Introduction II

"Essential Mathematica for Students of Science",James J. Kelly , 2006

http://www.physics.umd.edu/courses/CourseWare

"*Mathematica 4.1* Notebooks -Complimentary software to accompany our textbook",John H. Mathews, and Russell W. Howell, 2002

Partial Derivatives

Here is a good way to do partial derivatives:

```
Derivative[1, 0][f][t, y]
```

$$f^{(1,0)}[t, y]$$

```
Derivative[0, 1][f][t, y]
```

$$f^{(0,1)}[t, y]$$

```
f[t_, x1_, x2_, x3_] =
 Exp[Cos[x1 * Sqrt[x2 - x3]]] * BesselJ[0, t]
```

$$e^{\text{Cos}\left[\text{x1}\sqrt{\text{x2}-\text{x3}}\right]}\ \text{BesselJ}[0, t]$$

```
Derivative[0, 0, 0, 1][f][t, x1, x2, x3]
```

$$\frac{e^{\text{Cos}\left[x1\sqrt{x2-x3}\right]}\, x1\, \text{BesselJ}[0, t]\, \text{Sin}\left[x1\sqrt{x2-x3}\right]}{2\sqrt{x2-x3}}$$

```
Derivative[1, 1, 1, 1][f][t, x, y, z]
```

$$-\frac{e^{\text{Cos}\left[x\sqrt{y-z}\right]}\, x\, \text{BesselJ}[1, t]\, \text{Cos}\left[x\sqrt{y-z}\right]}{4\,(y-z)} +$$
$$\frac{e^{\text{Cos}\left[x\sqrt{y-z}\right]}\, \text{BesselJ}[1, t]\, \text{Sin}\left[x\sqrt{y-z}\right]}{4\,(y-z)^{3/2}} +$$
$$\frac{e^{\text{Cos}\left[x\sqrt{y-z}\right]}\, x^2\, \text{BesselJ}[1, t]\, \text{Sin}\left[x\sqrt{y-z}\right]}{4\sqrt{y-z}} +$$
$$\frac{1}{4\sqrt{y-z}}3\, e^{\text{Cos}\left[x\sqrt{y-z}\right]}\, x^2\, \text{BesselJ}[1, t]$$
$$\text{Cos}\left[x\sqrt{y-z}\right]\, \text{Sin}\left[x\sqrt{y-z}\right] +$$
$$\frac{e^{\text{Cos}\left[x\sqrt{y-z}\right]}\, x\, \text{BesselJ}[1, t]\, \text{Sin}\left[x\sqrt{y-z}\right]^2}{4\,(y-z)} -$$
$$\frac{e^{\text{Cos}\left[x\sqrt{y-z}\right]}\, x^2\, \text{BesselJ}[1, t]\, \text{Sin}\left[x\sqrt{y-z}\right]^3}{4\sqrt{y-z}}$$

Direction and Vector Fields

```
<< "VectorFieldPlots`"
```

Here we define a direction field:

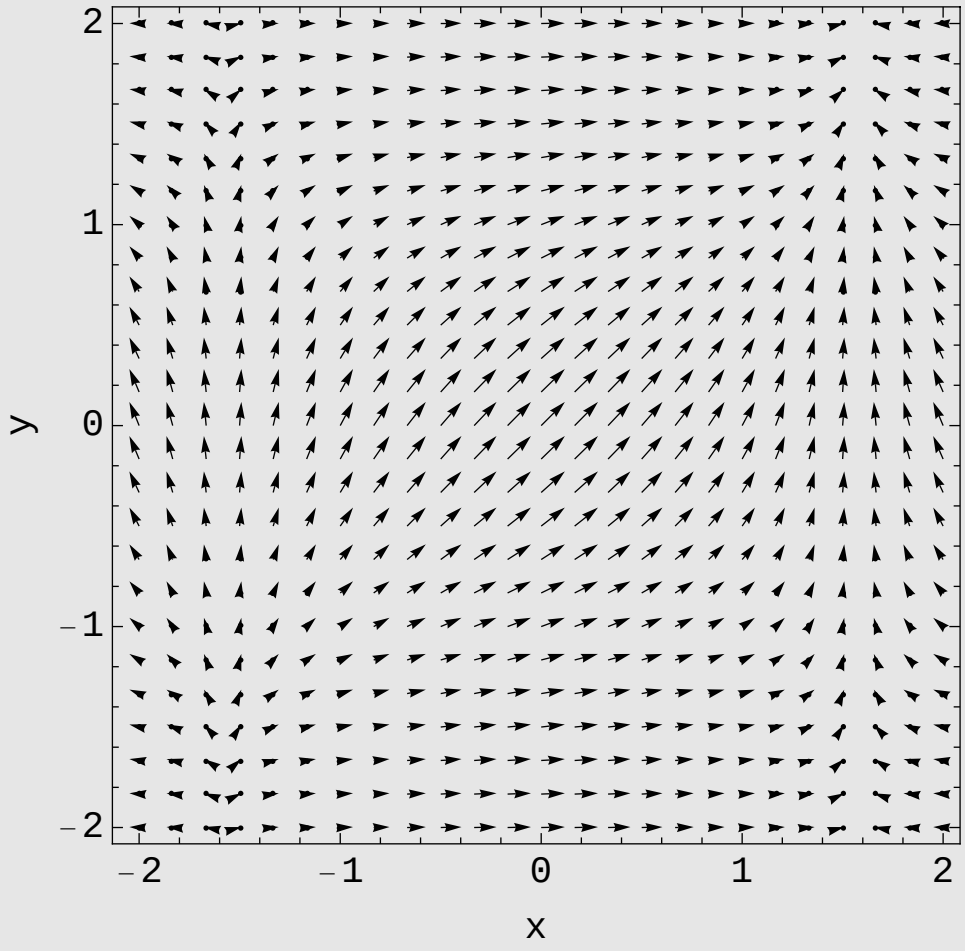
```
L[y_] := D[y, {x, 1}] == Cos[x] / Exp[-y^2]
```

```
L[y[x]]
```

```
y'[x] == e^y[x]^2 Cos[x]
```

The direction field plots small arrows with slope give by $< \Delta x, \Delta y > \sim < dx, dy >$.

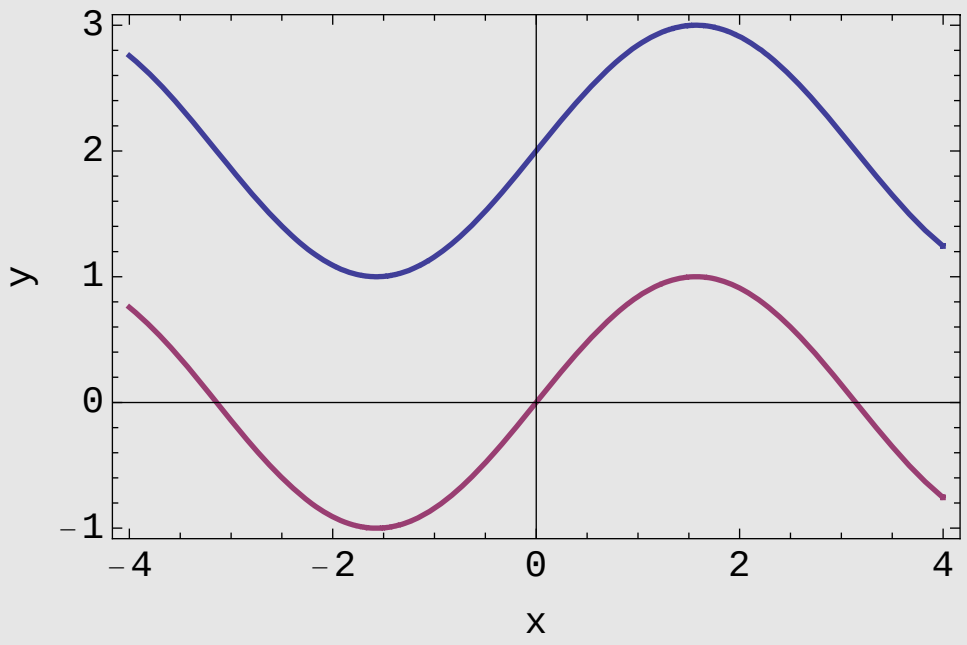
```
fieldeqn = VectorFieldPlot[{Cos[x], Exp[-y^2]},
    {x, -2, 2}, {y, -2, 2}, ImageSize → {500, 500},
    PlotPoints → 25];
Show[fieldeqn, Frame → True, FrameLabel → {"x", "y"}]
```



Plotting

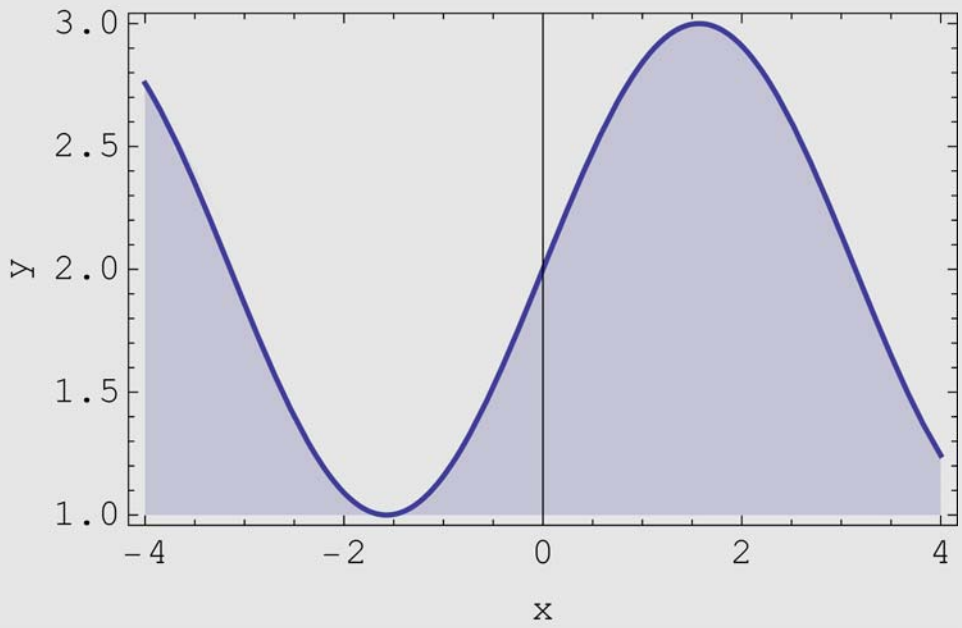
Plot

```
plot1 = Plot[{Sin[x] + 2, Sin[x]}, {x, -4, 4},
  Frame → True, FrameLabel → {"x", "y"},
  PlotStyle → Thick]
```

