A brief introduction to Mathematica

Note: This document is part of an appendix of a book that I hope to publish. The choice of material is driven by the needs of that book, so it is a bit odd as a general introduction. Nonetheless, it does cover some basics and I've got it, so I figured I'd make it available.

Mathematica is an immense computer software package for doing mathematical computation and exploration. It contains hundreds of mathematical functions, commands for producing graphics, and a complete programming language. This appendix is a brief introduction to *Mathematica*, focusing on the tools used in this book. The serious user of *Mathematica* will need more comprehensive reference and tutorial materials. While there are a not many resources available specifically for the latest version of *Mathematica* (V6, which this text uses), there are several excellent references which adequately cover the core programming tools...

■ The very basics

At the most basic level, *Mathematica* may be used interactively, like a calculator. Commands to be evaluated are entered into *input cells*, which are displayed **`in bold input`** in this text. They are evaluated by pressing the ENTER key. (Note that ENTER is different from RET which is used to start a new line.) Results are displayed in *output cells*. Here is a simple example:

```
2 + 2
```

```
4
```

You can enter longer expressions over several lines. You can also include comments in input cells by enclosing them in **`(* These *)`**

```
(5 * 7 - 32 / 6) / 4
(* Here is a comment *)
```

$$\frac{89}{12}$$

Notice that we get a fraction. *Mathematica* is capable of exact computations, rather than decimal approximations. We can always get numerical approximations using the **`N`** function (for numerical). In the following line, **`%`** refers to the previous result and **`// N`** passes that result to the **`N`** function

```
% // N
```

```
7.41667
```

Mathematica's ability to do exact computations is truly impressive.

```
2^1000
```

```
10 715 086 071 862 673 209 484 250 490 600 018 105 614 048 117 055 336 074 437 503 883 703 510 511 249 361 224 \
 931 983 788 156 958 581 275 946 729 175 531 468 251 871 452 856 923 140 435 984 577 574 698 574 803 934 567 \
 774 824 230 985 421 074 605 062 371 141 877 954 182 153 046 474 983 581 941 267 398 767 559 165 543 946 077 \
 062 914 571 196 477 686 542 167 660 429 831 652 624 386 837 205 668 069 376
```

Approximations may be computed to any desired degree of accuracy.. Here we use the **`N`** function again along with an optional second argument to obtain 100 decimal digits of π.

```
N[Pi, 100]
```

```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482\
 5342117068
```

Mathematica has extensive knowledge about mathematical functions.

```
Sin[Pi / 3]
```

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

Note: *built in function names are always capitalized and the arguments are enclosed in square brackets. Mathematica* can do algebra.

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

$1 + 10\,x + 45\,x^2 + 120\,x^3 + 210\,x^4 + 252\,x^5 + 210\,x^6 + 120\,x^7 + 45\,x^8 + 10\,x^9 + x^{10}$

```
Factor[x^4 - x^3 - 2 x - 4]
```

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

Mathematica can do calculus.

```
D[x^2, x]
```

$2\,x$

```
Integrate[Sin[x], {x, 0, Pi}]
```

2

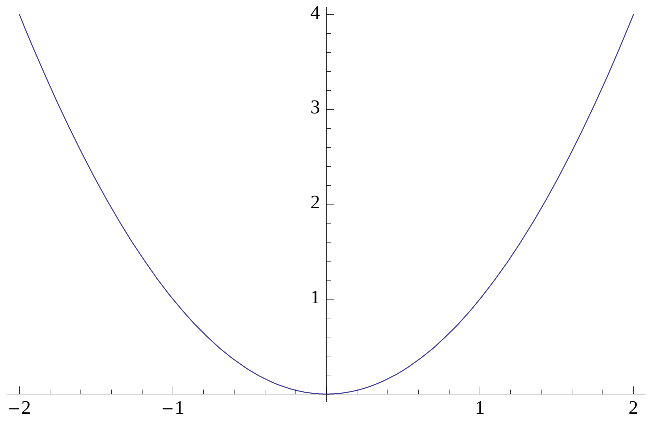
Mathematica can compute sums.

```
Sum[2 / 9^n, {n, 1, Infinity}]
```

$$\frac{1}{4}$$

Mathematica can plot functions.

```
Plot[x^2, {x, -2, 2}]
```



■ Brackets [], braces {}, and parentheses ()

Brackets, braces, and parentheses all have distinct purposes in *Mathematica*. Brackets **[]** are used to enclose the arguments of a function such as:

```
N[Tan[1], 20]
```

```
1.5574077246549022305
```

Parentheses **()** are used for grouping in mathematical expressions such as:

```
(2 * 5 - 1) / 3
```

```
3
```

Braces **{}** are used to form lists. Lists play a very important role in *Mathematica* as they are a fundamental data structure. Many functions automatically act on the individual elements of a list.

```
N[{Sqrt[2], E}, 20]
```

```
{1.4142135623730950488, 2.7182818284590452354}
```

Some commands return lists. The **Table** and **Range** commands are two of the most important such commands. The **Table** command has the syntax **Table[expr,{x,xMin,xMax}]**. This evaluates **expr** at the values **xMin**, **xMin+1**, …, **xMin+n**, where **n** is the largest integer so that **xMin+n** ≤ **xMax**. For example, here are the first 10 natural numbers squared.

```
Table[x^2, {x, 1, 10}]
```

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

It is sometimes convenient to use a slightly more general version of **Table** which specifies the step size: **Table[expr,{x,xMin,xMax,step}]**, where **step** represents the step size.

```
Table[x^2, {x, 1, 10, .5}]
```

```
{1., 2.25, 4., 6.25, 9., 12.25, 16., 20.25, 25.,
 30.25, 36., 42.25, 49., 56.25, 64., 72.25, 81., 90.25, 100.}
```

The more specific version **Table[expr,{n}]** simply produces **n** copies of **expr**. We will see several instances when this is useful. **Table** accepts multiple variables to create nested lists. In the following example, *x* is fixed for each sublist and ranges from 0 to 2 as we move from list to list; *y* ranges from 1 to 4 inside each sublist.

```
Table[x + y, {x, 0, 2}, {y, 1, 4}]
```

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

The **Range** command simply produces a list of numbers given a certain **min**, **max**, and **step**. This is not as limited as it might seem, since functions can act on lists.

```
Range[10]^2
```

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

```
Range[1, 10, .5]^2
```

```
{1., 2.25, 4., 6.25, 9., 12.25, 16., 20.25, 25.,
 30.25, 36., 42.25, 49., 56.25, 64., 72.25, 81., 90.25, 100.}
```

■ Entering typeset expressions

Up until this point, it has been fairly clear how to enter all of the example input cells. *Mathematica* allows input to resemble mathematical notation much more closely, however. For example, the following are equivalent.

```
N[Pi^2 / 6]
```

```
1.64493
```

```
N[π²/6]
```

```
1.64493
```

There are two basic ways of entering typeset expressions - using palettes or using keyboard shortcuts. Palettes are more intuitive, but typically take longer. The best approach is usually to know a few important keyboard shortcuts and to use the palettes when necessary.

Here is how to enter $\frac{\pi^2}{6}$ using a palette.
1) Select BasicMathInput under the Palettes menu.

2) Click on the $\frac{\square}{\square}$ button in the first row

3) Click on the $\square^{\square}$ button near the bottom row.

4) Click on the π button near the middle of the palette.

5) Press the TAB key.

6) Type "2".

7) Press the TAB key.

8) Type "6".

9) Press the right arrow twice key to exit the typeset expression.

Here is how to enter $\frac{\pi^2}{6}$ using the keyboard.
1) Press the ESC key.

2) Type "pi".

3) Press the ESC key.

4) Press the CTRL and ^ keys simultaneously.

5) Type "2".

6) Press the right arrow key.

7) Press the CTRL and / keys simultaneously.

8) Type "6".

9) Press the right arrow twice key to exit the typeset expression.

There is a more comprehensive discussion of *Mathematica*'s typesetting capabilities in [Gl, Gr 2000].

