x, -1, 1}, PlotRange → All],  
  {{n, 0, ""}, {0 → "zero", 1 → "one", 2 → "two", 3 → "three"}}]
```



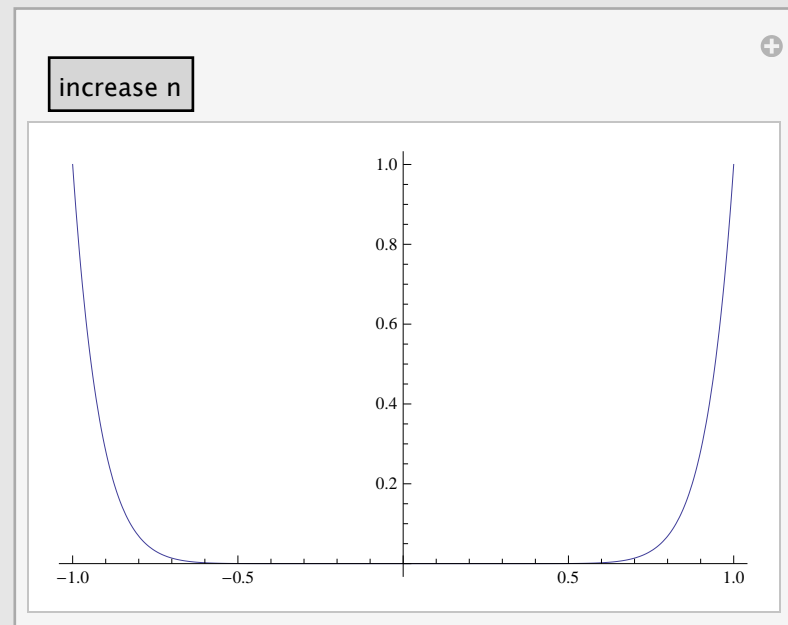

```
Manipulate[Plot[x^n, {x, -1, 1}, PlotRange -> All],
  {{n, 0, ""}, Evaluate[Table[i -> ToString[i], {i, 0, 10}]]}]
```



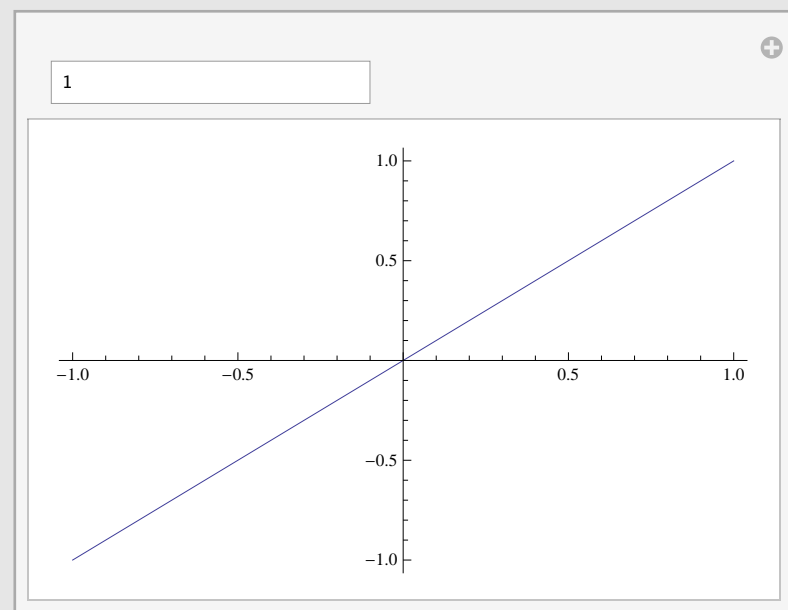
```
Manipulate[
  Graphics[{Pink, Line[pts]}, PlotRange -> 1.1],
  {{pts, {{0, 0}, {1, 0}, {0, 1}}}, Locator, LocatorAutoCreate -> True}]
```

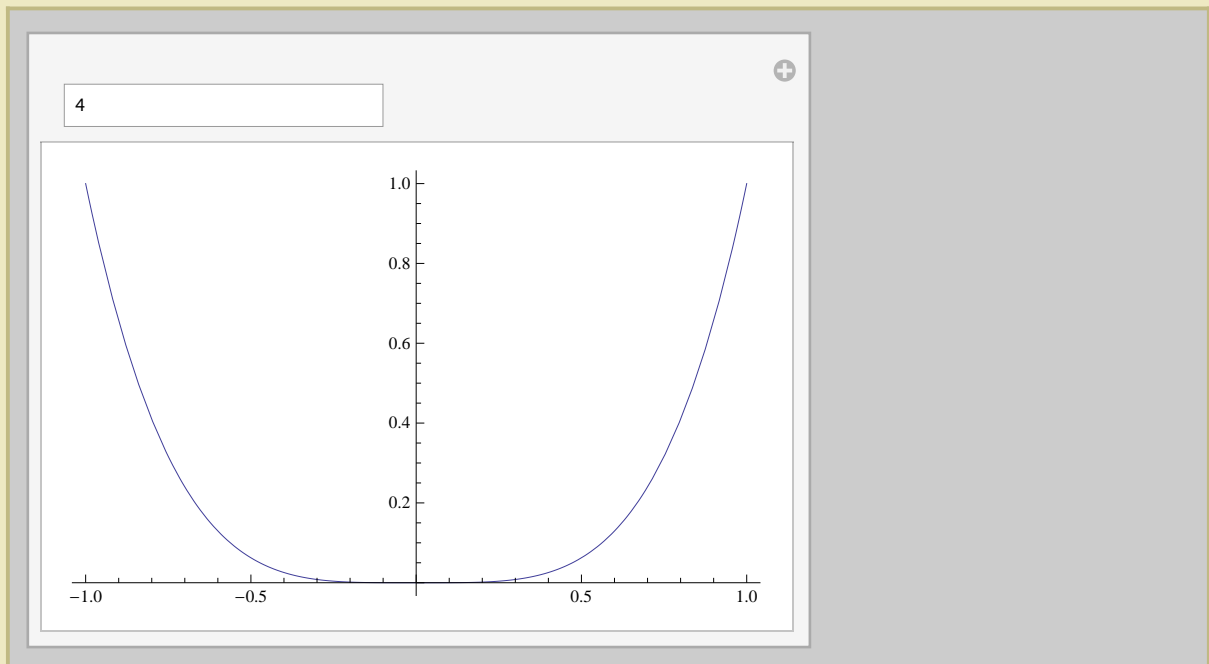


```
Manipulate[Plot[Tooltip[x^n, "x"^n], {x, -1, 1}, PlotRange -> All],  
{n, 1, ""}, Button["increase n", n = n + 1] &]
```



```
Manipulate[Plot[Tooltip[x^n, "x"^n], {x, -1, 1}, PlotRange -> All],  
{n, 1, ""}, InputField]
```





▣ Links

`http://reference.wolfram.com/mathematica/tutorial/IntroductionToDynamic.html`

`http://reference.wolfram.com/mathematica/tutorial/IntroductionToManipulate.html`