A Mathematica Primer

For Students of Physics 218: Oscillatory and Wave Phenomena

Brooks Thomas
Lafayette College

Third Edition

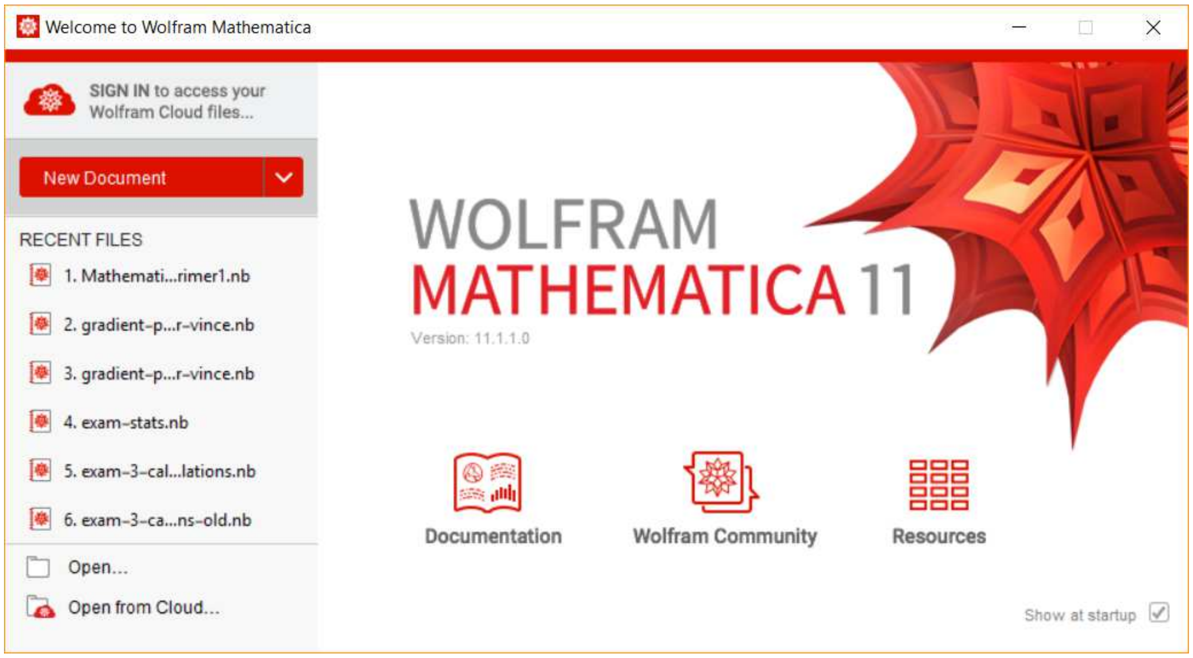
2018

Chapter 1

Basic Syntax

Getting started

After you double-clicking on the Mathematica 10 icon to open Mathematica, the first thing you'll want to do is create a new Mathematica notebook. A notebook is a file in which you can create, edit, and execute Mathematica code. The information in a notebook — including both your input (the code you write) and output (plots, drawings, the results of numerical calculations, *etc.*) can be saved and loaded. To create a notebook from Mathematica's welcome screen, click on the arrow next to the "New Document" icon and select "Notebook" from the drop-down menu.



Once you have a notebook open, you can always create a new notebook by navigating to "File → New → Notebook" from the main toolbar at the top of the window.

Once you've opened the notebook, you're ready to begin interacting with Mathematica. Mathematica is an incredibly powerful numerical tool, and there are a *lot* of things that it can do. Fortunately, despite all of that sophistication, Mathematica is also pretty easy to get started with. The basic procedure for interacting with a Mathematica notebook is that you begin by typing something (a formula, a command, the definition of a function, *etc.*) on a line. When you're finished with what you're typing, you can hit the "Enter" or "Return" key to go to a new line. When you're ready, you can also hit "Shift + Enter" or "Shift + Return" to evaluate all of the expressions or commands you've just written, at which point output of each line will appear, in order, beneath the code you've written. The set of lines of code that get evaluated together when you hit "Shift + Enter" are called a "cell." You can tell which lines of code are in the sane cell by looking on the right-hand side of the notebook window, thin, blue braces appear to indicate which items are in the same cell.

The fact that you have to hit "Shift + Enter" rather than just "Enter" to execute Mathematica code might take a little getting used to, but it becomes second nature pretty quickly.)

The basic syntax of Mathematica is explained below.

Functions

Arithmetic operations and exponentiation in Mathematica are fairly self-explanatory:

```
In[12]:=  6 * 2

Out[12]=  12


In[2]:=  6 + 2

Out[2]=  8


In[3]:=  6 - 2

Out[3]=  4


In[4]:=  6 / 2

Out[4]=  3


In[5]:=  6^2

Out[5]=  36
```

You can define a function in Mathematica using the syntax

```
FunctionName[variablename_] :=
```

followed by whatever you want the function to return. For example, if we wanted to define a function with an argument x that returned $2x^2$, we might write

```
In[13]:=  TwoXCubed[x_] := 2 * x^3
          TwoXCubed[5]

Out[14]=  250
```

Likewise, if we wanted to define a function that multiplies the argument by a variable called a, we might write

```
In[15]:=  MultBya[x_] := a * x
          MultBya[3]

Out[16]=  3 a
```

Note that the symbol a is blue in color in the input text here. This is Mathematica's way of indicating that the variable a hasn't been explicitly assigned a value yet. Assigning a value to a variable is done using the following syntax

```
In[17]:= a = 10

Out[17]= 10
```

Now the symbol a appears black rather than blue, indicating that it has been assigned a value. Now that it has been assigned a value, it will retain that value in future calculations. For example, if we execute the **MultBya** function we defined above again now, we'd obtain

```
In[18]:= MultBya[3]

Out[18]= 30
```

However, if we ever want to clear the value of a, we can do so by entering

```
In[19]:= Clear[a]
```

Note that after this command is executed, the symbol a appears blue again, and executing the **MultBya** command once again yields

```
In[20]:= MultBya[3]

Out[20]= 3 a
```

It's also worth pointing out that the argument which is fed to a function in Mathematica need not be explicitly defined. For example, feeding a itself to our function (after its value has been cleared) yields

```
In[21]:= MultBya[a]
```

Out[21]= a^2

You can also define functions with multiple arguments in Mathematica. For example, if we wanted a function that multiplied two numbers together and squared the result, we might write

```
In[22]:= ProductSquared[x_, y_] := (x*y)^2
         ProductSquared[4, 2]

Out[23]= 64
```

In addition to the functions that you define yourself, Mathematica also has a large number of built-in functions that you can call at any time. These include the commonly used functions listed below.

`Sin[x]`	$\sin x$	`Log[x]`	$\ln x$
`Cos[x]`	$\cos x$	`Log10[x]`	$\log_{10} x$
`Tan[x]`	$\tan x$	`Log[b, x]`	$\log_b x$
`ArcSin[x]`	$\arcsin x$	`Sqrt[x]`	$\sqrt{x}$
`ArcCos[x]`	$\arccos x$	`Sinh[x]`	$\sinh x$
`ArcTan[x]`	$\arctan x$	`Cosh[x]`	$\cosh x$
`Exp[x]`	e^x	`Tanh[x]`	$\tanh x$
`Abs[x]`	$\|x\|$	`ArcSinh[x]`	$\text{arcsinh} x$

There are also a few built-in constants:

`Pi`	π
`E`	e
`I`	$i \equiv \sqrt{-1}$
`Degree`	$\pi/180$

The last one of these is intended to serve as a conversion factor between degrees and radians.

Lists

Lists are another commonly-used construction in Mathematica. They are used, for example, to represent vectors, to store the results of multiple trials or measurements in a data set, and as arguments for certain functions (such as the **Plot** and **Integrate** functions discussed later in this tutorial). You can create a list by typing the elements (numbers, variables, *etc.*) that you want to be in that list enclosed by curly brackets and separated by commas, as shown here

```
In[38]:=  {x, y, z}

Out[38]=  {x, y, z}
```

At this point in the course, we won't be using lists too much except for when we need to use them as arguments for built-in Mathematica functions. However, we will make extensive use of them later on in the semester when we begin dealing with matrices.