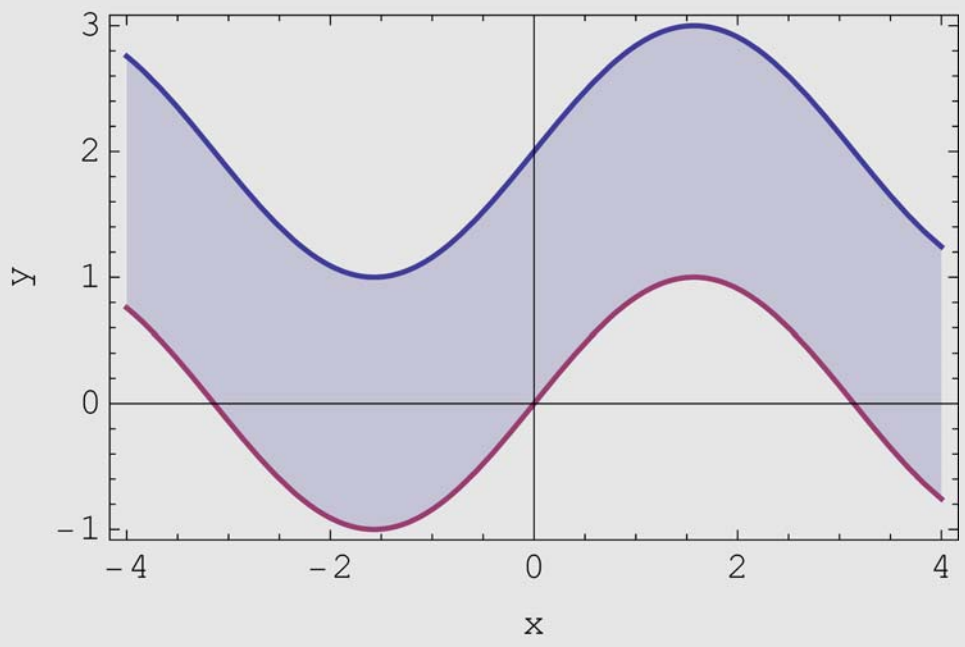


Filled Plots

```
plot1 = Plot[{Sin[x] + 2}, {x, -4, 4}, Frame → True,
  FrameLabel → {"x", "y"}, PlotStyle → Thick,
  Filling → Axis]
```

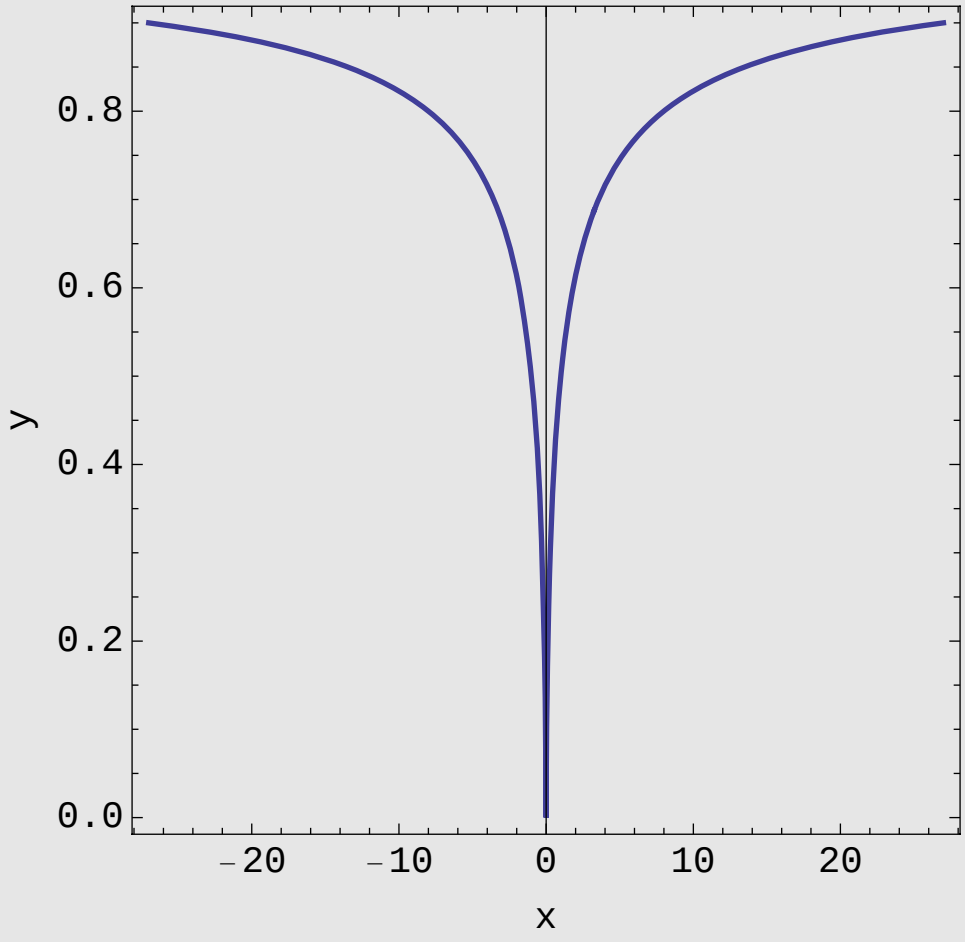


```
plot1 = Plot[{Sin[x] + 2, Sin[x]}, {x, -4, 4},
  Frame → True, FrameLabel → {"x", "y"},
  PlotStyle → Thick, Filling → {1}]
```



ParametericPlot

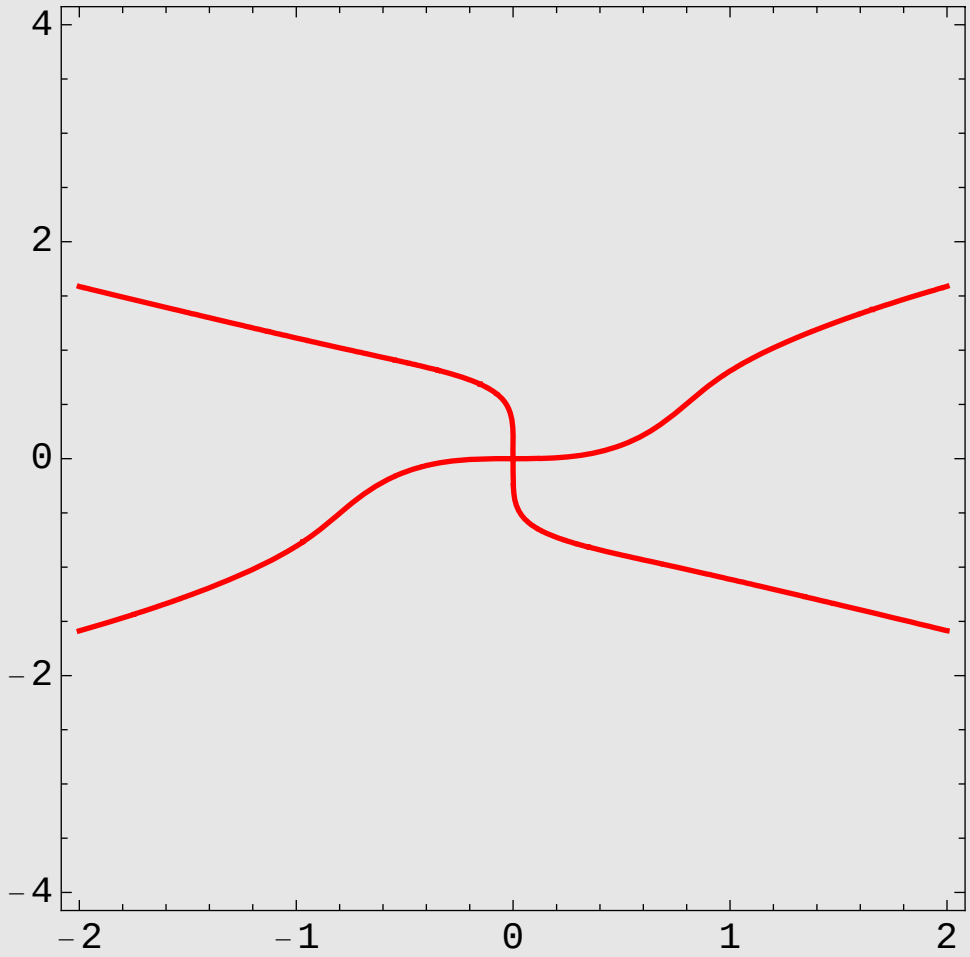
```
ParametricPlot[{t^3, t^2 / (1 + t^2)}, {t, -3, 3},
 Frame → True, FrameLabel → {"x", "y"},
 AspectRatio → 1, PlotStyle → Thick]
```



Plots of Implicit Functions

Mathematica plots implicit functions using the ContourPlot command:

```
ContourPlot[x^4 - y^6 == Sin[x * y], {x, -2, 2},
 {y, -4, 4}, PlotPoints → 100,
 ContourStyle → {Red, Thick}]
```