■ Defining constants and functions

Constants may be defined in the natural way.

```
c = 7
```

```
7
```

The symbol **c** will now be treated as 7.

```
c + 3
```

```
10
```

Symbol names may be longer alphanumeric strings and may refer to more complicated objects

```
theList = Range[10] + c
```

```
{8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

We may still act on this object.

$\mathtt{theList}^2$

```
{64, 81, 100, 121, 144, 169, 196, 225, 256, 289}
```

We may clear the contents of the symbols.

```
Clear[c, theList]
```

```
c + 3
```

```
3 + c
```

$\mathtt{theList}^2$

$\mathtt{theList}^2$

Here is how to define the function $f(x) = x + 2$.

```
f[x_] := x + 2
```

We may now plug in any value we want for **x**.

```
f[3]
```

```
5
```

$\mathtt{f}[\mathtt{y}^2]$

$2 + \mathtt{y}^2$

The important things to remember when defining a function are 1) (almost) always use a **:=** sign and 2) use underscores after variable names in the function declaration to the left of the **:=** sign. The underscore is used because this is how *Mathematica* recognizes a *pattern*. Pattern recognition is a central part of *Mathematica* and functions are defined in terms of patterns. The **:=** sign is the abbreviated form of **SetDelayed**, as opposed to **=** which is the abbreviated form of **Set**. The difference is that using **=** sets the definition immediately while using **:=** waits until the expression is actually evaluated, which is usually what you want when declaring a function. Here is an example where the difference is important. Suppose we want to define a function **diff** that automatically differentiates with respect to **x**. The correct way to do this is as follows. (Note that the semi-colon at the end of the first line suppresses the output of that command. The semi-colon is frequently used when entering multiple lines of code.)

```
Clear[f];
diff[f_] := D[f, x];
diff[x^2]
```

```
2 x
```

If we use **=** instead of **:=**, the function does not work properly.

```
Clear[diff];
diff[f_] = D[f, x];
diff[x^2]
```

```
0
```

The problem is that **D[f,x]** is evaluated immediately and returns 0.

```
D[f, x]
```

```
0
```

Using **:=** tells *Mathematica* to wait until some expression is substituted for **f** to take the derivative.

Sometimes it is convenient to use a *pure function*. A pure function has the syntax **expr &**, where **expr** is a *Mathematica* expression involving the **#** symbol and the **&** operator tells *Mathematica* to treat **expr** as a function with argument **#**. For example, **#^#&** is a pure function which raises a number (or possibly another *Mathematica* object) to itself. We use square brackets **[]** to plug in an argument just as with any other function.

```
#^# &[3]
```

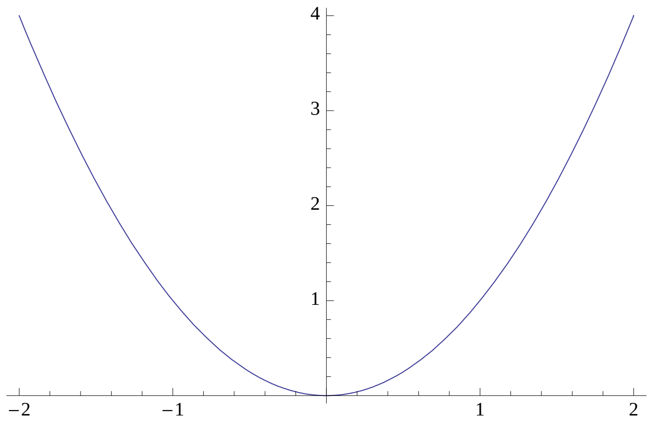
```
27
```

As we will see shortly, pure functions are particularly useful in the context of list manipulation.

■ Basic graphics

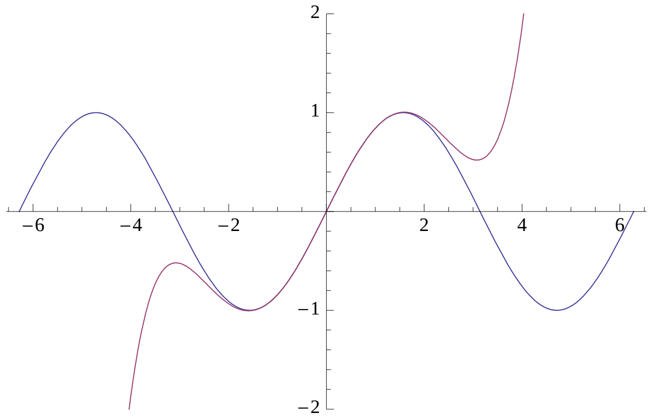
There are several basic commands for plotting functions or data. The most basic is the **Plot** command. For example, we can plot the function x^2 over the range $-2 \leq x \leq 2$ (denoted **{x,-2,2}**).

```
Plot[x^2, {x, -2, 2}]
```



We may plot more than one function on the same graph by using a list of functions. Here's the plot of sin(x) together with a polynomial approximation.

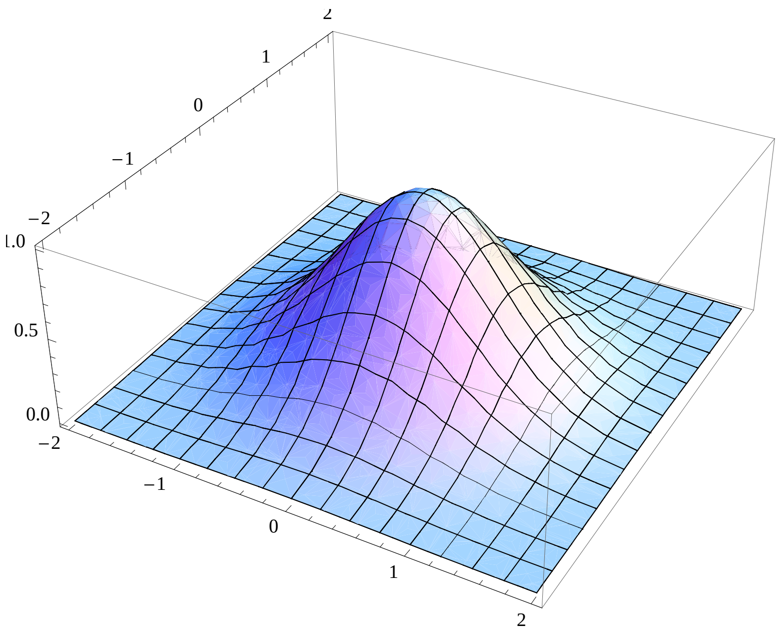
```
Plot[{Sin[x], x - x^3/6 + x^5/120}, {x, -2 π, 2 π},
 PlotRange → {-2, 2}]
```



Note that we have also included the option **`PlotRange→{-2,2}`** which controls the vertical range in the plot. The **`Plot`** command has many such options which you may examine through the online help.

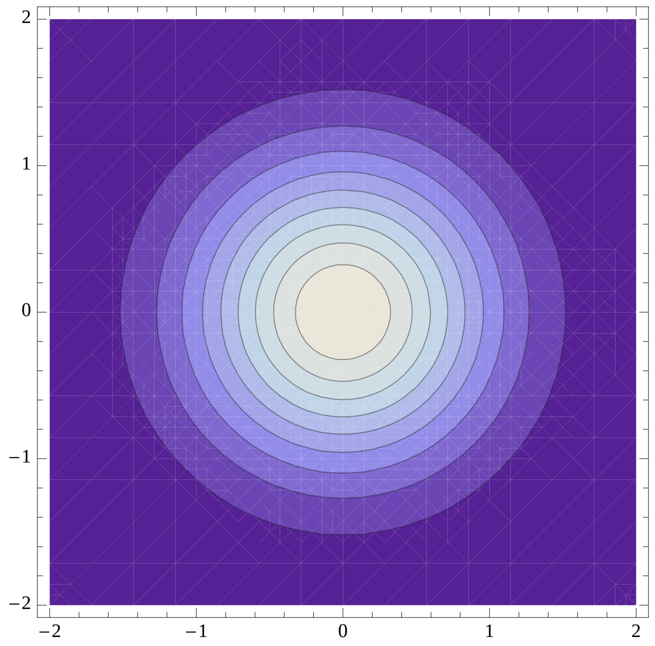
Three dimensional plots of functions of two variables may be generated using the **`Plot3D`** command.