Commenting

As with any other programming language, it's often a good idea to include comments and annotations in your Mathematica notebooks which explain your code. In Mathematica, comments are enclosed within parentheses and asterisks like so:

```
In[6]:=  (*This function multiplies two numbers together and
          squares them.*)
         ProductSquared[x_, y_] := (x*y)^2
         ProductSquared[4, 2]

Out[7]=  64
```

The text of any comments in a Mathematical notebook appear light gray in color as an additional indication that they are comments and not active code.

Numerical Values and Approximations

Mathematica is designed for symbolic mathematics as well as for numerical approximations, and many numbers in Mathematica including the numbers π and e, are treated exactly. If you're interested in getting a numerical approximation for a quantity that involves numbers like this, you'll want to use the **N** command:

```
In[35]:= 2 * π
         N[2 * π]

Out[35]= 2 π

Out[36]= 6.28319
```

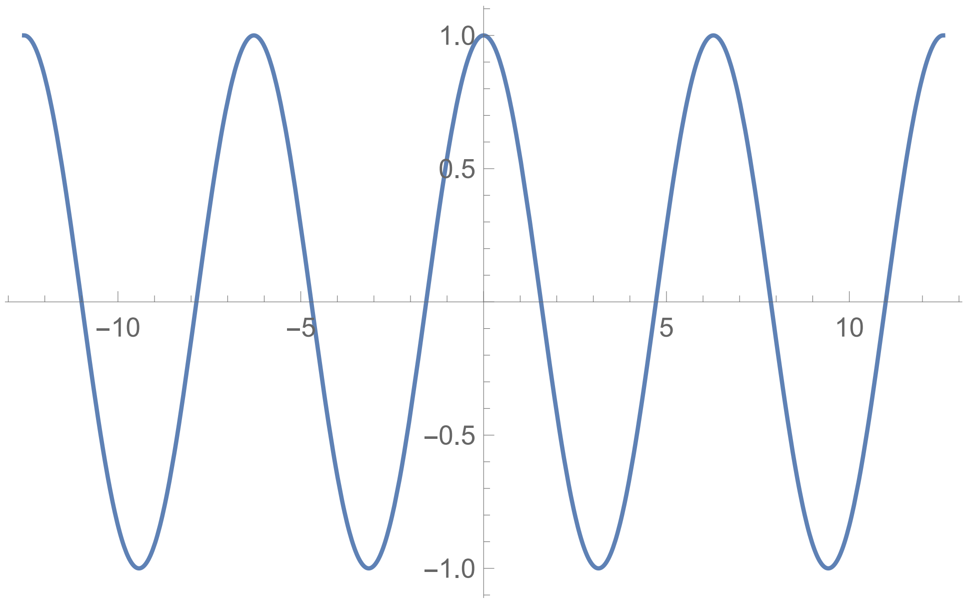
Other examples of quantities you'll need **N** to evaluate are square roots of integers and integers raised to miscellaneous powers. Also, if you want a numerical approximation of the quantity in question to a specified precision, you can add an additional argument to **N** in order to specify the number of digits of precision you want. For example, if you wanted to evaluate $s\pi$ to only 3 digits of precision, you could enter the following:

```
In[37]:= N[2 * π, 3]

Out[37]= 6.28
```

Plotting

Mathematica can also be used for plotting and graphical output. For example, let's say we wanted to plot the function $\cos x$ within the range $-4\pi \leq x \leq 4\pi$. To do this, we would evoke the the **Plot** command:

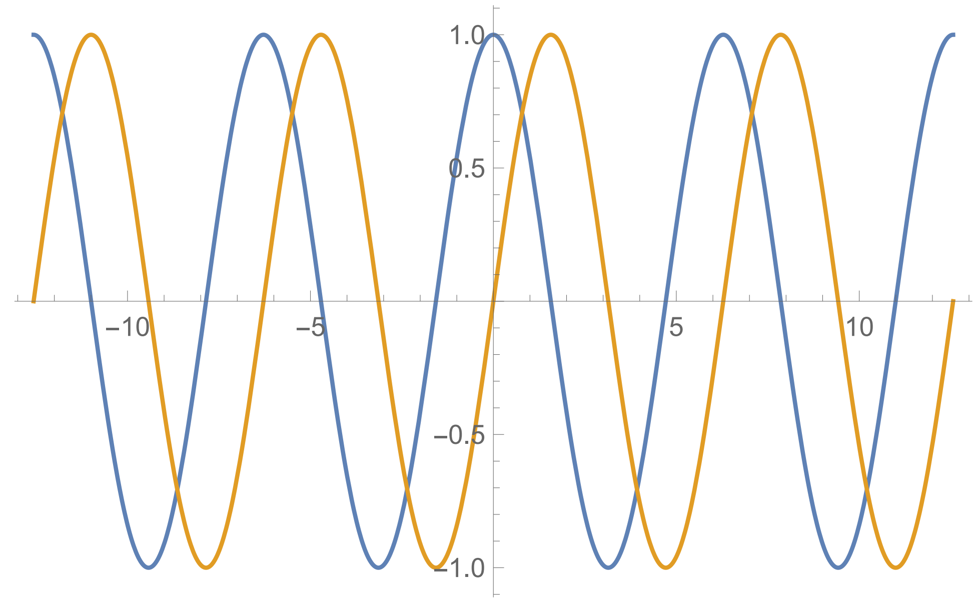
```
In[24]:= Plot[Cos[x], {x, -4 * Pi, 4 * Pi}]

Out[24]=
```



where the first argument of **Plot** is the function to be plotted and the second argument (which is obligatory for this command) consists comma-separated list of the name of the independent variable and the minimum and maximum values of that variable to be shown on the plot. We can also overlay plots of multiple functions by making the first argument a list. For example:

```
In[25]:= Plot[{Cos[x], Sin[x]}, {x, -4*Pi, 4*Pi}]
```

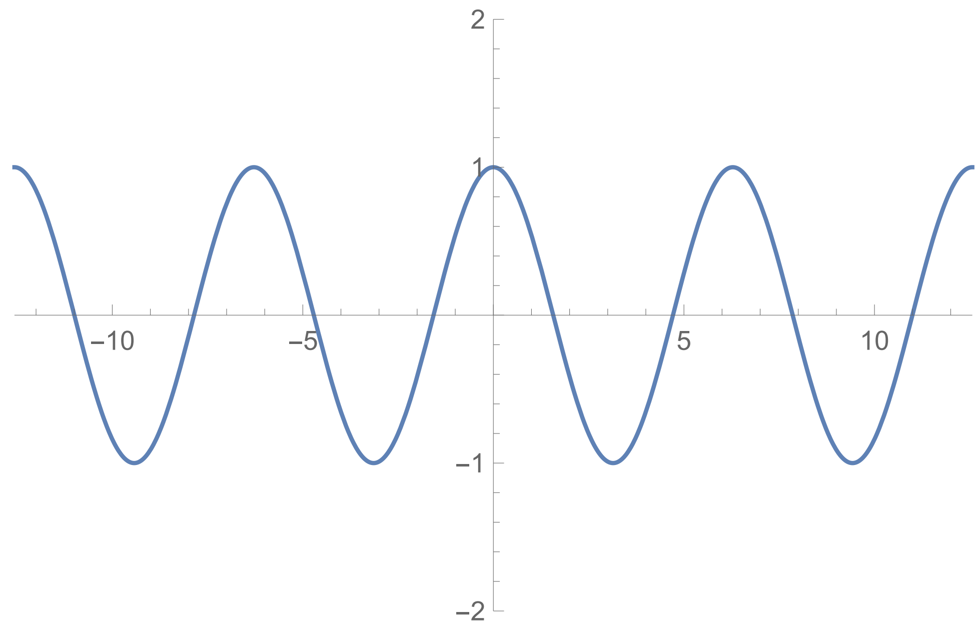
Out[25]=



This is the most basic plot we can create in Mathematica. However, Mathematica provides a large number of options which can be used to modify the appearance of the plot. For example, if we wanted to change the range of the y axis from the default value $-1 \leq y \leq 1$ Mathematica gave us, we can use the **PlotRange** option to specify the range for both x and y. The syntax is

```
In[27]:= Plot[{Cos[x]}, {x, -4*Pi, 4*Pi},
           PlotRange → {{-4*Pi, 4*Pi}, {-2, 2}}]
```