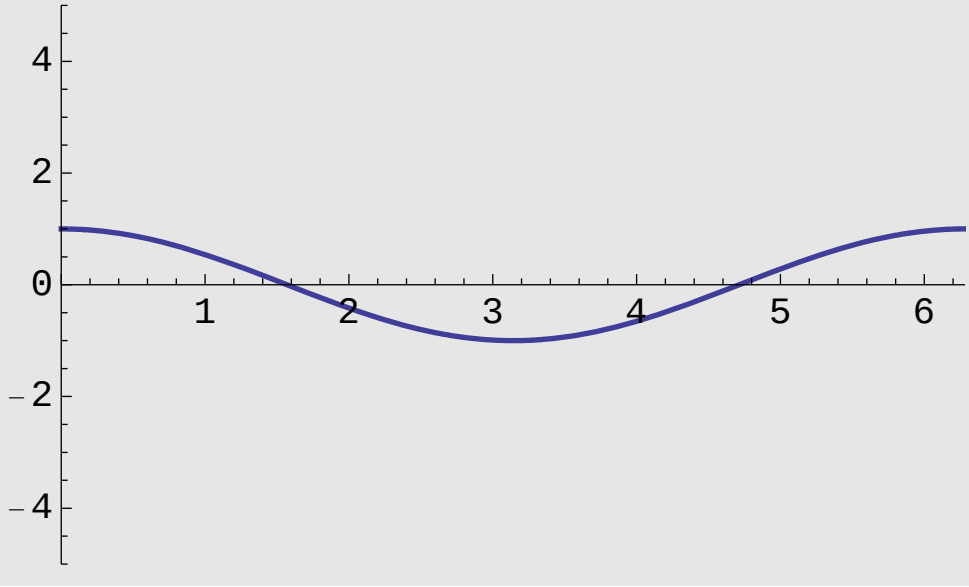


Manipulate

Basic Example

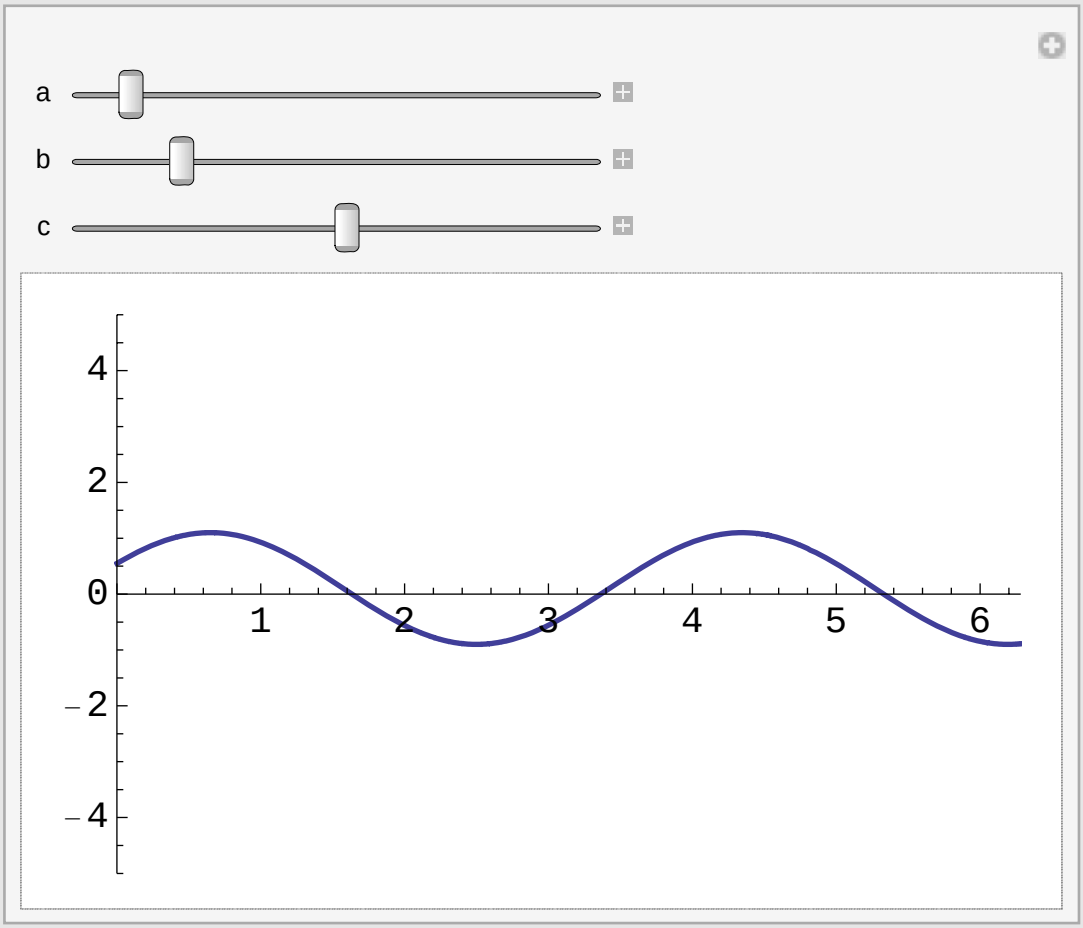
Manipulate is a wonderful command that produces output as a function of a parameter. It can be used to create animations, among other things.

```
Plot[Cos[x], {x, 0, 2 Pi}, PlotStyle → Thick,
 PlotRange → {{0, 2 Pi}, {-5, 5}}]
```



We can investigate the behaviour of the function $f(x) = \cos(a\,x + b) + c$ using Manipulate.

```
Manipulate[Plot[Cos[a * x + b] + c, {x, 0, 2 Pi},
  PlotStyle → Thick, PlotRange → {{0, 2 Pi}, {-5, 5}}],
 {a, -2, 2}, {b, 0, 2 Pi}, {c, -2, 2}]
```



If you click on the plus sign beside each slider, you can see the value of the parameter that the plot is for. You can use the slider to change the value of the parameter, and even create animations! Try it! Obvioulsy, too many parameters still becomes a bit much if you are doing animations, but certainly this can help you explore the behaviour of fucntions relative to a parameter.

Example from Differential Equations

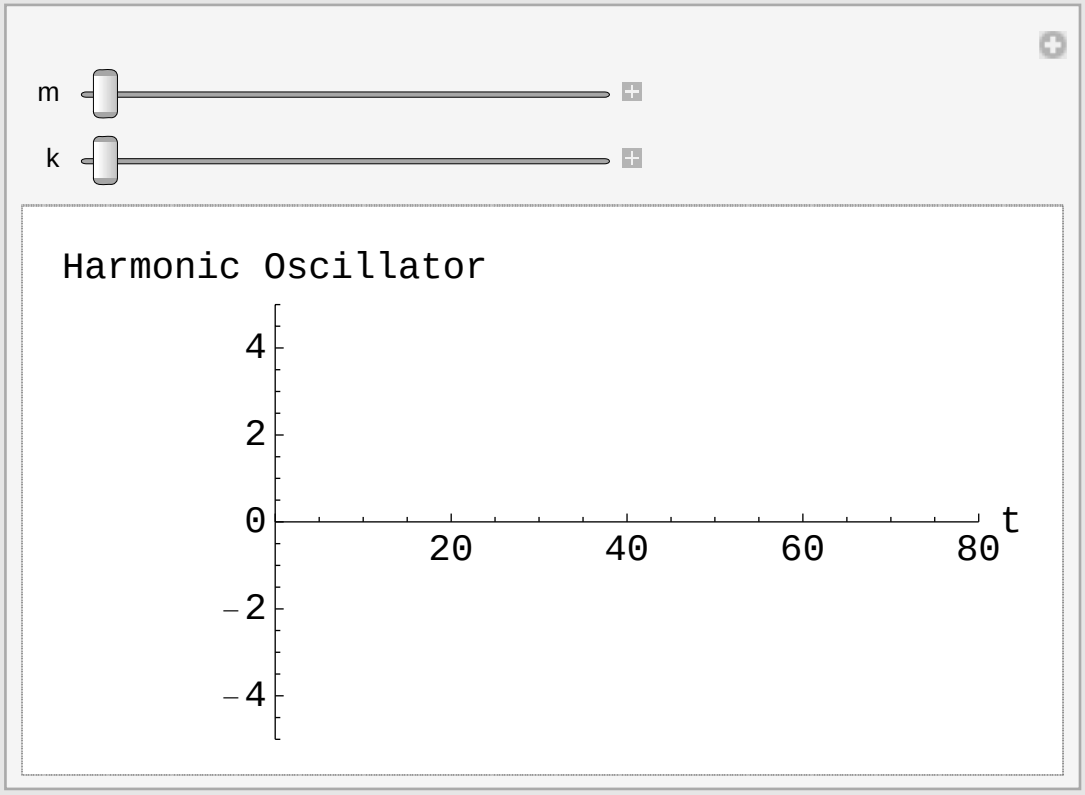
For example, here is a harmonic oscillator (starting at rest from the origin) as a function of mass (*m*) and spring constant (*k*) with a sinusoidal forcing function.

```
Clear[x, displacement]
sol =
  DSolve[{m x''[t] + k x[t] == Sin[t] , x[0] == 0,
    x'[0] == 0}, x[t], t];
displacement[t_, k_, m_] = x[t] /. Last[sol]
```

$$\frac{1}{\sqrt{k}\ (k-m)}\left(\sqrt{k}\ \text{Cos}\left[\frac{\sqrt{k}\ t}{\sqrt{m}}\right]^2 \text{Sin}[t] - \sqrt{m}\ \text{Sin}\left[\frac{\sqrt{k}\ t}{\sqrt{m}}\right] + \sqrt{k}\ \text{Sin}[t]\ \text{Sin}\left[\frac{\sqrt{k}\ t}{\sqrt{m}}\right]^2\right)$$

We can easily examine the baviour of the solution for varying mass and spring constant.

```
Manipulate[Plot[displacement[t, k, m], {t, 0, 80},
  PlotRange → {{0, 80}, {-5, 5}}, PlotStyle → Thick,
  AxesLabel → {"t", "Harmonic Oscillator"}],
 {m, 1, 5}, {k, 2, 5}]
```