```
Plot3D[Exp[-(x^2 + y^2)], {x, -2, 2}, {y, -2, 2}]
```



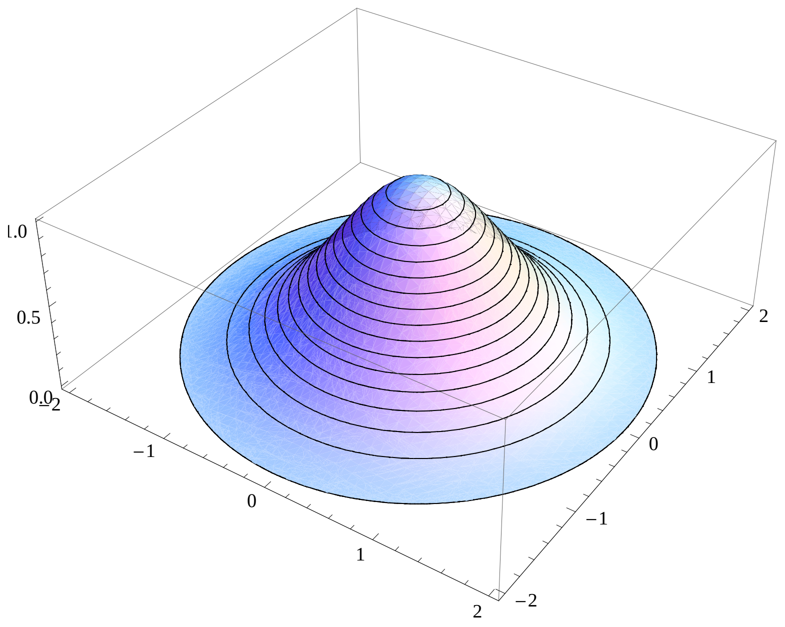
The **`ContourPlot`** command offers another way to visualize a function of two variables.

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



Use of the **`MeshFunctions`** option makes it easy to see how these are related.

```
Plot3D[Exp[-(x^2 + y^2)], {x, -2, 2}, {y, -2, 2},
 MeshFunctions → {#3 &}, PlotPoints → 50,
 RegionFunction → (#3 > 0.02 &)]
```

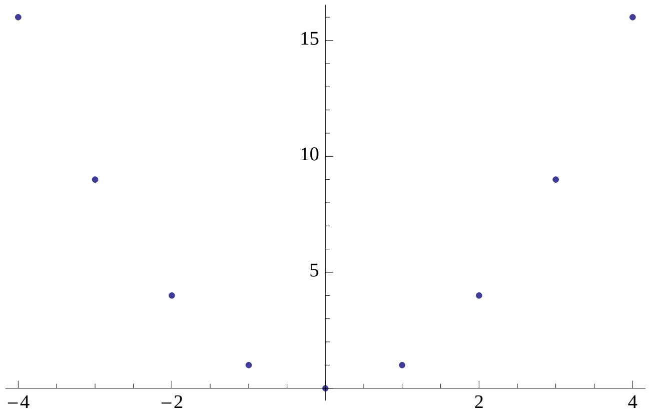


The **`ListPlot`** command plots a list of points in the plane.

```
Table[{x, x^2}, {x, -4, 4}]
```

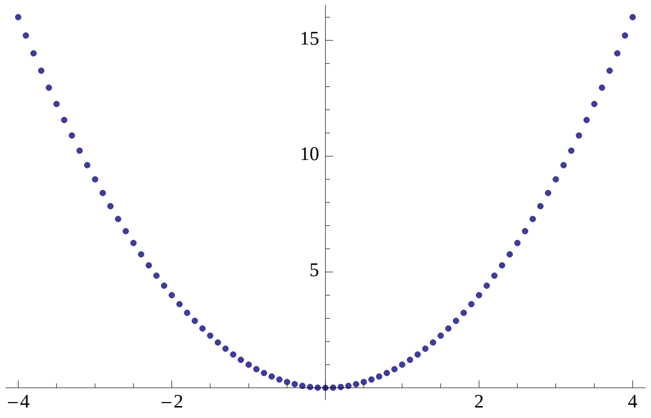
```
{{-4, 16}, {-3, 9}, {-2, 4}, {-1, 1}, {0, 0}, {1, 1}, {2, 4}, {3, 9}, {4, 16}}
```

```
ListPlot[%]
```



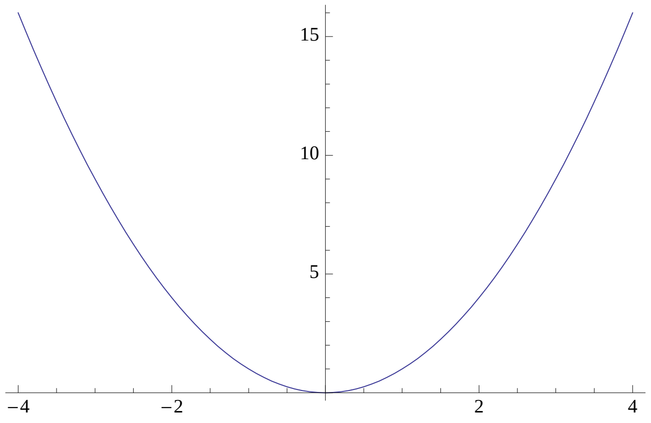
When plotting many points, it makes sense to assign the output of the **`Table`** command to a variable and to suppress the output.

```
data = Table[{x, x^2}, {x, -4, 4, .1}];
ListPlot[data]
```



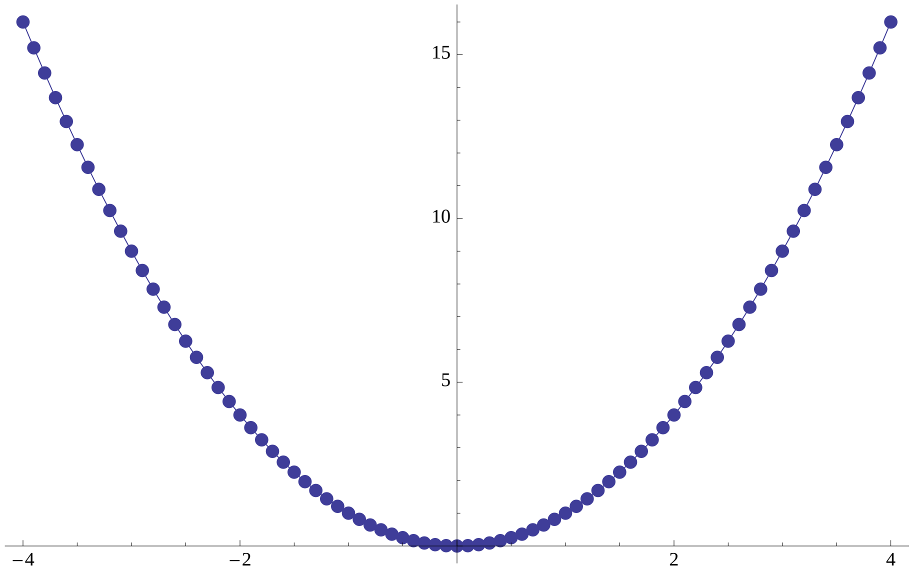
If you want the dots connected, you can use **`ListLinePlot`**.