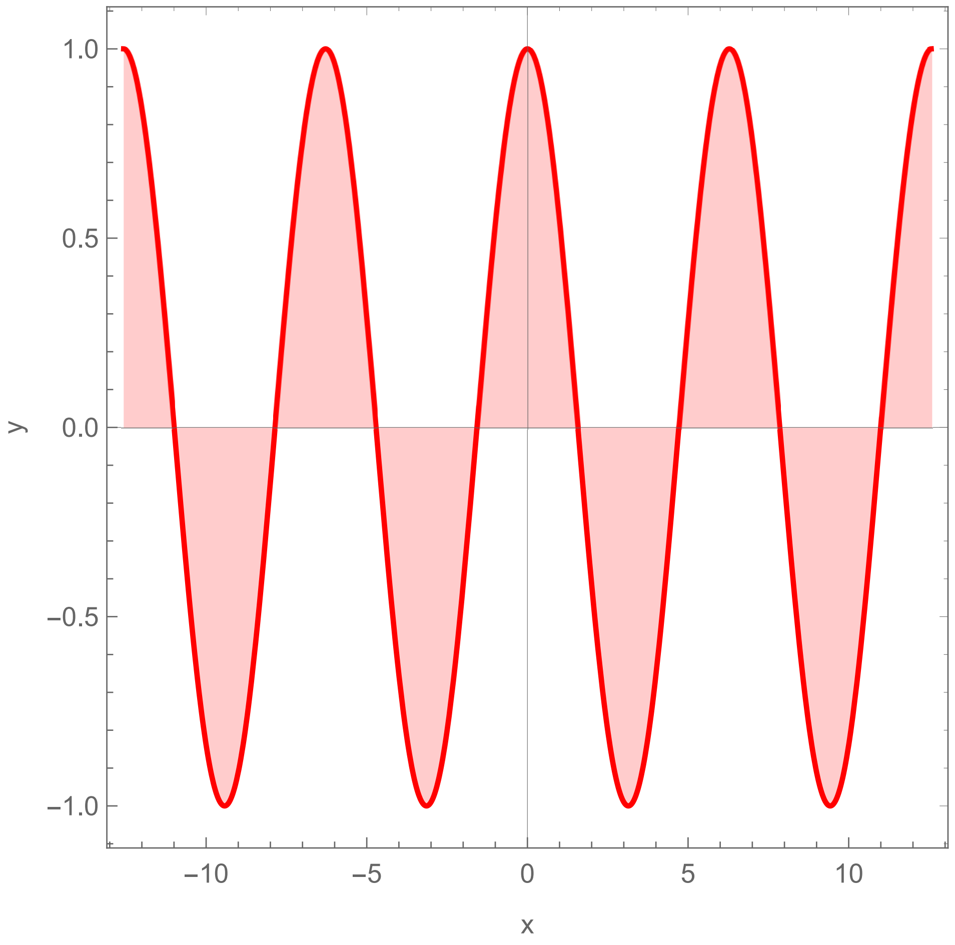
Out[27]=



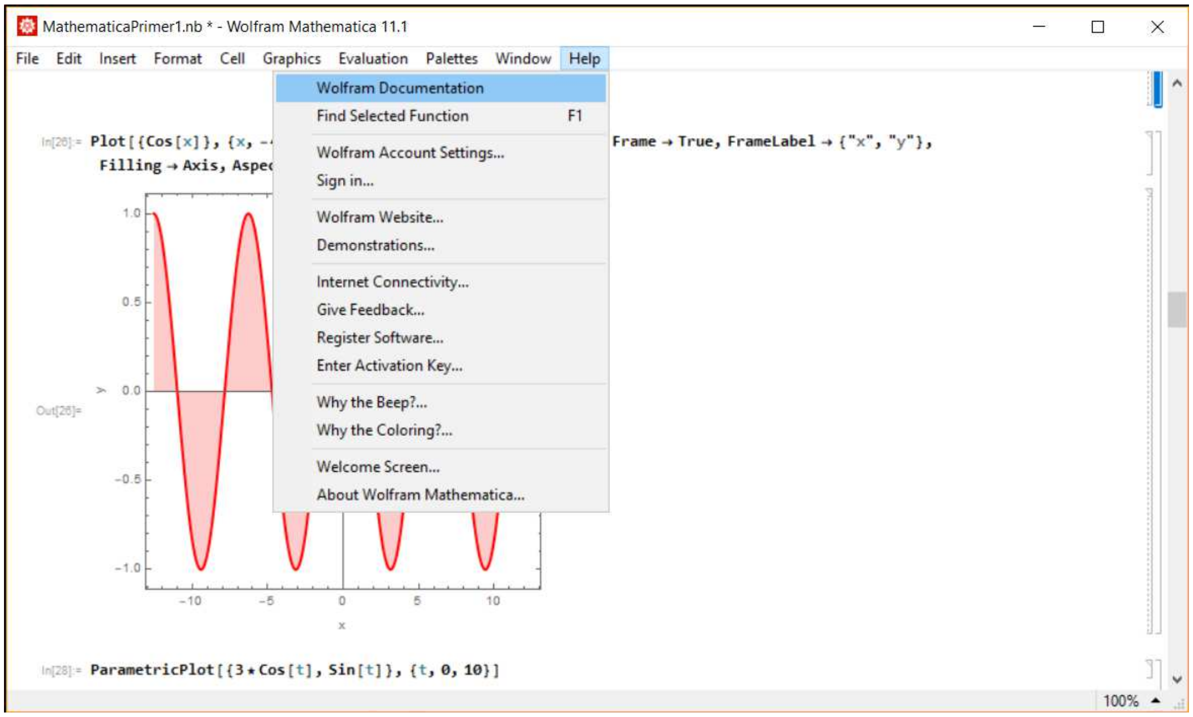
There are other options for **Plot** which allow the user to explicitly specify the colors, thicknesses, the dashings of the curves plotted; add labels to the x and y axes; enclose the plot in a rectangular frame; change the aspect ratio (the relative lengths of the x and y axes in the output image); and shade ares between different curves. All of these options are invoked in the following plot:

```
In[26]:= Plot[{Cos[x]}, {x, -4*Pi, 4*Pi}, PlotStyle → {Red, Thick},
          Frame → True, FrameLabel → {"x", "y"}, Filling → Axis,
          AspectRatio → 1]
```

Out[26]=



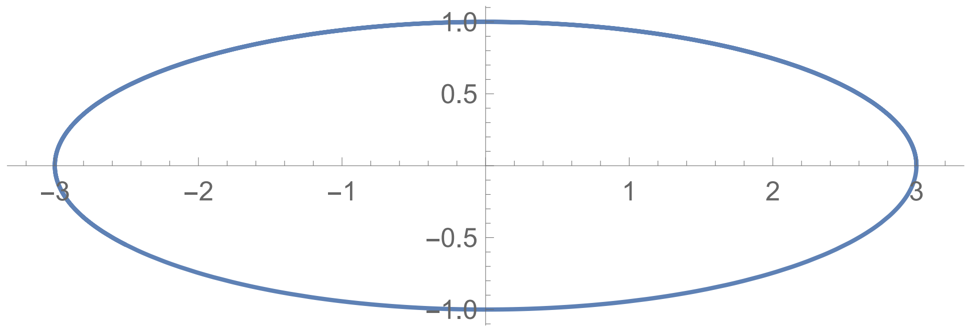
You can find out more about the syntax used to declare these options from the “Help” menu on the toolbar at the top of the Mathematica window.



```
In[26]:= Plot[{Cos[x]}, {x, -                              Frame → True, FrameLabel → {"x", "y"},
     Filling → Axis, Aspec
Out[26]=

In[28]:= ParametricPlot[{3 * Cos[t], Sin[t]}, {t, 0, 10}]
```