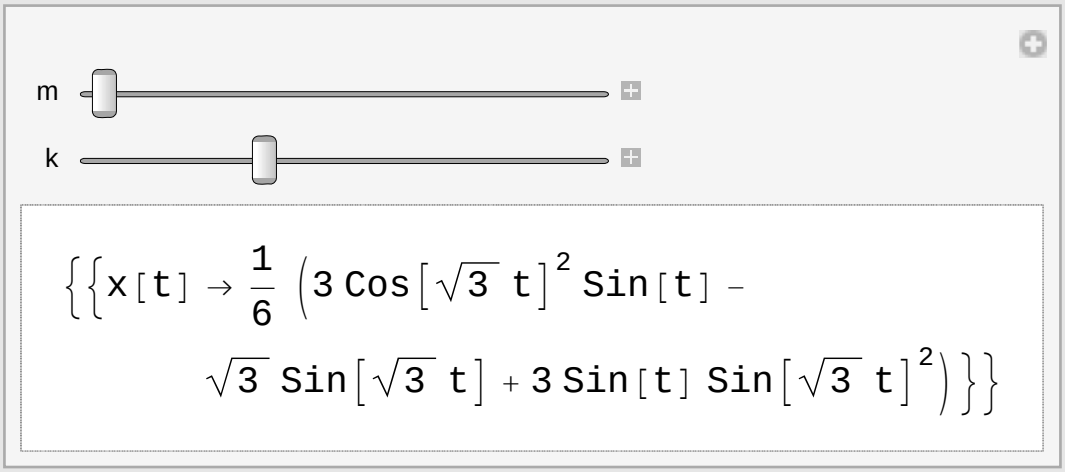


Notice that there are some value of *k* and *m* for which there is no plot. That is because the solution Mathematica found to the differential equation is not correct for those values of *k* and *m*. This is where *k* = *m*, and the solution Mathematica found is only valid where *k* ≠ *m*.

We can actually use Manipulate to invesitgate this further (notice I have now told Manipulate to vary the *m* and *k* by an integer amoun belwt). Notice that the form of the solution changes whenever *k* = *m* (you get a *t* factor in some of the terms, instead of pure trig

solutions). Obvioulsy, we study differential equations to help us understand why this is happening.

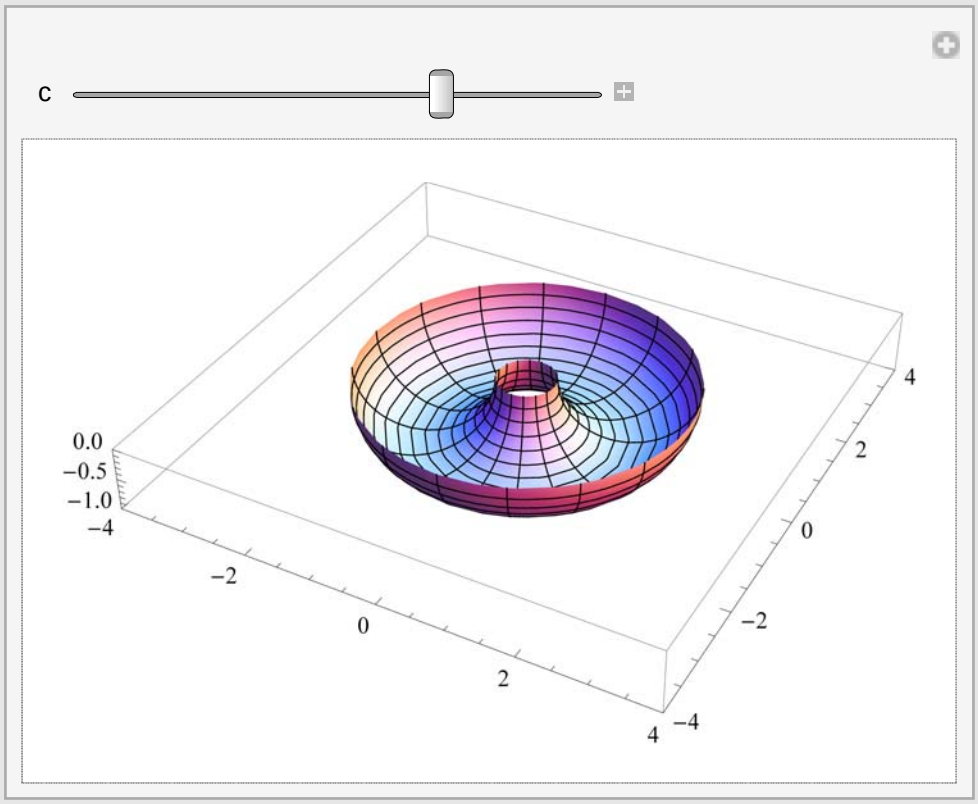
```
Clear[x]
Manipulate[
 DSolve[{m x''[t] + k x[t] == Sin[t] , x[0] == 0,
   x'[0] == 0}, x[t], t], {m, 1, 5, 1}, {k, 2, 5, 1}]
```



Example from Calculus III

The three types of torus--with the top cut off so you can see them.

```
Manipulate[ParametricPlot3D[
  {(c + 1 * Cos[t]) Cos[s], (c + 1 * Cos[t]) Sin[s],
   1 * Sin[t]}, {t, Pi, 2 Pi}, {s, 0, 2 Pi},
  PlotRange → {{-4, 4}, {-4, 4}, {-1, 0}}],
 {c, 0, 2}]
```



Recursion

If you have used Mathematica before, you have probably encountered recursion. Recursion is easily set up in Mathematica, but you need to be careful or you can run into difficulties.

Let's say we want to numerically calculate a root of $f(x) = 2\,x\cos(2\,x) - (x-2)^2$ that is near $x = 2$ using Newtons method. We can do that recursively using the following:

```
f[x_] = 2 x Cos[2 x] - (x - Pi)^2;
x[0] = 2.0;
x[i_] := x[i - 1] - f[x[i - 1]] / f'[x[i - 1]]

data = Table[{i, x[i]}, {i, 0, 10}];
TableForm[data]
```

```
0   2.
1   2.55727
2   2.41117
3   2.41156
4   2.41156
5   2.41156
6   2.41156
7   2.41156
8   2.41156
9   2.41156
10  2.41156
```

This worked, but it is not a very good implementation of Newton's method. You should notice that it is somewhat slow, and that it only produced 5 decimals of output. We can get more output using `SetPrecision` , and speed it up by storing the intermediate results so Mathematica doesn't have to calculate them again and again.

```
f[x_] = 2 x Cos[2 x] - (x - Pi)^2;
x[0] = 2.0;
x[i_] :=
 x[i] = SetPrecision[
   x[i - 1] - f[x[i - 1]] / f'[x[i - 1]], 20]

data = Table[{i, x[i]}, {i, 0, 10}];
TableForm[data]
```

```
0   2.
1   2.5572732634040038491
2   2.4111660464448099118
3   2.4115573115702641479
4   2.4115572855540860553
5   2.4115572855540859406
6   2.4115572855540859406
7   2.4115572855540859406
8   2.4115572855540859406
9   2.4115572855540859406
10  2.4115572855540859406
```

There is one more important change to make when we are doing recursion--we want to clear the variables before each implementation. We can do that by adding a Clear command at the beginning of the cell. If we don't do this, it is possible that if you come back and re-evaluate the cell you will get errors due to some of the quantities being previously defined. Worse than getting errors, you can sometimes get no errors but get incorrect results--this will happen if you change the function *f*!

```
Clear[f, x, data]
f[x_] = 2 x Cos[2 x] - (x - Pi)^2;
x[0] = 2.0;
x[i_] :=
 x[i] = SetPrecision[x[i - 1] - f[x[i - 1]] / f'[x[i - 1]],
   20]

data = Table[{i, x[i]}, {i, 0, 10}];
TableForm[data]
```

```
0  2.
1  2.5572732634040038491
2  2.4111660464448099118
3  2.4115573115702641479
4  2.4115572855540860553
5  2.4115572855540859406
6  2.4115572855540859406
7  2.4115572855540859406
8  2.4115572855540859406
9  2.4115572855540859406
10 2.4115572855540859406
```

Products and Sums

Mathematica can do some interesting things with products and sums. This is particularly useful when solving differential equations using infinite series, where you need to figure out the product form from the pattern, which can then be converted into a known function using Mathematica.

For example, let's say we have the function given by an infinte series with coefficients given by a set of recursion relations:

$$f(x) = \sum_{n=0}^{\infty} a_n x^n$$

$$a_0 = \pi$$

$$a_n = \frac{1 \times 2 \times 3 \times \ldots \times n}{1 \times 3 \times 5 \times 7 \times \ldots \times (2n-1)} a_0$$

Here's how we can figure out a simpler form for the function that does not involve recursion or summation. First, we use the Product command to determine a_n:

```
a[n_] = Product[i / (2 i - 1), {i, 1, n}] * Pi
```

$$\frac{256\,\pi}{88\,179}$$

Now, we use this value of a_n in the infinite sum for $f(x)$:

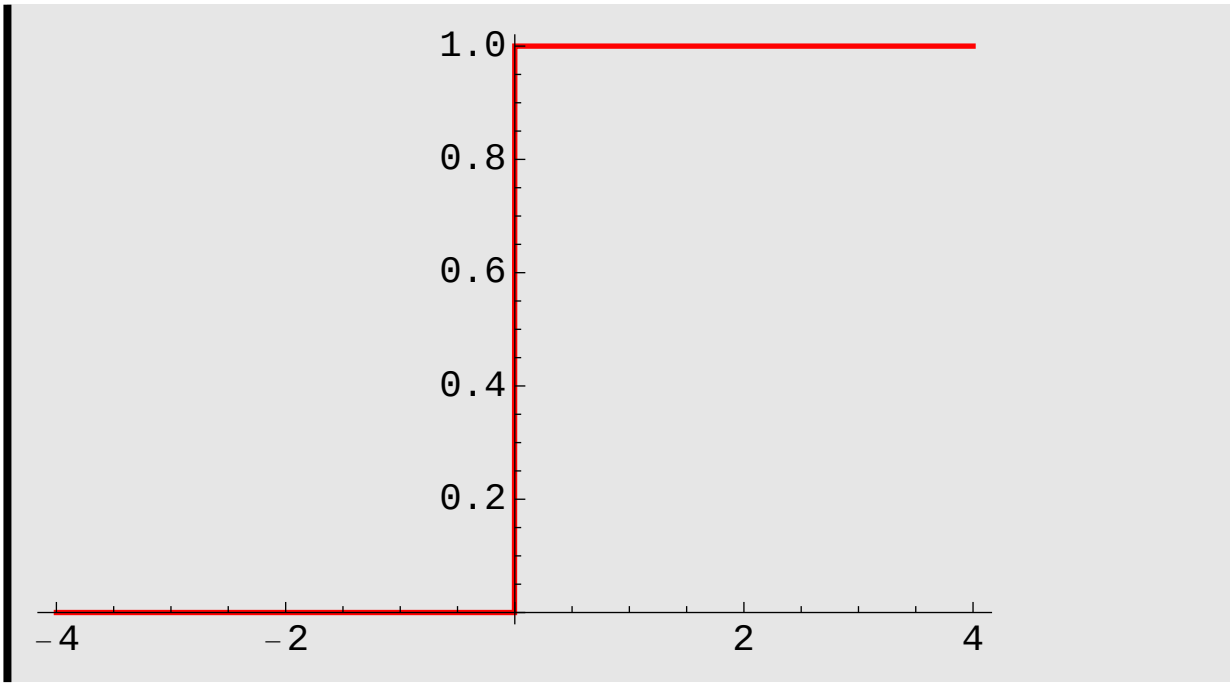
```
f[x_] = Sum[a[n] x^n, {n, 0, Infinity}]
```

$$\frac{256\,\pi}{88\,179\,(1-x)}$$

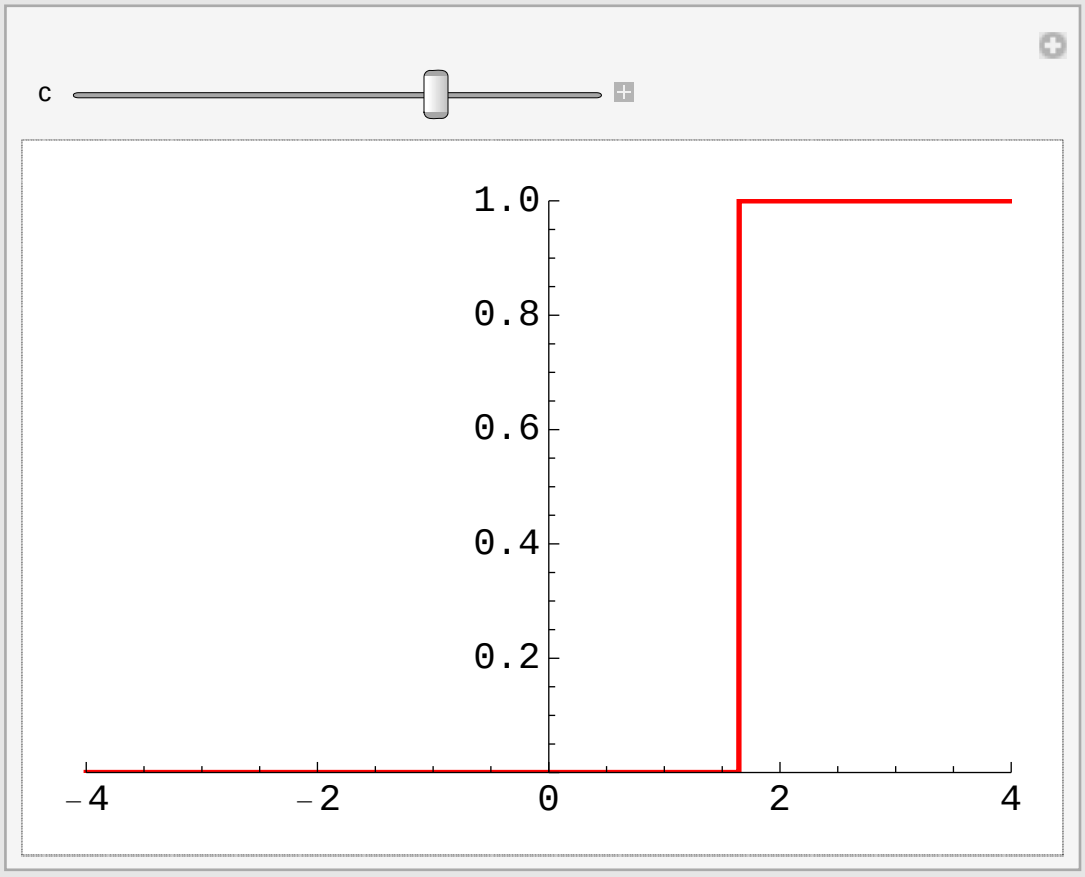
This won't always work, but you will be surprised how often it does!

Intrinsic Function: Heaviside (used with Laplace Transforms)

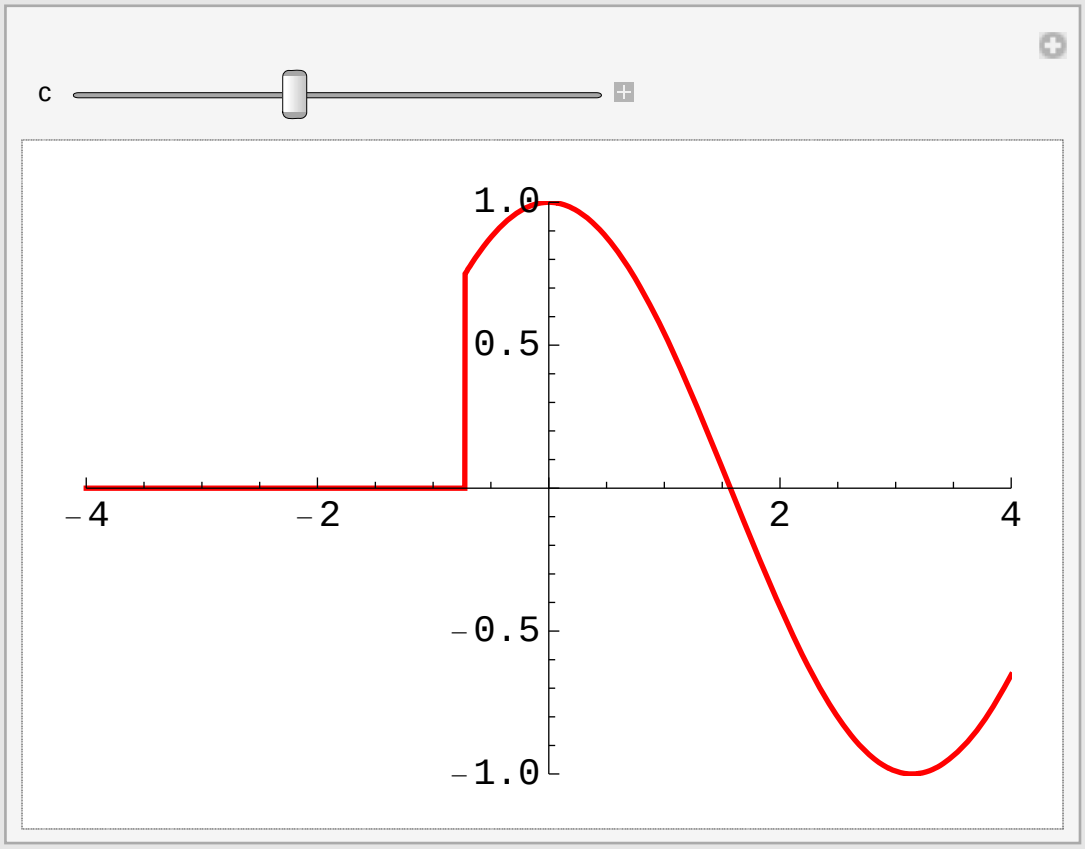
```
Plot[HeavisideTheta[t], {t, -4, 4},
 PlotStyle → {Red, Thick}]
```



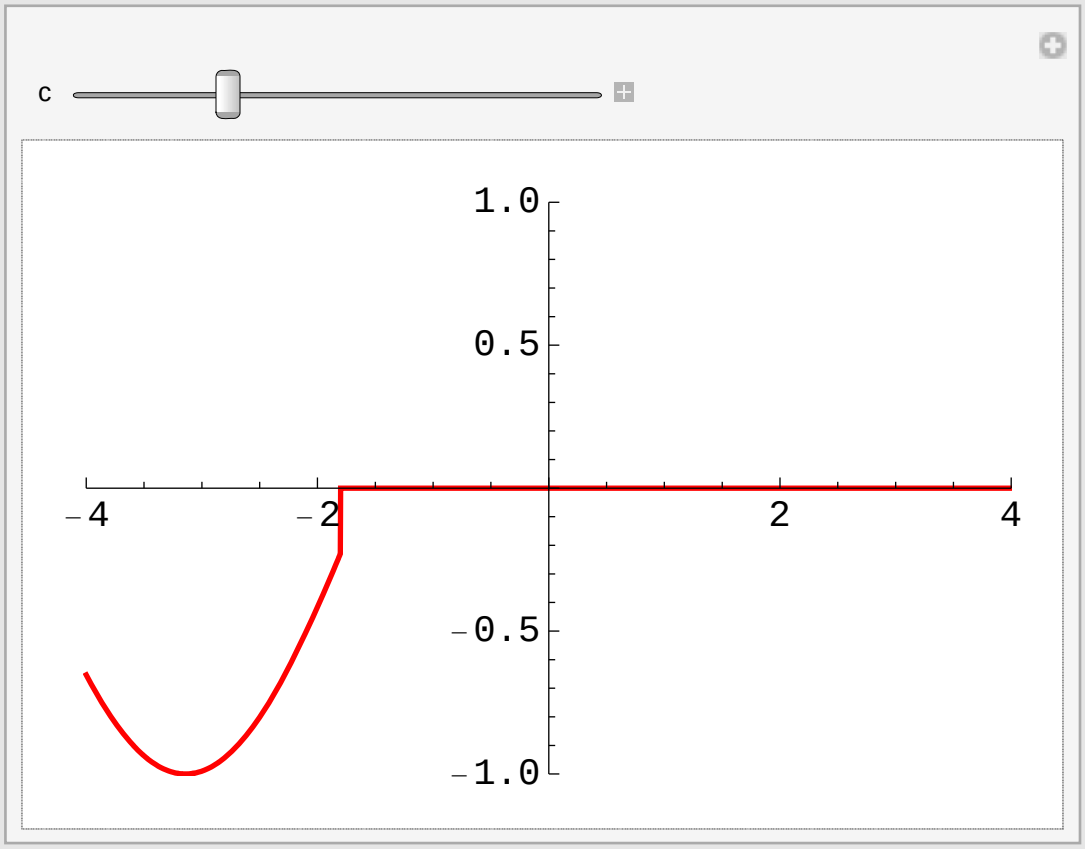
```
Manipulate[Plot[HeavisideTheta[t - c], {t, -4, 4},
  PlotStyle → {Red, Thick},
  PlotRange → {{-4, 4}, {0, 1}}], {c, -4, 4}]
```