```
ListLinePlot[data]
```



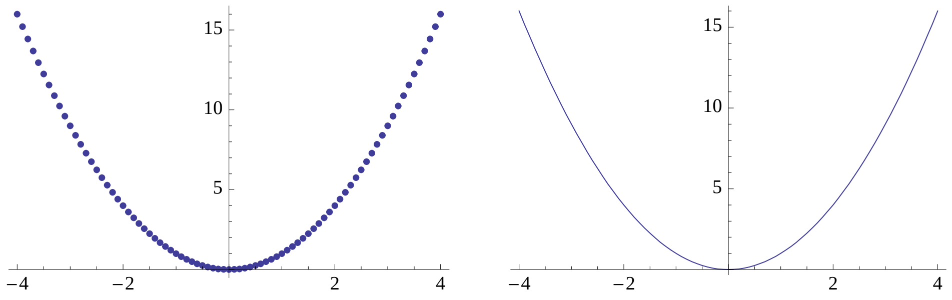
Again, a semi-colon suppresses the graphical output, but it does suppress textual output. This is convenient when you don't want to see the output, but want to combine it with subsequently generated graphics. For example, to plot the parabola above together with the sample points chosen to generate it, we could generate both plots separately, store those results using variables, and use the **Show** command to combine those plots.

```
plot1 = ListPlot[data,
   PlotStyle → {PointSize[.015]}];
plot2 = ListLinePlot[data];
Show[plot1, plot2]
```



We may also use a **GraphicsGrid** to display both plots separately.

```
Show[GraphicsGrid[{{plot1, plot2}}]]
```



Solving equations

Mathematica has powerful built in techniques for solving equations algebraically and numerically. The basic command is **Solve**. Here is how to use **Solve** to find the roots of a polynomial.

```
Solve[x^3 - 2 x - 4 == 0, x]
{{x → -1 - i}, {x → -1 + i}, {x → 2}}
```

The general quintic cannot be exactly solved in terms of roots, so *Mathematica* has it's own representation of such roots.

```
Solve[x^5 - 2 x - 3 == 0, x]
{{x → Root[-3 - 2 #1 + #1^5 &, 1]}, {x → Root[-3 - 2 #1 + #1^5 &, 2]},
 {x → Root[-3 - 2 #1 + #1^5 &, 3]}, {x → Root[-3 - 2 #1 + #1^5 &, 4]}, {x → Root[-3 - 2 #1 + #1^5 &, 5]}}
```

Note that the **NSolve** command is similar, but returns numerical approximations.

```
NSolve[x^5 - 2 x - 3 == 0, x]
```

```
{{x → -0.958532 - 0.498428 i}, {x → -0.958532 + 0.498428 i},
 {x → 0.246729 - 1.32082 i}, {x → 0.246729 + 1.32082 i}, {x → 1.42361}}
```

The **Solve** and **NSolve** commands work well with polynomials and systems of polynomials. However, many equations involving transcendental functions are beyond their capabilities. Consider, for example, the simple equation $\cos(x) = x$. **NSolve** will not solve the equation, but returns a statement to let you know this.

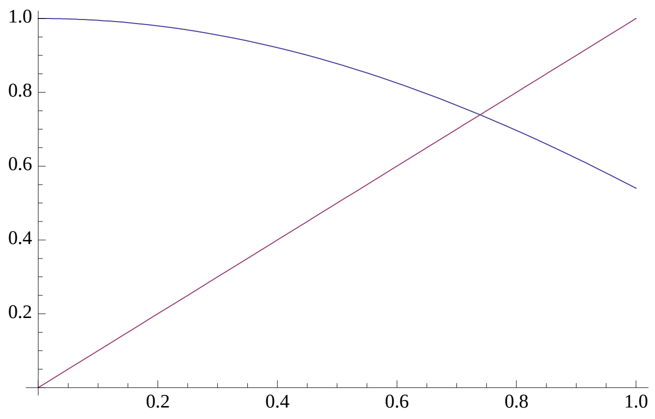
```
NSolve[Cos[x] == x, x]
```

```
Solve::tdep : The equations appear to involve the
   variables to be solved for in an essentially non-algebraic way. >>
```

```
NSolve[Cos[x] == x, x]
```

A simple plot shows that there is a solution in the unit interval.

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



The **FindRoot** command uses Newton's method to find this solution. Since this is an iterative method, an initial guess is required. The graph indicates that 0.8 would be a reasonable initial guess.

```
FindRoot[Cos[x] == x, {x, 0.8}]
```

```
{x → 0.739085}
```

Note that we will use **FindRoot** to solve certain equations for the fractal dimension of a set.

■ Random sequences

Many fractal algorithms have a random element so it is important to be able to generate random sequences. *Mathematica* has several commands for this purpose; **RandomInteger** is, perhaps, the most fundamental. **RandomInteger[{iMin,iMax},n]** generates a list of **n** random integer between **iMin** and **iMax**.

```
RandomInteger[{1, 4}, 10]
```

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

In applications, the terms in a randomly generated sequence need not be numbers and they might not be uniformly weighted. Suppose, for example, that **f1**, **f2**, and **f3** are functions and must be chosen randomly with the weights $1/2$, $1/3$, and $1/6$. This is easily accomplished using the **RandomChoice** command.

Of course, these are not genuinely random sequences but pseudo-random; they are deterministic procedures which appear random. The output of such a pseudo-random number generator depends upon an initial input called a *seed*. We may set the seed using the **SeedRandom** command. **SeedRandom** accepts an integer input.

```
SeedRandom[1];
RandomReal[{1, 4}, 5]
```

```
{3.45217, 1.33426, 3.36858, 1.56341, 1.72408}
```

If the table is generated again after another call to **SeedRandom[1]**, **Random** will generate the same sequence of numbers.

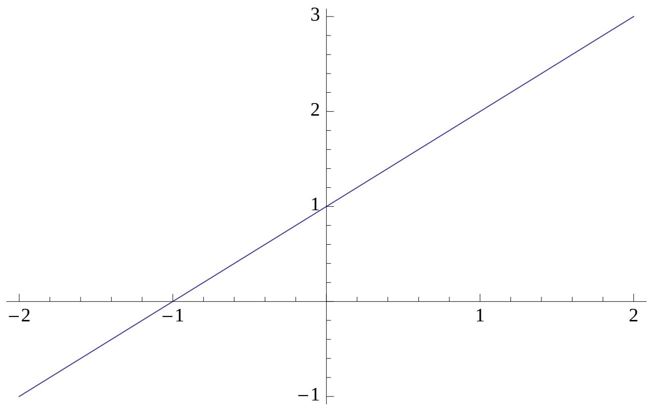
```
SeedRandom[1];
RandomReal[{1, 4}, 5]
```

```
{3.45217, 1.33426, 3.36858, 1.56341, 1.72408}
```

▪ Graphics primitives

Graphics functions like **Plot** generate graphical output.

```
Plot[x + 1, {x, -2, 2}, PlotPoints → 2,
 MaxRecursion → 0]
```



The graph is just one way to format the result. The **InputForm** is much closer to *Mathematica*'s internal representation

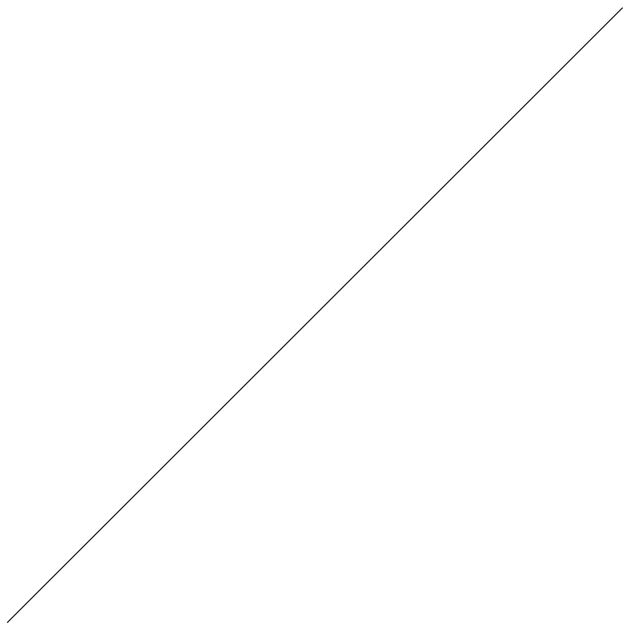
```
% // InputForm
```