In addition to the basic plots illustrated above, Mathematica is also capable of making other kinds of plots as well. One kind of plot which is particularly useful, for example, for plotting state-space trajectories is a parametric plot. Such a plot displays a curve whose x and y coordinates are both functions of some underlying parameter t. For example, if the relevant functions were $x(t) = 3\cos t$ and $y(t) = \sin t$, then we could create a parametric plot using the following syntax:

```
In[28]:= ParametricPlot[{3 * Cos[t], Sin[t]}, {t, 0, 10}]
```

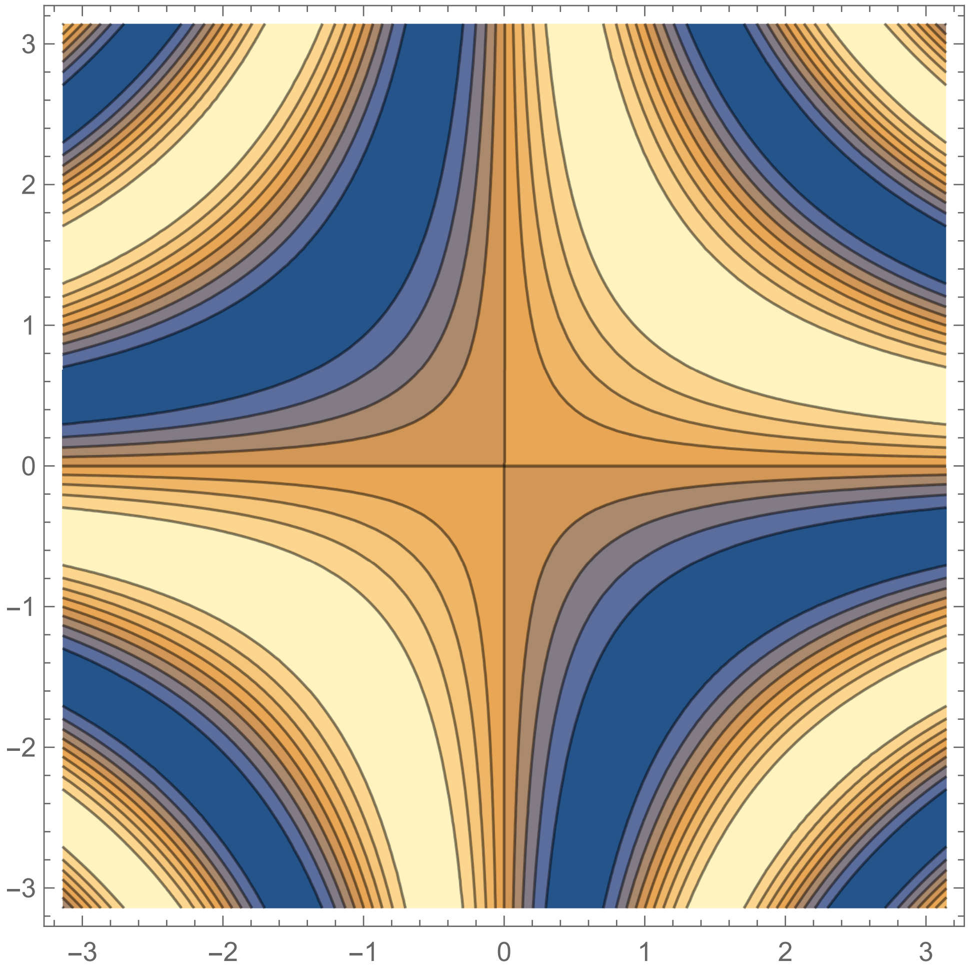
Out[28]=


where the first argument of the **ParametricPlot** function is a list of the functions $x(t)$ and $y(t)$, and the second argument gives the name of the underlying parameter and the endpoints of the range of values of this parameter for which $x(t)$ and $y(t)$ are to be plotted.

Mathematica can also produce a contour plot of a function $f(x, y)$ which depends on two variables. For example, a plot of the function $\sin(xy)$ can be created using the following syntax:

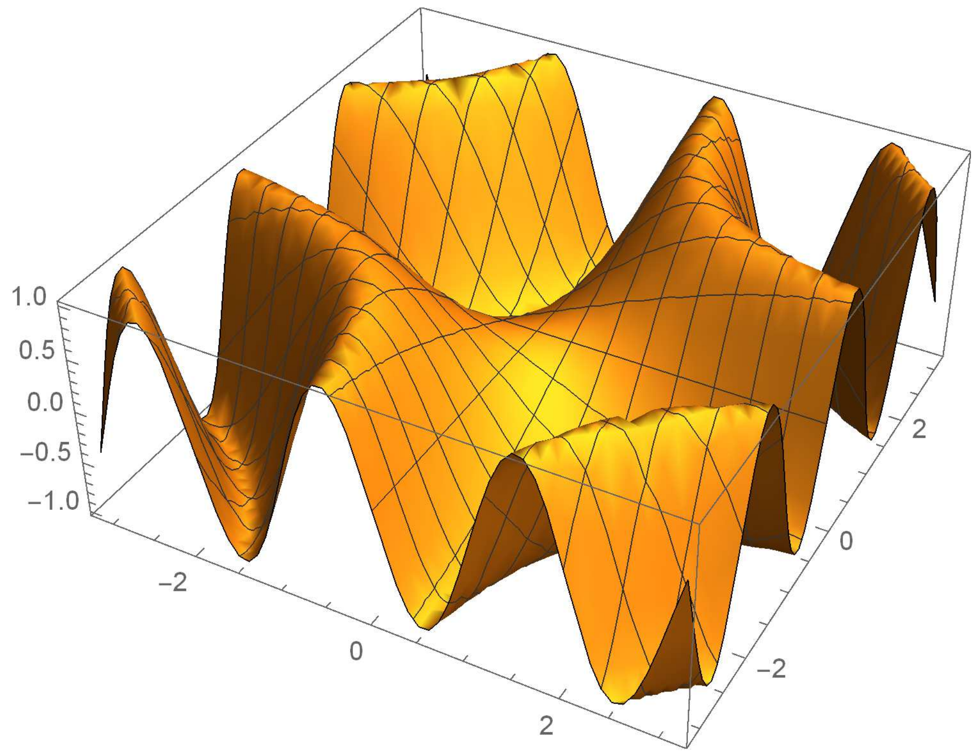
```
In[39]:= ContourPlot[Sin[x*y], {x, -Pi, Pi}, {y, -Pi, Pi}]
```

Out[39]=



The **ContourPlot** function takes three arguments, since we need to specify the range for both x and y. It is also possible to create a three-dimensional plot using the **Plot3D** command, the syntax for which is very similar to that of **ContourPlot**:

```
Plot3D[Sin[x*y], {x, -Pi, Pi}, {y, -Pi, Pi}]
```



Further documentation about these and other plotting commands is also available from the "Help" tab in the Mathematica toolbar.

Calculus

Mathematica is also capable of performing a large number of calculus operations, including basic differentiation and integration. The derivative

$$\frac{d}{dx}f(x)$$

of a function $f(x)$ with respect to the argument x is evaluated using the **D** function. For example, if we wanted to evaluate the derivatives of the functions $x^3 + 2x^2 + x$ and $\cos x$, we would write

```
In[29]:= D[x^3 + 2*x^2 + x, x]
         D[Cos[x], x]

Out[29]= 3 x^2 + 4 x + 1

Out[30]= -sin(x)
```

The first argument of **D** is the function to be evaluated, and the second is the differentiation variable.