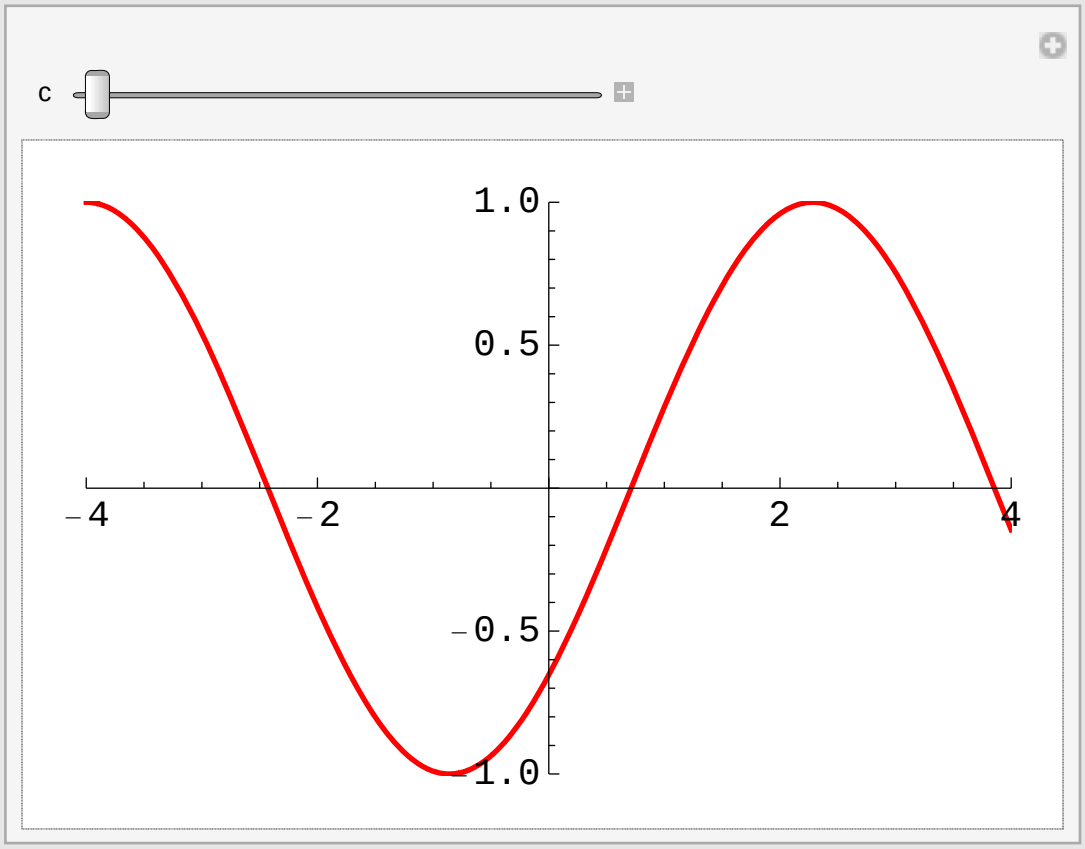
```
Manipulate[Plot[HeavisideTheta[t - c] * Cos[t],
  {t, -4, 4}, PlotStyle → {Red, Thick},
  PlotRange → {{-4, 4}, {-1, 1}}], {c, -4, 4}]
```



```
Manipulate[Plot[(1 - HeavisideTheta[t - c]) * Cos[t],
  {t, -4, 4}, PlotStyle → {Red, Thick},
  PlotRange → {{-4, 4}, {-1, 1}}], {c, -4, 4}]
```

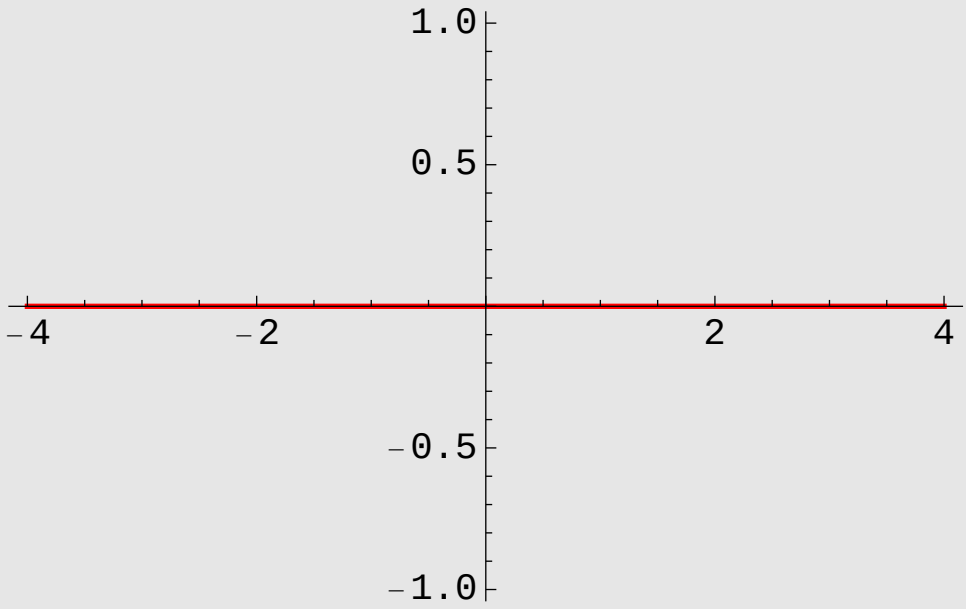


```
Manipulate[Plot[HeavisideTheta[t - c] * Cos[t - c],
  {t, -4, 4}, PlotStyle → {Red, Thick},
  PlotRange → {{-4, 4}, {-1, 1}}], {c, -4, 4}]
```



Intrinsic Function: Dirac delta (used with Laplace Transforms)

```
Plot[DiracDelta[t], {t, -4, 4},
 PlotStyle → {Red, Thick}]
```




```
Integrate[DiracDelta[t] f[t], {t, -20, 20}]
```

$$\frac{256\,\pi}{88\,179}$$

```
Integrate[DiracDelta[t] f[t], {t, 0, 20}]
```

$$\frac{128\,\pi}{88\,179}$$

```
Integrate[DiracDelta[t - 4] f[t], {t, 0, 20}]
```

$$-\frac{256\,\pi}{264\,537}$$

Matrix Manipulations

Mathematica is quite comfortable working with matrices and vectors. The command MatrixForm is used to make the output look nice.

Here is a matrix:

```
A = {{1, -1, 2}, {-1, -1 / 10, 4}, {6, 3, 0}}
MatrixForm[A]
```

$$\left\{\{1, -1, 2\}, \left\{-1, -\frac{1}{10}, 4\right\}, \{6, 3, 0\}\right\}$$

$$\begin{pmatrix} 1 & -1 & 2 \\ -1 & -\frac{1}{10} & 4 \\ 6 & 3 & 0 \end{pmatrix}$$

And some vectors:

```
vec1 = {1, 4, -3}
MatrixForm[vec1]

vec2 = {3, 4, 3}
MatrixForm[vec2]
```

```
{1, 4, -3}
```

$$\begin{pmatrix} 1 \\ 4 \\ -3 \end{pmatrix}$$

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

$$\begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix}$$

Vector Addition

```
vec1 + vec2
```

```
{4, 8, 0}
```

Dot Product

```
MatrixForm[A.vec1]
```

$$\begin{pmatrix} -9 \\ -\frac{67}{5} \\ 18 \end{pmatrix}$$

```
MatrixForm[Dot[A, vec1]]
```

$$\begin{pmatrix} -9 \\ -\frac{67}{5} \\ 18 \end{pmatrix}$$

Cross Product

```
MatrixForm[Cross[vec1, vec2]]
```

$$\begin{pmatrix} 24 \\ -12 \\ -8 \end{pmatrix}$$

Matrix Power

```
MatrixPower[A, 2]
MatrixForm[%]
```

$$\left\{\left\{14, \frac{51}{10}, -2\right\}, \left\{\frac{231}{10}, \frac{1301}{100}, -\frac{12}{5}\right\}, \left\{3, -\frac{63}{10}, 24\right\}\right\}$$

$$\begin{pmatrix} 14 & \frac{51}{10} & -2 \\ \frac{231}{10} & \frac{1301}{100} & -\frac{12}{5} \\ 3 & -\frac{63}{10} & 24 \end{pmatrix}$$

Exponential of a Matrix

```
MatrixExp[N[A]]
MatrixForm[%]
```

```
{{30.1232, 3.74021, 16.8767},
 {57.207, 14.6329, 38.506},
 {77.3634, 15.5128, 48.4183}}
```

$$\begin{pmatrix} 30.1232 & 3.74021 & 16.8767 \\ 57.207 & 14.6329 & 38.506 \\ 77.3634 & 15.5128 & 48.4183 \end{pmatrix}$$

Eigenvales and Eigenvectors

```
Eigenvalues[A] // N
Eigenvectors[A] // N
Eigensystem[A] // N
```

```
{-5.29522, 4.47243, 1.72279}
```

```
{{-0.453885, -0.857304, 1.},
 {0.345816, 0.799178, 1.}, {-1.11629, 2.80684, 1.}}
```

```
{{-5.29522, 4.47243, 1.72279},
 {{-0.453885, -0.857304, 1.},
  {0.345816, 0.799178, 1.}, {-1.11629, 2.80684, 1.}}}
```

Complex Numbers

Here are a couple of complex numbers:

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

```
a = 3 - I
b = 7 + 5 I
```

```
3 - ⅈ
```

```
7 + 5 ⅈ
```

The **ComplexExpand** function helps us break a complicated complex valued function into its real and imaginary parts.

```
ComplexExpand[Exp[a * x]]
```

```
e^(3 x) Cos[x] - i e^(3 x) Sin[x]
```

```
ComplexExpand[Sin[b * y] Exp[a * x]]
```

```
e^(3 x) Cos[x] Cosh[5 y] Sin[7 y] +
 e^(3 x) Cos[7 y] Sin[x] Sinh[5 y] +
 i (-e^(3 x) Cosh[5 y] Sin[x] Sin[7 y] +
    e^(3 x) Cos[x] Cos[7 y] Sinh[5 y])
```

Mathematica's programming syntax

We will not be doing a tremendous amount of programming in this class, but you will probably want to know how to some simple programming in Mathematica to help you with some of your assignments.