```
Graphics[{{{}, {}, {Hue[0.67, 0.6, 0.6],
    Line[{{-1.999996, -0.9999960000000001},
     {1.8472175954999577, 2.8472175954999575},
     {1.999996, 2.9999960000000003}}]}}},
 {AspectRatio -> GoldenRatio^(-1),
  Axes -> True, AxesOrigin -> {0, 0},
  PlotRange -> {{-2, 2}, {-0.9999960000000001,
     2.9999960000000003}},
  PlotRangeClipping -> True,
  PlotRangePadding -> {Scaled[0.02],
    Scaled[0.02]}}]
```

This output is in the form **Graphics[primitives_,options_]**, where **primitives** is a list of graphics primitives and **options** is a list of options. *Graphics primitives* are the building blocks of graphics objects; you can build your own list of graphics primitives and display them using the **Graphics** command. Graphics primitives are important for this book since initial approximations to fractal sets will frequently be expressed in terms of graphics primitives.

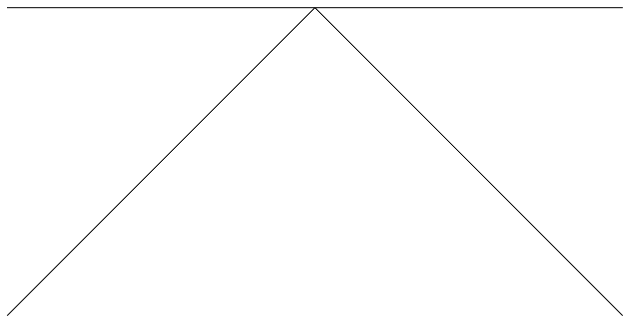
There are more than 10 two-dimensional graphics primitives; perhaps the simplest are **Line** and **Polygon**. **Line** accepts a single list of the form **{{x_1, y_1}, ..., {x_n, y_n}}** to define the line through the points with coordinates {{x_1, y_1}, ..., {x_n, y_n}}. If the points are not collinear, then this defines a polygonal path. If we encase such a line in the **Graphics** command, *Mathematica* will display the line. For example, here is the line from the origin to {1, 1}.

```
Graphics[Line[{{0, 0}, {1, 1}}]]
```



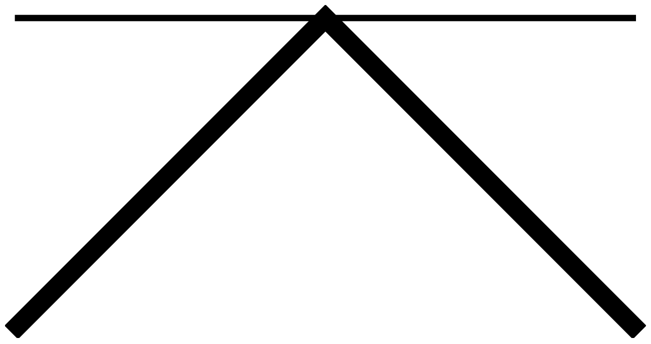
The argument of **Line** may take the more complicated form **{{{**x_1**,** x_2**} …}, …}** to yield multiple lines.

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



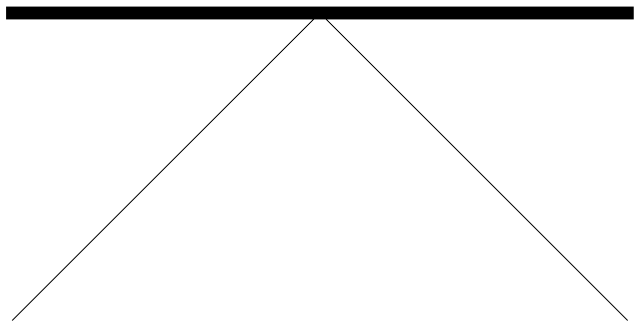
The appearance of a graphics primitive may be affected using a *graphics directive*. For example, the **Thickness** directive affects the thickness of a **Line**. A graphics directive should be placed in the list of graphics primitives and will affect any applicable graphics primitives which follow it until changed by another graphics directive. If we want to use conflicting graphics directives, we need distinct graphics primitives. For example, we can distinguish the two lines above as follows.

```
Graphics[{
  Thickness[.01], Line[{{0, 1}, {2, 1}}],
  Thickness[.03], Line[{{0, 0}, {1, 1}, {2, 0}}]
 }]
```



Nested lists may be used inside the **`Graphics`** command. In this case, a graphics directive does not have any effect outside the list it is in. For example, only the top line is affected by the **`Thickness`** directive the next example.

```
Graphics[{{Thickness[.02], Line[{{0, 1}, {2, 1}}]},
  Line[{{0, 0}, {1, 1}, {2, 0}}]
 }]
```



The **`Polygon`** primitive also accepts a list of points, but represents a filled polygon.

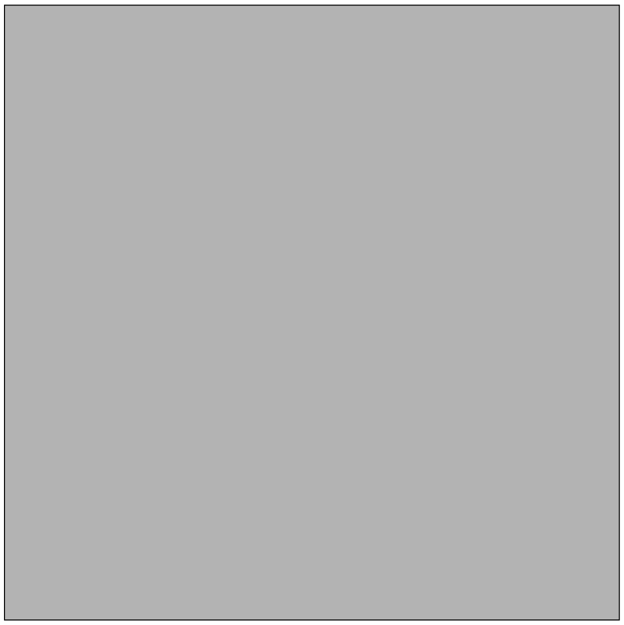
```
Graphics[
 Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}]]
```

Color directives may be used to change the color.

```
Graphics[{GrayLevel[.7],
  Polygon[{{0, 0}, {1, 0}, {1, 1}, {0, 1}}]}]
```

We may combine **`Line`** and **`Polygon`** primitives to include a border.

```
vertices = {{0, 0}, {1, 0}, {1, 1}, {0, 1}, {0, 0}};
Graphics[{
  {GrayLevel[.7], Polygon[vertices]},
  Line[vertices]},
 AspectRatio → Automatic]
```



▪ Manipulating lists

Mathematica can act on lists in many ways. Here is a list of randomly chosen integers.

```
SeedRandom[1];
theList = RandomInteger[{1, 10}, {10}]
```

```
{2, 5, 1, 8, 1, 1, 9, 7, 1, 5}
```

We may sort the list

```
sortedList = Sort[theList]
```

```
{1, 1, 1, 1, 2, 5, 5, 7, 8, 9}
```

We may reverse the order.

```
Reverse[sortedList]
```

```
{9, 8, 7, 5, 5, 2, 1, 1, 1, 1}
```

We may shift the list.

```
RotateLeft[sortedList, 3]
```

```
{1, 2, 5, 5, 7, 8, 9, 1, 1, 1}
```

We may drop some elements from the list

```
Drop[sortedList, 3]
```

```
{1, 2, 5, 5, 7, 8, 9}
```

We may access elements from the list

```
sortedList[[3]]
```

```
1
```

We may partition the list into subparts.

```
Partition[sortedList, 2]
```

```
{{1, 1}, {1, 1}, {2, 5}, {5, 7}, {8, 9}}
```

Nested lists may be flattened to remove the nested structure.

```
Flatten[%]
```

```
{1, 1, 1, 1, 2, 5, 5, 7, 8, 9}
```

We may select data from the list that satisfies some particular criteria. For example, we may select the odd numbers from the list.