Integration in Mathematica is handled by the **Integrate** function, which is used for both definite and indefinite integrals. To evaluate the indefinite integral

$$\int f(x)dx$$

of a function $f(x)$ with respect to the variable x, the syntax is

```
In[31]:= Integrate[3 x^2 + 4 x + 1, x]
```

Out[31]= $x^3 + 2\,x^2 + x$

The syntax for a definite integral

$$\int_a^b f(x)dx \ ,$$

with limits of integration a and b is similar, except for that the second argument of **Integrate** is the three-item list $\{x, a, b\}$. For example, if we wanted to evaluate the definite integral

$$\int_0^4 \left[3x^2 + 4x + 1\right] dx \ ,$$

we would write

```
In[32]:= Integrate[3 x^2 + 4 x + 1, {x, 0, 4}]
```

Out[32]= 100

It's also possible to use Mathematica to obtain Taylor-series expansions of a given function using the **Series** function. For example, in order to obtain a Taylor-series expansion for $\cos x$ up to and including terms in x^4, we would write

```
In[33]:= Series[Cos[x], {x, 0, 4}]
```

Out[33]= $1 - \frac{x^2}{2} + \frac{x^4}{24} + O\left(x^5\right)$

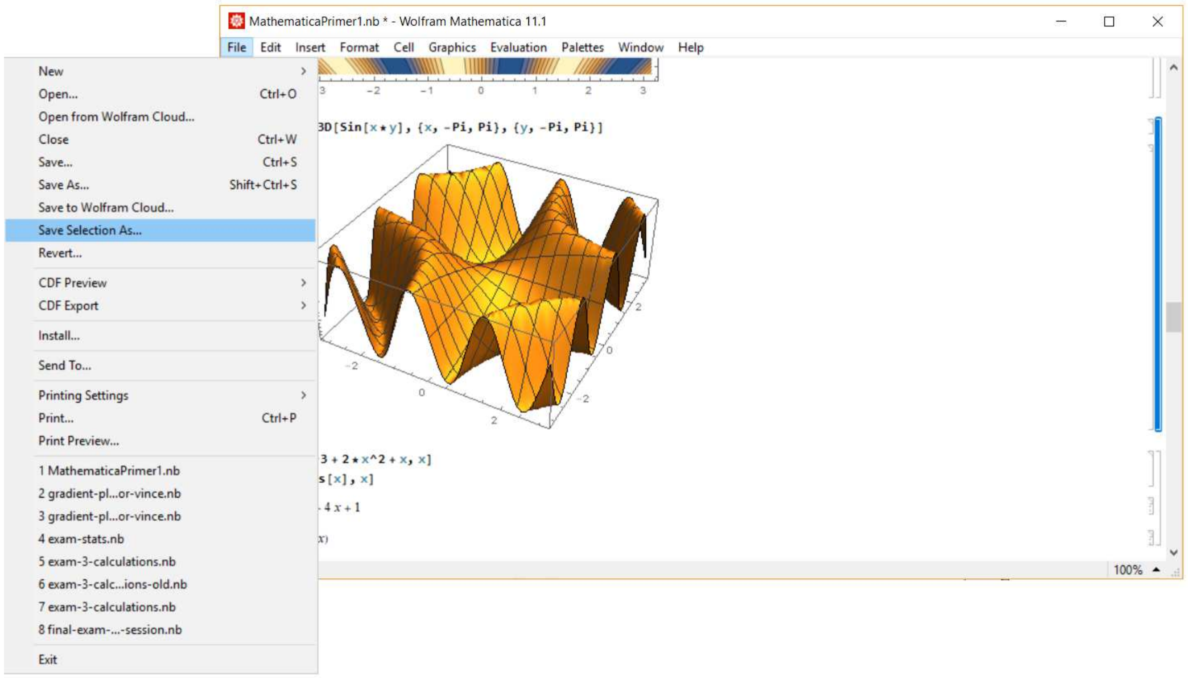
The three items appearing in the kist in the second argument of this function are the name of the expansion variable, the value around which we're expanding, and the highest power of x to keep in the expansion. The final symbol appearing in the output line indicates that terms in x^5 and higher are being neglected. In order to get rid of this symbol (which you actually have to do in order to substitute the series into another expression or manipulate it in some other way), just use the **Normal** command:

```
In[34]:= Normal[Series[Cos[x], {x, 0, 4}]]
```

Out[34]= $\frac{x^4}{24} - \frac{x^2}{2} + 1$

Saving and Exporting

Saving a Mathematica notebook you've created is as simple as going to the "File" menu in the toolbar and selecting "Save" or "Save as…" There is also a procedure for exporting specific cells to files. This can be useful, for example, for saving individual plots that you might want to include in other files, send to other people, or print out. The first step in this procedure is to left-click your mouse on the blue brace at the right side of the scree which corresponds to that cell to highlight it. Next, go to the "File" menu and scroll down to "Save selection as...":



```
3D[Sin[x*y], {x, -Pi, Pi}, {y, -Pi, Pi}]
3 + 2*x^2 + x, x]
s[x], x]
4 x + 1
x)
```

A pop-up menu will then appear, prompting you for a filename, a file format for the new file, *etc.*

Exercises

1. The general solution $x(t)$ to the simple harmonic oscillator equation is

$$x(t) \; = \; A\cos(\omega t + \phi) \; .$$

   (a) Make a plot in Mathematica which shows x as a function of t over the range $-T \le t \le T$, where T is the period of oscillation. In costructing your plot, take $A = 0.8$ m, $\phi = -\pi/2$, and $\omega = 5.0$ s^{-1}.

   (b) Now make a single plot that shows $x(t)$ together with both $v(t)/\omega$ and $a(t)/\omega^2$, where $v(t)$ is the velocity and $a(t)$ is the acceleration, as functions of t for the same range of t. (The factors of ω are included so that all of these quantities have units of distance.) If you can, make the $x(t)$ curve blue, the $v(t)/\omega$ curve green, and the $a(t)/\omega^2$ curve red. (For a hint on how to do this, see the syntax used to produce the figure on p. 8.)

   (c) How far out of phase are the $x(t)$ and the $v(t)/\omega$ curves with each other? What about the $x(t)$ and the $a(t)/\omega^2$ curves?

2. Consider

   (a) Use Mathematica to construct the state-space curve for a simple harmonic oscillator with an oscillator frequency $\omega = 3.0$ s^{-1} that begins from rest (no initial velocity) at $x = 0.50$ m.

   (b) Find the initial velocity for which the oscillator would trace out exactly the same phase-space trajectory if it had started at an initial position $x = -0.20$ m .

3. Consider an object that executes simple harmonic motion independently along two axes — for example, pendulum bob free to swing in both the x and y directions. The motion of the object would be given by

$$x(t) \; = \; A_1\cos(\omega_1 t + \phi) \; , \qquad y(t) \; = \; A_2\cos(\omega_2 t + \phi) \; .$$

   Make a parametric plot of y vs. x for the case $A_1 = A_2 = 1$, with $\phi_1 = \phi_2 = 0$, and $\omega_1 = 4\omega_2/3$. Make sure to set the range of t values so that you see the curve in its entirety. Make another plot with the $\omega_1 = 4\omega_2/7$ and all of the other parameters unchanged. These curves are called **Lissajous curves**. You can produce different Lissajous curves by varying the ratio ω_1/ω_2.

4. Consider the function
$$f(x) \;=\; \frac{1}{\sqrt{1+x^3/3}}\;.$$
   (a) Use Mathematica to determine the first four non-vanishing terms in the Taylor series for $f(x)$.
   (b) Make a plot which shows both the function $f(x)$ itself and a set of curves representing the Taylor-series approximations for $f(x)$ with the first non-vanishing term alone, with the first two non-vanishing terms, with the first three such terms, and with the first four such terms.

5. The Bessel function $J_0(t)$ is a set special functions which represents one of the solutions to the equation of motion for a particular kind of damped oscillator with a damping coefficient that gets weaker with time. In Mathematica, this function is represented by the command **BesselJ[0, t]**.
   (a) Plot $J_0(t)$ as a function of t on the range $0 \leq t \leq 50$. Try to convince yourself that the resulting curve is qualitatively what you'd expect the position function $x(t)$ for an oscillator with a damping coefficient that gets weaker with time. Find the corresponding velocity function $v(x)$ and plot it alongside $J_0(t)$.
   (b) Plot the state-space curve associated with this Bessel-function solution. Is the shape of the curve what you expect? Is the energy of the oscillator conserved?
   (c) The successive zeroes of $J_0(t)$ are given in Mathematica by the function **BesselJZero[0,n]**, where $n = 1$ yields the first zero, $n = 2$ yields the second zero, *etc.*. Use this information to determine the limiting value for the angular frequency of oscillation at very late times (note that this frequency is not a constant), assuming that t is in seconds.