Looping

Do, While and For do not print any output; they simply do what they are asked to do. After the calculation you will need to output whatever you are interested in. Table is very similar to Do, but it forms a list of the output. In many situations Table is all you will need. Other commands that may prove useful are Sum and Product.

Do Loops

```
Clear[a, b, c]

a = 0;
b = 0;
c = 0;
Do[
 a = a + n;
 b = a + b;
 c = b^2,
 {n, 1, 4}]

{a, b, c}
```

```
{10, 20, 400}
```

■ While Loops

```
Clear[a, b, c, n]

a = b = c = 0; n = 1;
While[n ≤ 4,
 a = a + n;
 b = a + b;
 c = b^2;
 n = n + 1]

{a, b, c}
```

```
{10, 20, 400}
```

For Loops

```
Clear[a, b, c, n]

For[a = b = c = 0; n = 1,
 n ≤ 4,
 n = n + 1,
 a = a + n;
 b = a + b;
 c = b^2]

{a, b, c}
```

```
{10, 20, 400}
```

Table Loops

```
Clear[a, b, c]

a = b = c = 0;
Table[
 a = a + n;
 b = a + b;
 c = b^2,
 {n, 1, 4}]

{a, b, c}
```

```
{1, 16, 100, 400}
```

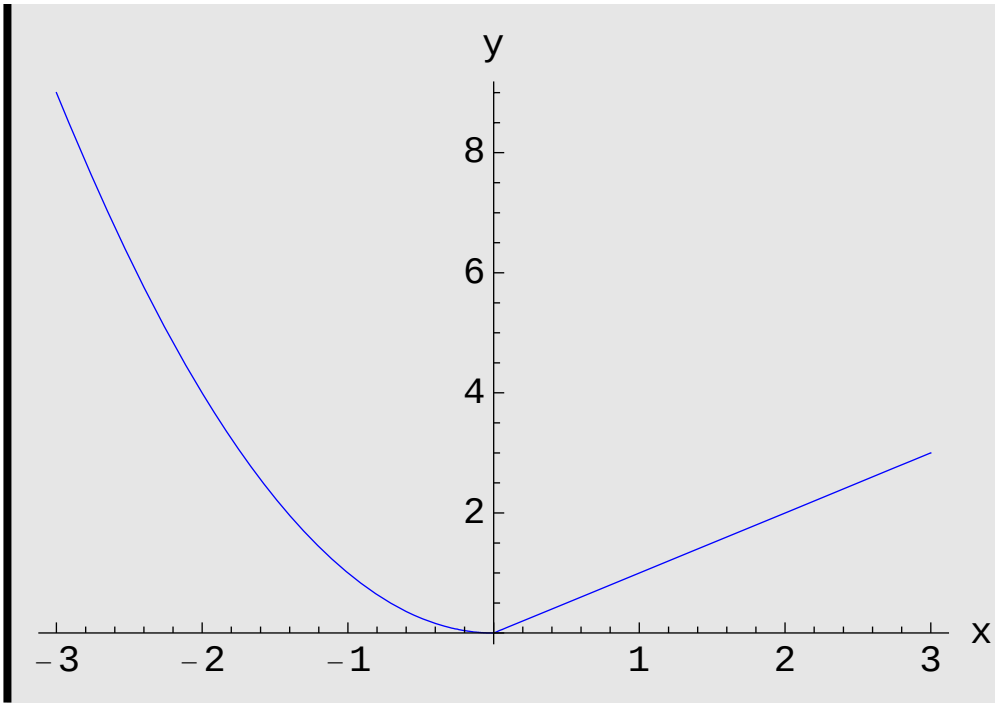
```
{10, 20, 400}
```

Branching

Branching in Mathematica can be accomplished via the commands If, Which, or Switch. The most common one you will use will probably be IF.

■ If

```
f[x_] = If[x ≤ 0, x^2, x];
Plot[f[x], {x, -3, 3}, AxesLabel → {"x", "y"},
 PlotStyle → {Blue}]
```



Flow Control

You can control the flow of programming in Mathematica using the commands Continue, Break, Return, and Goto. Here is a basic example which shows how these work.

```
f[n_] :=
 Do[
  Which[
   n == 1, Print["n=1 ", "k= ", k]; Continue[],
   n == 2, Print["n=2 ", "k= ", k]; Break[],
   n == 3, Return["That's All Folks"],
   True, Goto[jump];
   Label[jump]; Return[Print["n= ", n]]]
  , {k, 1, 3}]
```

```
f[1]
```

```
n=1 k= 1
n=1 k= 2
n=1 k= 3
```

```
f[2]
```

```
n=2 k= 1
```

```
f[3]
```

```
That's All Folks
```

```
f[4]
```

n= 4

Selecting Elements from a List

To select elements from a list, use the command Part.

```
Clear[a, b, c]
data = Table[{a[n], b[n], a[n] * b[n]^2}, {n, 0, 5}];
TableForm[data,
 TableHeadings → {None, {"1", "2", "3"}}]

data2 =
  Table[{Part[data, n + 1, 2], Part[data, n + 1, 3]},
   {n, 0, 5}];
TableForm[data2, TableHeadings → {None, {"2", "3"}}]
```

1	2	3
a[0]	b[0]	a[0] b[0]^2
a[1]	b[1]	a[1] b[1]^2
a[2]	b[2]	a[2] b[2]^2
a[3]	b[3]	a[3] b[3]^2
a[4]	b[4]	a[4] b[4]^2
a[5]	b[5]	a[5] b[5]^2

2	3
b[0]	a[0] b[0]^2
b[1]	a[1] b[1]^2
b[2]	a[2] b[2]^2
b[3]	a[3] b[3]^2
b[4]	a[4] b[4]^2
b[5]	a[5] b[5]^2

A shorthand does exist for `Part`, and it is done by typing <esc> [[<esc> and then the part of the item you want followed by <esc>]] <esc>. Here is an example, try it yourself!

The full list of data:

```
data
```

```
{{a[0], b[0], a[0] b[0]^2}, {a[1], b[1], a[1] b[1]^2},
 {a[2], b[2], a[2] b[2]^2}, {a[3], b[3], a[3] b[3]^2},
 {a[4], b[4], a[4] b[4]^2}, {a[5], b[5], a[5] b[5]^2}}
```

The third element in data:

```
data[[3]]
```

```
{a[2], b[2], a[2] b[2]^2}
```

The second element in the third element in data:

```
data[[3, 2]]
```

```
b[2]
```

Numerical Output

If you simply leave Mathematica to its defaults, it will output numbers to 6 digit precision, or significant digits precision. For example:

```
N[Pi]
N[Pi, 10]
N[3 / 4]
N[3 / 4, 8]
```

```
3.14159
```

```
3.141592654
```

```
0.75
```

```
0.75000000
```

If we want to include more digits in the output, we should use the command `SetPrecision` when we output the number. We would still like Mathematica to work with high digits of precision internally, but we want to be able to see as many digits as we like when needed.

```
SetPrecision[Pi, 9]
SetPrecision[3 / 4, 1]
SetPrecision[3 / 4, 5]
```

```
3.14159265
```

```
0.8
```

```
0.75000
```

The machine precision of Mathematica can be determined:

```
$MachinePrecision
```

```
15.9546
```

If we use the symbolic representation of numbers in Mathematica to do arithmetic (variable precision arithmetic) we get no round off error, but our calculations will be slow.

```
Pi - Pi
Timing[Sum[(1 - 24/n)^n, {n, 1, 600}]];
SetPrecision[%, 16]
```

```
0
```

```
{2.311999999999999, -1.318913252788608}
```

That took over 2 seconds on my computer (the first number), and the result was –1.3189 ...(we actually calculated it as a rational number, not a decimal, but it is so long I supressed the output!).

If we use the hardware to do our arithmetic (fixed precision arithmetic), we get results much faster, but we introduce roundoff error:

```
N[Pi, 25] - N[Pi, 25]
Timing[Sum[(1 - 24.0/n)^n, {n, 1, 600}]]
SetPrecision[%, 16]
```

$$0. \times 10^{-25}$$

$$\{3.20056 \times 10^{-16}, -1.31891\}$$

$$\{3.200564813177209 \times 10^{-16}, -1.318913252785305\}$$

This may not seem like a big time difference for the accuracy you get out at the end, and it really isn't for this simple case. However, when you inadvertantly use arbitrary precision numbers inside a looping structure, you can end up with code which takes days to run instead of minutes.

Putting It All Together: Euler's Method

For solutions to assignment problems, I am not expecting you to do high level programming in Mathematica. For example, here are some ways of implementing Euler's method to solve an initial value problem. I would expect you to be able to create the recursion relation method, or the While loop method, but not the module method. However, you should feel free to explore some of the more advanced syntax of Mathematica if it pleases you!