```
Select[theList, OddQ]
```

```
{5, 1, 1, 1, 9, 7, 1, 5}
```

The second argument to **`Select`** should be a function returning either `True` or `False`. For example, **`OddQ`** returns `True` if called with an odd integer argument or `False` otherwise.

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

```
{False, True}
```

Pure functions are convenient in this context. For example, **`(#>5)&`** represents a function which returns `True` if called with a numeric argument which is bigger than 5 or `False` if called with a numeric argument which is less or equal to 5. Thus the following returns those elements of the list which are greater than 5.

```
Select[theList, # > 5 &]
```

```
{8, 9, 7}
```

As we've seen, many functions act on the individual elements of a list.

$\texttt{sortedList}^2$

```
{1, 1, 1, 1, 4, 25, 25, 49, 64, 81}
```

In this example, we say that the square function has *mapped* onto the list. A function which automatically maps onto lists is said to be *listable*. The arbitrary function is not listable.

```
Clear[f];
f[theList]
```

```
f[{2, 5, 1, 8, 1, 1, 9, 7, 1, 5}]
```

Any function may be mapped onto a list using the **`Map`** command.

```
Map[f, theList]
```

```
{f[2], f[5], f[1], f[8], f[1], f[1], f[9], f[7], f[1], f[5]}
```

`Map` may be abbreviated using **`/@`**.

```
f /@ theList
```

```
{f[2], f[5], f[1], f[8], f[1], f[1], f[9], f[7], f[1], f[5]}
```

We may map a pure function onto a list. For example, we may use a function which accepts a number and returns a list whose elements are the original number and the original number squared.

```
{#, #^2} & /@ theList
```

```
{{2, 4}, {5, 25}, {1, 1}, {8, 64}, {1, 1}, {1, 1}, {9, 81}, {7, 49}, {1, 1}, {5, 25}}
```

We may view the internal representation of a list using the **FullForm** command.

```
theList // FullForm
```

```
List[2, 5, 1, 8, 1, 1, 9, 7, 1, 5]
```

All non-atomic expressions in *Mathematica* are represented this way. That is all expressions other than numbers or symbols (called atoms) are represented **head[args___]**. We may change the head of an expression using the **Apply** command. For example,

```
Apply[f, theList]
```

```
f[2, 5, 1, 8, 1, 1, 9, 7, 1, 5]
```

As with **Map**, there is an abbreviated form, namely **@@**.

```
f @@ theList
```

```
f[2, 5, 1, 8, 1, 1, 9, 7, 1, 5]
```

We can use this in conjunction with the **Plus** command to add the elements of a list.

```
Plus @@ theList
```

```
40
```

Here is a function which finds the average of a list.

```
average[l_] := Total[l] / Length[l];
average[theList]
```

```
4
```

Some more advanced techniques to apply functions to lists include the commands **Through**, **Inner**, and **Outer**. These commands are used in some of the programs defined in the **FractalGeometry** packages. **Through** is similar to **Map**, but applies each of a list of functions to a single argument, rather than a single function to a list of arguments.

```
Through[{f1, f2, f3}[x]]
```

```
{f1[x], f2[x], f3[x]}
```

Inner and **Outer** are a bit more complicated and will be discussed in appendix **A.2** on linear algebra.

■ Iteration

Iteration is fundamental to both fractal geometry and computer science. If a function f maps a set to itself, then an initial value x_0 may be plugged into the function to obtain the value x_1. This value may then be plugged back into f continuing the process inductively. This yields the sequence $\{x_0, x_1, x_2, \ldots\}$ where $x_n = f(x_{n-1})$ for every integer $n > 0$. Put another way, $x_n = f^n(x_0)$ where f^n represents the n-fold composition of f with itself. The basic commands which perform this operation in *Mathematica* are **`Nest`** and **`NestList`**. **`Nest[f,x0,n]`** returns **`f`**$^{\mathbf{n}}$**`[x0]`**.

```
Clear[f];
Nest[f, x0, 5]
```

```
f[f[f[f[f[x0]]]]]
```

`NestList[f,x,n]` returns the list **`{x`**$_0$**`, x`**$_1$**`, x`**$_2$**`, ...}`**.

```
NestList[f, x0, 5]
```

```
{x0, f[x0], f[f[x0]], f[f[f[x0]]], f[f[f[f[x0]]]], f[f[f[f[f[x0]]]]]}
```

Of course, this will be more interesting if we use a specific function and a specific starting point. For example, iteration of **`Cos`** starting at 0.8, should yield a sequence which converges to the fixed point of **`Cos`**. Note that we type **`Cos`** instead of **`Cos[x]`** since **`NestList`** anticipates a function.

```
NestList[Cos, .8, 30]
```

```
{0.8, 0.696707, 0.76696, 0.720024, 0.75179, 0.730468, 0.744863,
 0.735181, 0.741709, 0.737315, 0.740276, 0.738282, 0.739626, 0.738721, 0.73933,
 0.73892, 0.739196, 0.73901, 0.739136, 0.739051, 0.739108, 0.73907, 0.739096,
 0.739078, 0.73909, 0.739082, 0.739087, 0.739084, 0.739086, 0.739084, 0.739086}
```

Note that the command **`FixedPoint[f,x0]`** automatically iterates **`f`** starting at **`x0`** until the result no longer changes, according to *Mathematica*'s internal representation.

```
FixedPoint[Cos, .8]
```

```
0.739085
```

■ Pattern matching

When a function is declared, internally *Mathematica* adds a *pattern* to its global rule base. As we have seen, here is how to define $f(x) = x^2$.

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

```
f[2]
```

```
4
```

If we have an algebraic expression, rather than a function, we may do this manually using a *rule*.

```
x^2 /. x → 2
```

```
4
```

In this example, the expression is **`x`**$^{\mathbf{2}}$, the rule is **`x→2`** and **`/.`** is the replacement operator which tells *Mathematica* to implement the rule. The arrow → may also be typed **`->`** which *Mathematica* automatically converts to → in an input cell. An alternative form for **`x→`**2 is **`Rule[x,2]`**.

Note that the commands for solving algebraic formulae return lists of rules. This makes it easy to plug solutions back into expressions. For example, suppose we use **Solve** to find the critical points of a polynomial.

```
poly = x^3 + 4 x^2 - 3 x + 1;
cps = Solve[D[poly, x] == 0, x]
```

$\left\{\{x \to -3\}, \left\{x \to \frac{1}{3}\right\}\right\}$

We can plug the critical points back into **poly**.

```
poly /. cps
```

$\left\{19, \frac{13}{27}\right\}$

An underscore is used to denote a more general pattern. For example, suppose we want to square the *integers* in a list.

```
{1, 2, a, b, π} /. {1 → 1^2, 2 → 2^2}
```

```
{1, 4, a, b, π}
```

We have used a list of rules - one for each integer in the list. We would prefer a pattern which matches each integer. One way to do this is as follows.

```
{1, 2, a, b, π} /. n_Integer → n^2
```

```
{1, 4, a, b, π}
```

Any functions to the right of the **Rule** operator → are evaluated immediately. There is a delayed version of **Rule** called **RuleDelayed** which waits until the rule is applied to evaluate any functions. **RuleDelayed** may be abbreviated by :→ which may be typed as **:>**. The difference between → and :→ is similar to the difference between **=** and **:=**. Suppose, for example, we want to expand any powers in an expressions. We might try the following.

```
(a + b)^(1/2) (c + d)^3 /. x_^n_ → Expand[x^n]
```

$\sqrt{a+b}\ (c+d)^3$

This doesn't work since the **Expand[x^n]** is evaluated immediately to yield **x^n.** Thus this rule is equivalent to **x_^n_→x^n** which does nothing. By contrast, the rule delayed version **x_^n_ :→ x^n** waits until any substitutions occur to perform the expansion.