Chapter 2

Differential Equations

Solving Differential Equations

In addition to all of the features described in the previous chapter, Mathematica also provides a number of tools for solving differential equations. Some of these tools are used for solving differential equations symbolically, meaning that they output the function Such more powerful, but they're only useful when a closed-form solution to the equation that you're trying to solve exists and is known. On the other hand, Mathematica also includes additional tools for obtaining accurate numerical estimates of what the solution to a differential equation looks like. These tools can be used even in cases in which no closed-form solution to the equation can be found.

We'll begin by looking at the tools Mathematica provides for solving differentia equations symbolically. The most important of these tools is the **DSolve** function. In order to see how this function is used, it's probably best to start with an example. In particular, we'll use **DSolve** to solve the equation of motion for a simple harmonic oscillator:

$$\frac{d^2x}{dt^2} \; = \; -\,\omega_0^2 x \ ,$$

where ω_0 is the (angular) frequency of oscillation. Here's the syntax:

```
In[40]:=  DSolve[D[x[t], {t, 2}] == -ω0^2*x[t], x[t], t]

Out[40]=  {{x(t) → c2 sin(t ω0) + c1 cos(t ω0)}}
```

Let's begin by focusing on the syntax that's being used the input line. First of all, we see that the **DSolve** function takes three arguments. The first of these arguments is the differential equation you're trying to solve, the second is the dependent variable — in this case, **x[t]** — expressed as a function of the independent variable **t**, and the third is the independent variable itself. There are two other important things to notice about the input syntax. The first is that a double equals sign == is used in the equation that's being fed to **DSolve** rather than a single equals sign. As we saw in the last chapter of the Mathematica primer, a single equals sign = is used to do things like assigning a value to a variable. By contrast, a double equals sign is used for formulating equations, expressing conditions in **If** statements, and other things of this nature. The second important thing to note about the way our equation is written is that our dependent variable is always expressed as a function of the independent variable. In other words, we always write **x[t]** rather than just **x** in this equation. This also applies to the definition of the dependent variable in the second argument of **DSolve**: here too, we write **x[t]** rather than **x**.

Now let's turn to interpreting the output that Mathematica returns when **DSolve** is called. First of all, note that the output comes enclosed by curly braces and that each element consists of the dependent variable followed by an arrow pointing to an expression. This entire object — including **x[t]**, the arrow, and the expression that follows it — is an example of what's called a "rule" in Mathematica syntax. A rule is essentially an instruction to replace every instance of the the expression on the left of the arrow with the expression on the right side of the arrow. For example,

```
In[41]:= a → 2 b

Out[41]= a → 2 b
```

is an instruction to replace every instance of the variable **a** with the expression **2b**. The syntax for applying a rule to an expression is as follows. Let's say we wanted to apply the above rule to the expression

```
In[46]:= 3 * a + 2

Out[46]= 3 a + 2
```

The syntax is

```
In[42]:= 3 * a + 2 /. a → 2 b

Out[42]= 6 b + 2
```

You can also define symbols in Mathematica to represent rules in the same way you can define symbols to represent numbers:

```
In[43]:= Seta2b = a → 2 b
         3 * a + 2 /. Seta2b

Out[43]= a → 2 b

Out[44]= 6 b + 2
```

As you may have already guessed, the expression on the right side of the arrow in the output produced by **DSolve** above is the general solution to the to the differential equation we called on this command to solve for us. The symbols **C[1]** and **C[2]** are the two undetermined coefficients that characterize the general solution to this second-order linear differential equation. There are other ways of writing this general solution, of course. The **DSolve** function may not always express the solution to a differential equation in the form that you'd want, so you'll often have to think a bit about how to re-express the result it gives you in a more familiar or canonical form.

This input syntax might seem a little cumbersome, but fortunately Mathematica does provide a useful shorthand for expressing derivatives with respect to the independent variable in ordinary differential equations. This shorthand is to write **x′[t]** rather than **D[x[t],t]** in order to express a first derivative, **x′′[t]** in order to express a second derivative, and so on. Since the independent variable is specified in the third argument of **DSolve**, Mathematica knows the variable with respect to which you're differentiating when you use this prime notation. Thus, when we write

```
In[47]:= DSolve[x''[t] == -ω0^2 * x[t], x[t], t]

Out[47]= {{x(t) → c2 sin(t ω0) + c1 cos(t ω0)}}
```

we get exactly the same output as when we used the full **D[x[t],{t,2}]** notation for the second derivative.

Now, for a slightly more complicated example, let's consider what happens if we add an additional term to this equation to represent the effect of damping forces which serve to dissipate the energy of the system to its surroundings as time goes on. In particular, we'll add a term which is proportional to the velocity **x′[t]** of the oscillator with a constant coefficient we'll call β. As we'll see later on, the damping term associated with air resistance acting on a sufficiently slowly moving object takes precisely this form. The resulting differential equation is

$$\frac{d^2x}{dt^2} + 2\beta\frac{dx}{dt} + \omega_0^2 x \;=\; 0 \;.$$

This way

```
In[48]:= DSolve[D[x[t], {t, 2}] + 2*β*D[x[t], t] + ω0^2*x[t] == 0,
          x[t], t]
```

Out[48]= $\left\{\left\{x(t) \to c_1\, e^{t\left(-\sqrt{\beta^2-\omega 0^2}-\beta\right)} + c_2\, e^{t\left(\sqrt{\beta^2-\omega 0^2}-\beta\right)}\right\}\right\}$

If you're familiar with the physics of damped harmonic oscillators, you may already have a lot of questions about this solution. There are three different kinds of behavior that a damped harmonic oscillator can exhibit, depending on how strong the damping is. When the damping is very slight, the oscillator oscillates back and forth around the equilibrium point just like an undamped oscillator, except that the amplitude of oscillation decreases over time. This is called "underdamped" motion. In the opposite regime, in which the damping is very strong, the oscillator simply settles back toward the equilibrium point without oscillating around it. This is called "overdamped" motion. At the transition point between these two regimes, we have what's called "critically damped" motion (which qualitatively resembles overdamped motion, but has some special properties). If you're familiar with the mathematical forms for the trajectory $x(t)$ of the oscillator associated with each of these three kinds of motion, you'll recognize the functional form of **x**[**t**] that **DSolve** returns as the form for $x(t)$ associated with the overdamped case.

So what about the solution underdamped and critically-damped cases? We haven't specified the values of β or ω_0 yet, so why did Mathematica output a solution of this form? These are important questions, but in order to answer them, we first need to be able to take the general solution that **DSolve** outputs and convert it into a function that we can evaluate for specific choices of these parameters and plot. Once we've done that, we'll be far better equipped to understand where the underdamped and critically-damped solutions went.

The first issue we must deal with is that the output of **DSolve** is the two sets of curly braces which enclose our rule for solving **x**[**t**]. As we learned in the last chapter of this Mathematica primer, curly braces are used in Mathematica to define lists — objects which consist of sets of elements (which can be numbers, functions, varibles, *etc.*) separated by commas. For example,

```
In[50]:= L1 = {1, -3, 7}

Out[50]= {1, -3, 7}
```

is a list of three numbers. We have already seen that many functions in Mathematica require certain of their arguments to be in the form of lists. However, lists are used for a wide variety of things in Mathematica: they're used to write vectors and matrices, to store sets of data, to define piecewise functions, and much more. The output of **DSolve** is also a list — in fact it is a nested list, because its one and only element is also a list. This sub-list also contains only a single element, which a rule for setting **x**[**t**] to the expression on the right side of the arrow.