Here we want to solve the initial value problem $y'(t) = f(y, t)$, $y(\alpha) = \beta$ for $y(t)$ in a given range $\beta \le t \le \gamma$. Euler's method generates approximations to the solution $y(t)$ using the following relations, once you have picked a value for the step size h:

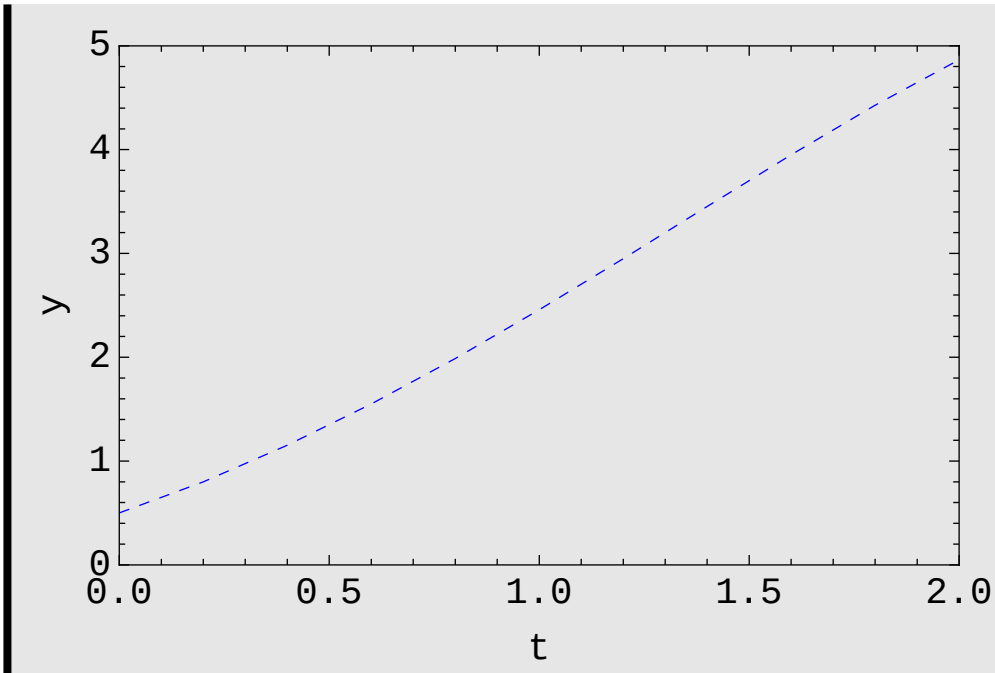
$$t_n = t_{n-1} + h$$
$$y_n = y_{n-1} + h * f(y_{n-1}, t_{n-1}),\ n = 1, 2, 3, \ldots$$
$$t_0 = \alpha,\ y_0 = y(\alpha) = \beta.$$

Let $f(y, t) = y - t^2 + 1$, $y(0) = 1/2$ and $0 \le t \le 2$. I've included some extra tables just to show you what Mathematica can do.

■ Euler's Method: While Loop

```
Clear[f, y, t, a, b, α, data, num]
f[t_, y_] = y - t^2 + 1;
a = 0.0;
b = 2.0;
α = 0.5;
num = 10;
h = (b - a)/num;
yi = α;
ti = a;
data = {{ti, yi}};
n = 1;
While[n ≤ num,
 {ti, yi} = N[{t + h, y + h f[t, y]} /. {t → ti, y → yi}];
 data = Append[data, {ti, yi}]; n = n + 1]
SetPrecision[data, 9]
ListPlot[data, PlotRange → {{0, 2}, {0, 5}},
 Joined → True, Frame → True, FrameLabel → {"t", "y"},
 PlotStyle → {Blue, Dashed}]
```

```
{{0, 0.500000000}, {0.200000000, 0.800000000},
 {0.400000000, 1.15200000},
 {0.600000000, 1.55040000},
 {0.800000000, 1.98848000},
 {1.00000000, 2.45817600}, {1.20000000, 2.94981120},
 {1.40000000, 3.45177344}, {1.60000000, 3.95012813},
 {1.80000000, 4.42815375}, {2.00000000, 4.86578450}}
```

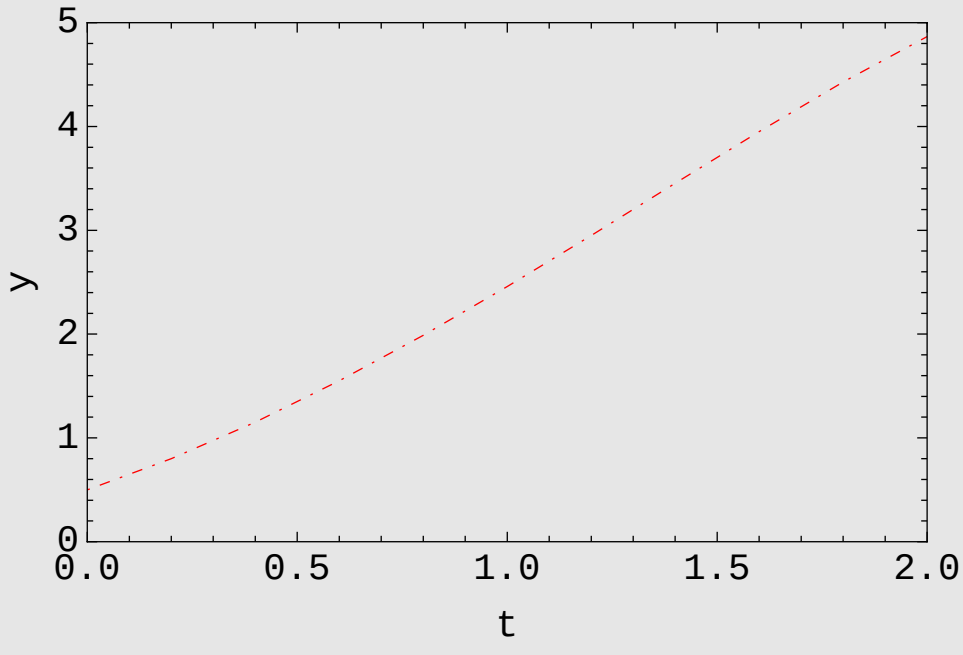

5
4
3
2
1
0
y
0.0
0.5
1.0
1.5
2.0
t

■ Euler's Method: Recursion Relations

```
Clear[f, y, t, a, b, α, data]
f[t_, y_] = y - t^2 + 1;
a = 0.0;
b = 2.0;
α = 0.5`;
num = 10;
h = (b - a)/num;
t[0] = a;
y[0] = α;
y[n_] := y[n] = y[n - 1] + h f[t[n - 1], y[n - 1]]
t[n_] := t[n] = a + n h
data = Table[{n, t[n], N[y[n], 20]}, {n, 0, num}];
SetPrecision[data, 9]
data2 = Table[{data[[n + 1, 2]], data[[n + 1, 3]]},
   {n, 0, num}];
SetPrecision[data2, 9]
phase = Table[{data[[n + 1, 2]], data[[n + 1, 3]]},
   {n, 0, num}];
ListPlot[phase, PlotRange → {{0, 2}, {0, 5}},
 Joined → True, Frame → True, FrameLabel → {"t", "y"},
 PlotStyle → {Red, DotDashed}]
```

```
{{0, 0, 0.500000000},
 {1.00000000, 0.200000000, 0.800000000},
 {2.00000000, 0.400000000, 1.15200000},
 {3.00000000, 0.600000000, 1.55040000},
 {4.00000000, 0.800000000, 1.98848000},
 {5.00000000, 1.00000000, 2.45817600},
 {6.00000000, 1.20000000, 2.94981120},
 {7.00000000, 1.40000000, 3.45177344},
 {8.00000000, 1.60000000, 3.95012813},
 {9.00000000, 1.80000000, 4.42815375},
 {10.0000000, 2.00000000, 4.86578450}}
```

```
{{0, 0.500000000}, {0.200000000, 0.800000000},
 {0.400000000, 1.15200000},
 {0.600000000, 1.55040000},
 {0.800000000, 1.98848000},
 {1.00000000, 2.45817600}, {1.20000000, 2.94981120},
 {1.40000000, 3.45177344}, {1.60000000, 3.95012813},
 {1.80000000, 4.42815375}, {2.00000000, 4.86578450}}
```



Euler's Method: Modules

Modules allow you to create executable blocks of code, with variables that are internal to the module.

```
Clear[f, y, t, a, b, α, data]
```

```
euler[f_, t_, y_, t0_, y0_, h_, n_] :=
 Module[{ti = t0, yi = y0},
  Prepend[
   Table[
    {ti, yi} = N[{t + h, y + h * f} /. {t → ti, y → yi}],
    {k, 1, n}], {t0, y0}]]
```

Here is the output from executing the module:

```
euler[ y - t^2 + 1, t, y, 0.0, 0.5, 0.2, 10]
```

```
{{0., 0.5}, {0.2, 0.8}, {0.4, 1.152},
 {0.6, 1.5504}, {0.8, 1.98848}, {1., 2.45818},
 {1.2, 2.94981}, {1.4, 3.45177},
 {1.6, 3.95013}, {1.8, 4.42815}, {2., 4.86578}}
```

Here the output is put into a table, and plotted.

```
approx = euler[y - t^2 + 1, t, y, 0., 0.5, 0.2, 10];
SetPrecision[approx, 8]
ListPlot[approx, PlotRange → {{0, 2}, {0, 5}},
 Joined → True, Frame → True, FrameLabel → {"t", "y"},
 PlotStyle → {Dotted}]
```

```
{{0, 0.50000000}, {0.20000000, 0.80000000},
 {0.40000000, 1.1520000},
 {0.60000000, 1.5504000}, {0.80000000, 1.9884800},
 {1.0000000, 2.4581760}, {1.2000000, 2.9498112},
 {1.4000000, 3.4517734}, {1.6000000, 3.9501281},
 {1.8000000, 4.4281538}, {2.0000000, 4.8657845}}
```