```
(a + b)^(1/2) (c + d)^3 /. x_^n_ :→ Expand[x^n]
```

$\sqrt{a+b}\ \left(c^3 + 3 c^2 d + 3 c d^2 + d^3\right)$

Pattern matching is a powerful and important yet subtle aspect of *Mathematica*. An excellent discussion of pattern matching appears in [Bl, Wi].

■ Programming

Mathematica has a full featured, high-level programming language including loop constructs such as **Do** and **While**, flow control devices such as **If** and **Switch**, and scoping constructs which allow you to declare local variables. Programming with *Mathematica* is a deep subject and entire books, such as [Mae], have been written about it. In this appendix, we'll develop one short program which illustrates some important themes for this book. Of course, the book contains many more such examples.

A word of warning is in order to experienced programmers. The procedural programming paradigm used for many compiled languages, such as C, C++, or Java, is emphatically not appropriate for a high-level language such as *Mathematica*. This book primarily follows a paradigm known as functional programming and, to a lesser extent, rule based programming. The idea is to use *Mathematica*'s vast set of tools to define functions which act naturally on *Mathematica*'s rich set of data structures. The "natural" way to do something in any language is determined by the internal structure of the language; it takes experience to learn what the most appropriate technique for a given situation might be. Consider, for example, the following procedural approach to generating a list of squares of the first 10 squares.

```
squareList = {};
For[i = 1, i ≤ 10, i++,
 squareList = AppendTo[squareList, i^2]]
squareList
```

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

Most people with procedural programming experience should be comfortable with this example. If you are not, don't worry because this is not the way to do it anyway. It is much more natural in *Mathematica* to do the following.

```
Range[10]^2
```

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

The functional approach is much shorter and more readable. More importantly, the functional approach generally executes faster. Suppose we want to generate a list of 20,000 squares and measure how long it takes. Here is a procedural approach.

```
squareList = {};
For[i = 0, i ≤ 20 000, i++,
   squareList = AppendTo[squareList, i^2]]; // Timing
```

```
{8.33079, Null}
```

Ouch!   A big part of the problem is that **AppendTo** is a very expensive operation, particularly when applied to long lists. This may be sped up considerably by generating an initial list and modifying it in place as follows.

```
squareList = Range[20 000];
For[i = 0, i ≤ 20 000, i++,
   squareList[[i]] = i^2]; // Timing
```

```
{0.099322, Null}
```

The functional approach is still considerably faster.

```
Range[20 000]^2; // Timing
```

```
{0.000853, Null}
```

Linear Algebra

Vectors are represented as lists of the appropriate length in *Mathematica*. Thus a two dimensional vector is simply a list of length two. Matrices are represented as lists of lists. A two dimensional matrix should be a list of length two and each of its elements should be a list of length two; the first element should represent the first row and the second element should represent the second row. You can pass a matrix to the **`MatrixForm`** command to see it typeset as a matrix.

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

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

You can also enter matrices directly using the $\begin{pmatrix} \square & \square \\ \square & \square \end{pmatrix}$ button from the BasicInputs palette or the using **Table/Matrix** command from the **Insert** menu.

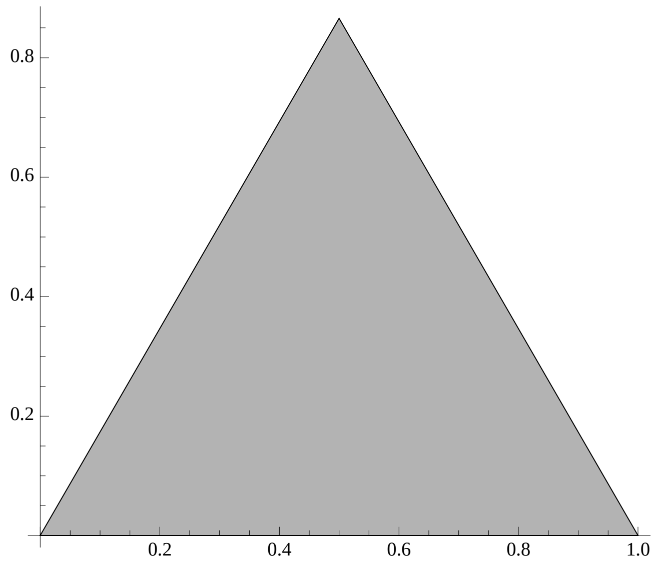
Matrix multiplication is represented using a **`.`** between two matrices or between a matrix and a vector.

```
{{a, b}, {c, d}}.{x_1, x_2}
```

$$\{a\, x_1 + b\, x_2,\ c\, x_1 + d\, x_2\}$$

We can now explore the effect of multiplication by certain types of matrices on the general two dimensional vector. First we create a simple picture to act on.

```
triangle = Graphics[{{GrayLevel[.7],
    Polygon[{{0, 0}, {1, 0},
      {1 / 2, √3 / 2}}]},
   {Line[{{0, 0}, {1, 0},
      {1 / 2, √3 / 2}, {0, 0}}]}},
  Axes → True]
```

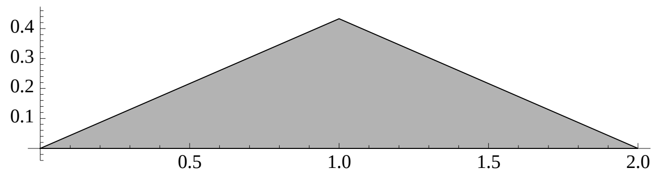


The following simple function accepts a **`Graphics`** object and a matrix and applies the linear transformation defined by the matrix to all two-dimensional numerical vectors in the **`Graphics`** object.

```
transform[Graphics[g_, opts___], M_?MatrixQ] :=
  Graphics[GeometricTransformation[g,
    AffineTransform[M]], opts];
```

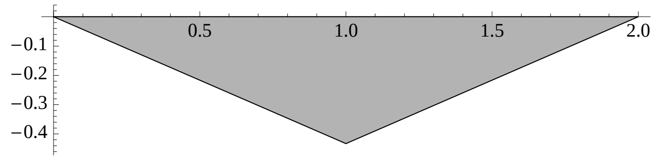
Now, a diagonal matrix $\begin{pmatrix} a & 0 \\ 0 & d \end{pmatrix}$ should stretch or compress the graphic by the factors *a* in the horizontal direction and *d* in the vertical direction.

```
M = {{2, 0}, {0, 1 / 2}};
transform[triangle, M]
```



We can introduce a reflection by placing a minus sign in the correct place.

```
M = {{2, 0}, {0, -1 / 2}};
transform[triangle, M]
```



Rotation is a bit trickier. Rotation about the origin through the angle θ may be represented by the matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

This is exactly the matrix encode in the *Mathematica* function **`RotationMatrix`**.

```
RotationMatrix[θ] // MatrixForm
```

```
( Cos[θ]  -Sin[θ] )
( Sin[θ]  Cos[θ]  )
```

In fact, there is a **`RotationTransform`** command analgous to the **`AffineTransform`** command we used above, but it defeats the purpose of our linear algebra review.

Now, positive θ implies counter-clockwise rotation, while negative θ implies clockwise rotation. To see why this matrix works, first apply it to the standard basis vectors $\vec{\imath} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $\vec{\jmath} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. This should simply extract the two columns of the rotation matrix. We first consider the matrix multiplied by $\vec{\imath}$.

```
{{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}}.{1, 0}
```

```
{Cos[θ], Sin[θ]}
```

By basic trigonometry, this is the basis vector $\vec{\imath}$ rotated through the angle θ. We next look at the matrix multiplied by $\vec{\jmath}$.

```
{{Cos[θ], -Sin[θ]}, {Sin[θ], Cos[θ]}}.{0, 1}
```

```
{-Sin[θ], Cos[θ]}
```

Now the basis vector $\vec{\jmath}$ may be written

$$\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\cos\pi/2\\ \sin\pi/2\end{pmatrix}.$$

Thus $\vec{j}$ rotated by the angle theta may be written

$$\begin{pmatrix}\cos(\theta+\pi/2)\\ \sin(\theta+\pi/2)\end{pmatrix} = \begin{pmatrix}-\sin\theta\\ \cos\theta\end{pmatrix},$$

which exactly the rotation matrix multiplied by $\vec{j}$. This last equality can be seen by applying basic trigonometric identities using **TrigExpand**.