The syntax for picking out an element of a list is as follows. For example, to get the second element of the three-element list **L1** I defined above, I would write

```
In[51]:= L1[[2]]

Out[51]= -3
```

The syntax for picking an element out of a nested list is very similar. For example, let's define the nested list **L2** to be

```
In[54]:= L2 = {{1, 8, 3}, {-2, 4, 6}, {3, 12, 5}}
```

Out[54]= $\begin{pmatrix} 1 & 8 & 3 \\ -2 & 4 & 6 \\ 3 & 12 & 5 \end{pmatrix}$

If I wanted to get the third element of a sub-list which is the second element of this list **L2** which I define below, I would write

```
In[99]:= L2[[2, 3]]

Out[99]= 6
```

It's worth remarking that Mathematica outputs this nested list is the form of a matrix. Indeed, matrices are yet another application of nested lists in Mathematica. We'll discuss this application of lists further in Chapter **??**

Thus, to extract the rule for setting **x[t]** to the solution for the damped harmonic oscillator from the nested list that **DSolve** produces as its output, I would want to get the first (and only) element of that list and then get the first (and only) element of the resulting sub-list:

```
In[52]:= DHOSolList =
          DSolve[D[x[t], {t, 2}] + 2 * β * D[x[t], t] + ω0 ^ 2 * x[t] ==
            0, x[t], t]
         DHOSol = DHOSolList[[1, 1]]
```

Out[52]= $\left\{\left\{x[t] \to e^{t\left(-\beta-\sqrt{\beta^2-\omega0^2}\right)} C[1] + e^{t\left(-\beta+\sqrt{\beta^2-\omega0^2}\right)} C[2]\right\}\right\}$

Out[53]= $x[t] \to e^{t\left(-\beta-\sqrt{\beta^2-\omega0^2}\right)} C[1] + e^{t\left(-\beta+\sqrt{\beta^2-\omega0^2}\right)} C[2]$

Here, I have defined the the symbol **DHOSol** to represent the rule for setting **x[t]** to the solution. In order to apply this rule to **x[t]**, I use the syntax that I introduced above for applying rules to variables:

```
In[102]:= x[t] /. DHOSol
```

Out[102]= $c_1\, e^{t\left(-\sqrt{\beta^2-\omega0^2}-\beta\right)} + c_2\, e^{t\left(\sqrt{\beta^2-\omega0^2}-\beta\right)}$

Moreover, I can also use this same syntax to assign values to the undetermined coefficients **C[1]** and **C[2]**. All I need to do is apply a rule for assigning values to each of these coefficients. In fact, I can do this in one step by defining a list of rules, like so

```
In[103]:= x[t] /. DHOSol /. {C[1] → 1, C[2] → 1}
```

Out[103]= $e^{t\left(-\sqrt{\beta^2-\omega0^2}-\beta\right)} + e^{t\left(\sqrt{\beta^2-\omega0^2}-\beta\right)}$

You can verify for yourself using relations from the lecture notes for this course that the example values I've chosen here for **C[1]** and **C[2]** correspond to the initial conditions $x_0 = 2$ and $v_0 = 0$.

The next step is to convert the resulting expression we get for **x[t]** into a function. We already know how to define functions (including functions of multiple variables) from the last chapter of this Mathematica primer, so it would seem that all we'd need to do in order to turn our solution into a function and then evaluate it for some particular choice of variables — say, $\beta = 0.5$, $\omega = 5$, and $t = 3$ — is the following:

```
In[104]:= DHOSolFunction[β_, ω0_, t_] :=
           x[t] /. DHOSol /. {C[1] → 1, C[2] → 1}
          DHOSolFunction[1, 5, 3]

Out[105]= x(3)
```

However, you can see from the second line that we didn't get the output we'd expected. What went wrong here? The problem is that there are certain default protocol in Mathematica about when to assign and when not to assign values to the variables that appear in certain kinds of expressions. However, there's an easy way to override those protocols. Indeed, there built-in function called **Evaluate** which simply instructs Mathematica to override them and assign the values anyhow. Thus, we need to write

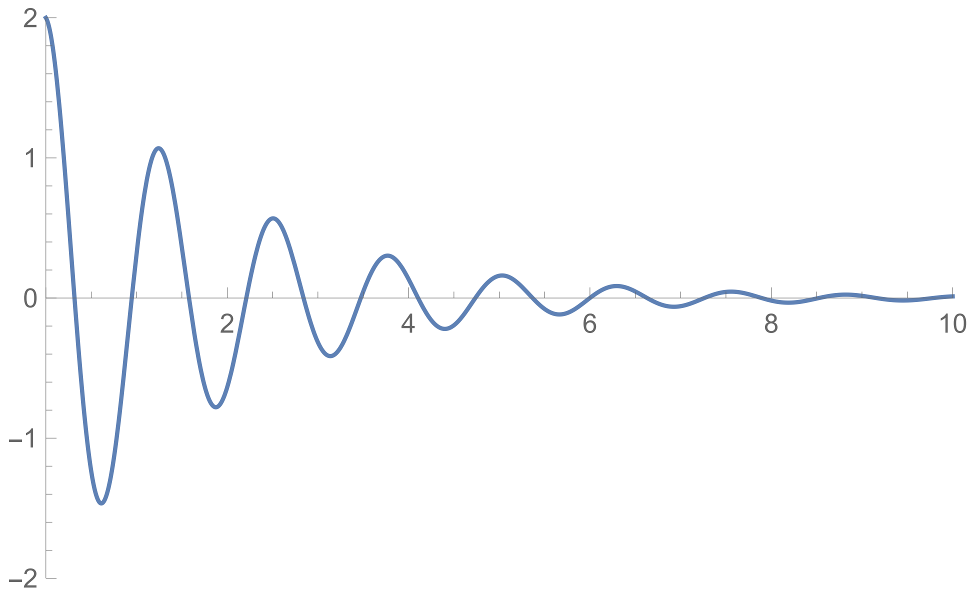
```
In[106]:= DHOSolFunction[β_, ω0_, t_] :=
            Evaluate[x[t] /. DHOSol /. {C[1] → 1, C[2] → 1}];
          DHOSolFunction[0.5, 5, 3]

Out[107]= -0.316262 + 0. i
```

Now when we call the function **DHOSolFunction** for numerical values of its arguments, Mathematica returns a number. It's expressed in the form of a complex number, as you can see, but since the imaginary part is zero, this number is purely real, as you'd expect.

Speaking of complex numbers, this might be a good time to revisit the question about what happened to the underdamped solutions for $x(t)$. You might have noticed that the values $\beta = 0.5$ and $\omega_0 = 5$ that I fed to our newly-defined **DHOSolFunction** function in the above example actually correspond to underdamped motion ($\omega_0 > \beta$) rather than overdamped motion ($\omega_0 < \beta$). Nevertheless, Mathematica was able to evaluate this function just fine. In fact, if we plot **DHOSolFunction** at a function of t for these values of β and ω_0, we get precisely the kind of $x(t)$ curve we'd expect for an underdamped oscillator:

```
In[108]:= Plot[DHOSolFunction[0.5, 5, t], {t, 0, 10},
           PlotRange → {{0, 10}, {-2, 2}}]
```

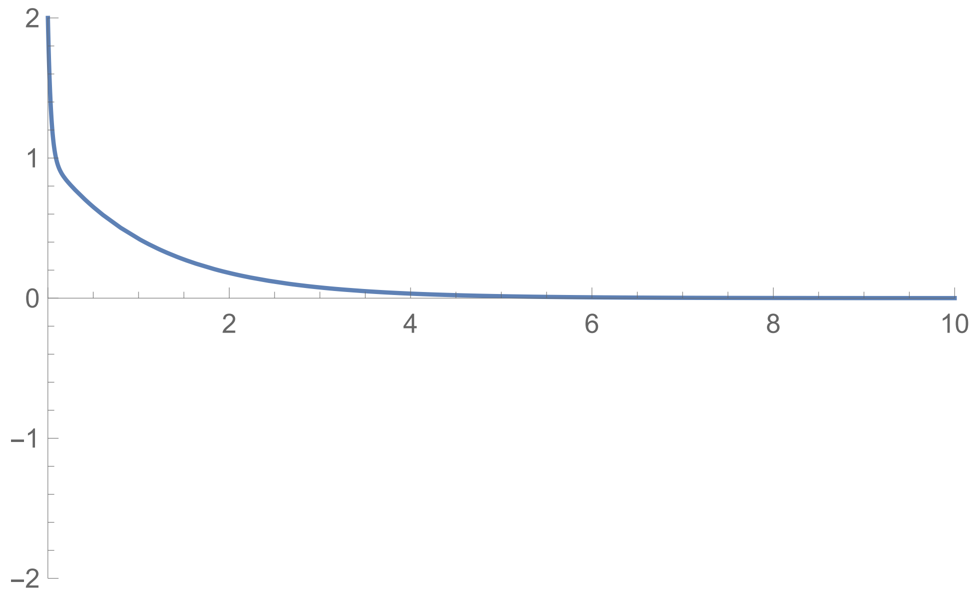
Out[108]=



Likewise, when we choose values for β and ω_0 that correspond to the overdamped case, we get a plot which looks like the sum of two falling exponentials:

```
In[109]:= Plot[DHOSolFunction[15, 5, t], {t, 0, 10},
           PlotRange → {{0, 10}, {-2, 2}}]
```