```
{Cos[θ + π / 2], Sin[θ + π / 2]} // TrigExpand
```

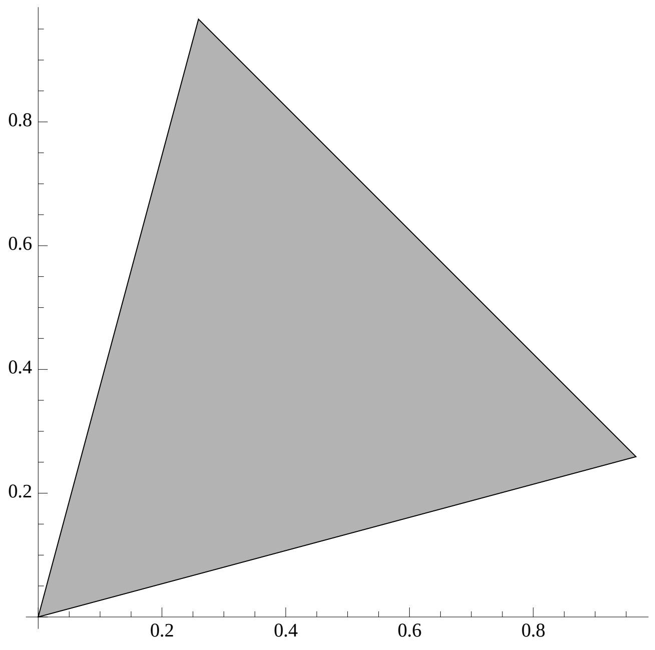
```
{-Sin[θ], Cos[θ]}
```

Now since M rotates both $\vec{i}$ and $\vec{j}$ by the angle θ, it rotates any vector through that angle by linearity since

$$M\begin{pmatrix}x_1\\x_2\end{pmatrix} = M\left(x_1\begin{pmatrix}1\\0\end{pmatrix} + x_2\begin{pmatrix}0\\1\end{pmatrix}\right) = x_1 M\begin{pmatrix}1\\0\end{pmatrix} + x_2 M\begin{pmatrix}0\\1\end{pmatrix}.$$

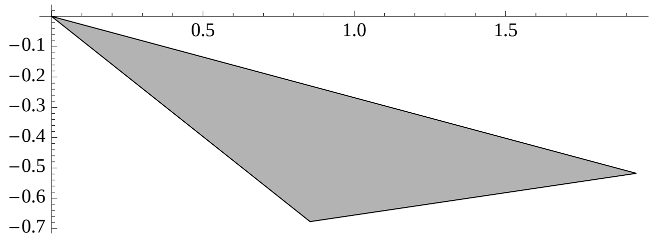
For example, we can rotate our triangle rotated through the angle $\pi/12$.

```
M = {{Cos[π / 12], -Sin[π / 12]},
   {Sin[π / 12], Cos[π / 12]}};
Show[transform[triangle, M],
 Axes → True, AspectRatio → Automatic]
```



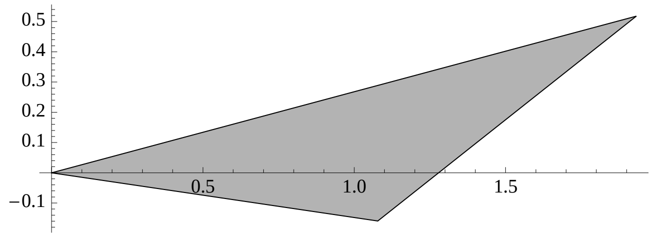
We may combine these geometric transformations by multiplying the appropriate matrices.

```
compression = {{2, 0}, {0, 1 / 2}};
reflection = {{1, 0}, {0, -1}};
rotation = {{Cos[π / 12], -Sin[π / 12]},
   {Sin[π / 12], Cos[π / 12]}};
M = reflection.rotation.compression;
transform[triangle, M]
```



These operations are not generally commutative.

```
M = rotation.reflection.compression;
transform[triangle, M]
```



The set of eigenvectors and eigenvalues forms a more advanced way to understand the geometric behavior of certain matrices. We will use eigenvalues to compute the fractal dimension of digraph self-similar sets. A vector v is an *eigenvector* of a matrix M if there is a number λ so that $M v = \lambda v$. In this case, λ is the corresponding *eigenvalue*. The collection of eigenvalues and eigenvectors is sometimes called the *eigensystem* of the matrix. The eigenvector determines a one dimensional subspace which is invariant under M and the eigenvalue determines the factor by which M expands or compresses this subspace. *Mathematica* has built in commands **`Eigenvalues`**, **`Eigenvectors`**, and **`Eigensystem`** to compute these.

```
M = {{1 / 2, 1}, {0, 2}};
Eigenvalues[M]
```

$$\left\{2, \frac{1}{2}\right\}$$

```
Eigenvectors[M]
```

$$\left\{\left\{\frac{2}{3}, 1\right\}, \{1, 0\}\right\}$$

Thus this matrix should compress the vector $\begin{pmatrix}1\\0\end{pmatrix}$ by the factor $1/2$ and it should expand the vector $\begin{pmatrix}2/3\\1\end{pmatrix}$ by the factor 2.

```
M.{1, 0}
```

$$\left\{\frac{1}{2}, 0\right\}$$

```
M.{2 / 3, 1}
```

$$\left\{\frac{4}{3}, 2\right\}$$

The command **Eigensystem** returns a list of two lists; the first list contains the eigenvalues and the second list contains the corresponding eigenvectors.

```
Eigensystem[M]
```

$$\left\{\left\{2, \frac{1}{2}\right\}, \left\{\left\{\frac{2}{3}, 1\right\}, \{1, 0\}\right\}\right\}$$

Finally, the commands **Inner** and **Outer** generalize the dot and matrix products of vectors. These commands will be useful when implementing the digraph iterated function system scheme. The dot product of two vectors simply multiplies termwise and adds the results. The dot product of *w* and *v* is denoted **w.v**. Vectors are represented as lists. For example,

```
w = {1, 2, 3};
v = {a, b, c};
w.v
```

```
a + 2 b + 3 c
```

The inner product generalizes this by allowing operations other than addition or multiplication to be used. In general, operations may be defined by multi-variate functions **f** and **g** where **f** plays the role of multiplication and **g** plays the role of addition. **Inner** implements this generalized inner product as follows.

```
Clear[f, g];
Inner[f, {1, 2, 3}, {a, b, c}, g]
```

```
g[f[1, a], f[2, b], f[3, c]]
```

Note that if **f** is **Times** denoting multiplication and **g** is **Plus** denoting addition, then we recover the basic dot product.

```
Inner[Times, {1, 2, 3}, {a, b, c}, Plus]
```

```
a + 2 b + 3 c
```

One way to think of the inner product is as matrix multiplication of a row vector times a column vector.

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.\begin{pmatrix} a \\ b \\ c \end{pmatrix} \text{ // MatrixForm}$$

$$\begin{pmatrix} a + 2 b + 3 c \end{pmatrix}$$

If we multiply a column vector by a row vector, we obtain the outer product. Note that the lengths of the vectors need not be the same.

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}.\begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix} \text{ // MatrixForm}$$

$$\begin{pmatrix} a & 2a & 3a & 4a \\ b & 2b & 3b & 4b \\ c & 2c & 3c & 4c \end{pmatrix}$$

This may be generalized using the command **Outer**

```
Outer[f, {a, b, c}, {1, 2, 3, 4}] // MatrixForm
```

$$\begin{pmatrix} f[a, 1] & f[a, 2] & f[a, 3] & f[a, 4] \\ f[b, 1] & f[b, 2] & f[b, 3] & f[b, 4] \\ f[c, 1] & f[c, 2] & f[c, 3] & f[c, 4] \end{pmatrix}$$

If we use **Times** in place of **f**, we recover the usual outer product.

```
Outer[Times, {a, b, c}, {1, 2, 3, 4}] // MatrixForm
```

$$\begin{pmatrix} a & 2a & 3a & 4a \\ b & 2b & 3b & 4b \\ c & 2c & 3c & 4c \end{pmatrix}$$