Out[109]=



The reason we get sensible results in both of these cases in that unlike a lot of other programming languages an numerical evaluation packages, Mathematica knows how to handle complex numbers. In the underdamped case, the roots $r_\pm = -\beta \pm \sqrt{\beta^2 - \omega^2}$ to the characteristic equation for the damped harmonic oscillator are complex, but that's no problem for Mathematica. The single rule that **DSolve** gives us is therefore valid for handling both the underdamped and overdamped cases.

For the critically-damped case, however, the situation is very different. Indeed, this case highlights on of the important caveats about using computational tools like Mathematica to evaluate differential equations for you. Recall that the general solution for the critically-damped case takes the form

$$x(t) \;=\; \big(B_1 + B_2 t\big)e^{-\beta t}\ ,$$

where B_1 and B_2 are the two undetermined coefficients. Let's define a Mathematics function that corresponds to this solution for the critically-damped case:

```
In[110]:= CritDampFunction[β_, ω0_, t_, B1_, B2_] :=
           (B1 + B2 * t) * Exp[-β * t]
```

We have also shown in the lecture notes that the undetermined coefficients B_1 and B_2 for critically-damped motion are related to the initial position x_0 and velocity v_0 of the oscillator at time $t = 0$ by

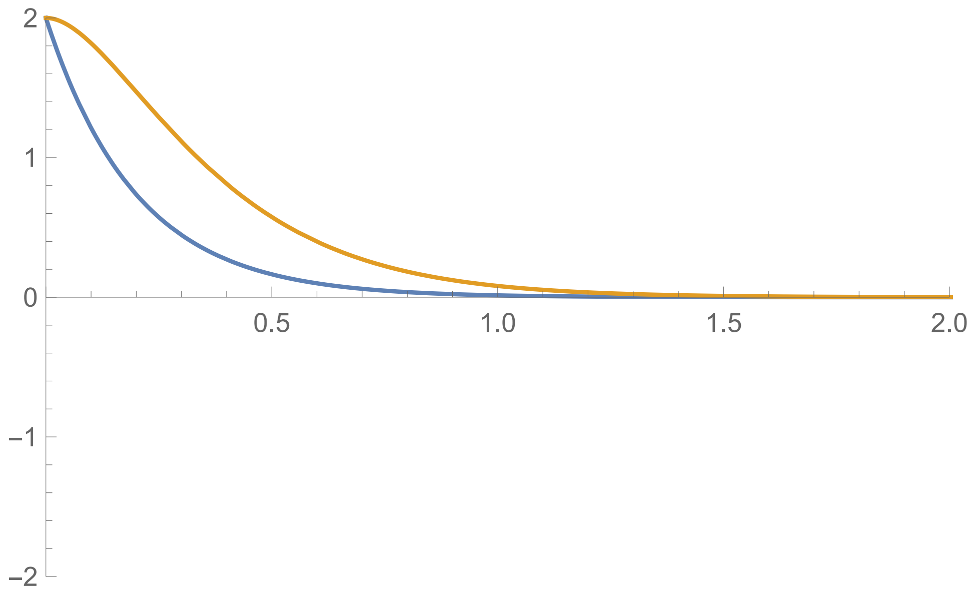
$$B_1 \;=\; x_0\ , \qquad B_2 = v_0 + \beta x_0\ .$$

Thus, the assignment **C[1] = 1** and **C[2] = 1** we have been making above corresponds to the assignment $B_1 = 2$ and $B_2 = 10$ for the critically-damped solution.

If **DSolve** is doing what it should be doing, the function **DHOSolFunction** that we obtained from the output of **DSolve** and the function that we have defined in **CritDampFunction** should give the same results for this assignment of B_1 and B_2. However, when we plot these two functions side by side, we see that these two functions do *not* yield the same results:

```
In[111]:= Plot[{DHOSolFunction[5, 5, t],
             CritDampFunction[5, 5, t, 2, 10]}, {t, 0, 10},
           PlotRange → {{0, 2}, {-2, 2}}]
```

Out[111]=



In this case, something clearly really is going wrong. We can also see evidence of this problem if we evaluate **DHOSolFunction** with undefined variables rather than numerical values for its three arguments. Indeed, when we do this for the special case where $\omega_0 = \beta$, the function returns

```
In[115]:= DHOSolFunction[β, β, t]
```

Out[115]= $2\, e^{\beta\,(-t)}$

There should also be a term that looks like $te^{-\beta t}$ with a non-vanishing coefficient, but this term does not appear. In other words, we have discovered an important limitation in how Mathematica solves differential equations in symbolic form. We have seen that Mathematica yields the correct expression for most values you assign to the symbols you use to parametrize the differential equation you feed to **DSolve**. However, it often "misses" special cases like the critically-damped case in which might arise for particular special choices of values. On the other hand, if we specify that $\omega_0 = \beta$ ahead of time, before we plugged our symbolic equation into **DSolve**, we get the correct functional form for $x(t)$ for the critically-damped case:

```
In[116]:= DSolve[D[x[t], {t, 2}] + 2*β*D[x[t], t] + β^2*x[t] == 0,
           x[t], t]
```

Out[116]= $\left\{\left\{x(t) \to c_1\, e^{\beta\,(-t)} + c_2\, t\, e^{\beta\,(-t)}\right\}\right\}$

Again, the lesson is that one must always be careful in interpreting the results that you get out of a computer program — even a sophisticated symbolic-evaluation package like Mathematica.

Piecewise Functions

As we'll see later in this course, certain differential equations — for example, the equation of motion for a harmonic oscillator with damping caused by friction with a surface — can most easily be solved in terms of a piecewise solution. A piecewise solution to a differential equation is a solution in which the solution $x(t)$ takes different functional forms for different ranges of the variable t. These different functions are "sewn together" at the transition points between the different ranges by boundary conditions. For example, $x(t)$

must be continuous across each transition point, since the oscillator can't suddenly "teleport" from one position to another.

Fortunately, Mathematica is equipped to deal with piecewise functions. In order to define a piecewise function, we can use the built-in function **Piecewise**. One application of piecewise functions that we'll deal with later on in this course is to describe the motion of an oscillator which is damped by friction — for example, a mass m attached to a spring sliding along a tabletop with coefficient of kinetic friction μ_k. We'll defer the discussion of how to set up and solve this equation of motion until later in th esemester, but the important thing for our present purposes is that the solution $x(t)$ to the equation of motion for such an oscillator has the form

$$x(t) = \begin{cases} (x_0 - d)\cos(\omega t) + d & 0 \le t < T/2 \\ (x_0 - 3d)\cos(\omega t) - d & T/2 \le t < T \\ (x_0 - 5d)\cos(\omega t) + d & T \le t < 3T/2 \\ (x_0 - 7d)\cos(\omega t) - d & 3T/2 \le t < 2T \\ \cdots & \cdots , \end{cases} \tag{2.1}$$

where $d = \mu_k mg/k$. Let's say that we wanted to define a function **xFricDamp** in Mathematica which represents this piecewise function. The syntax for doing this is

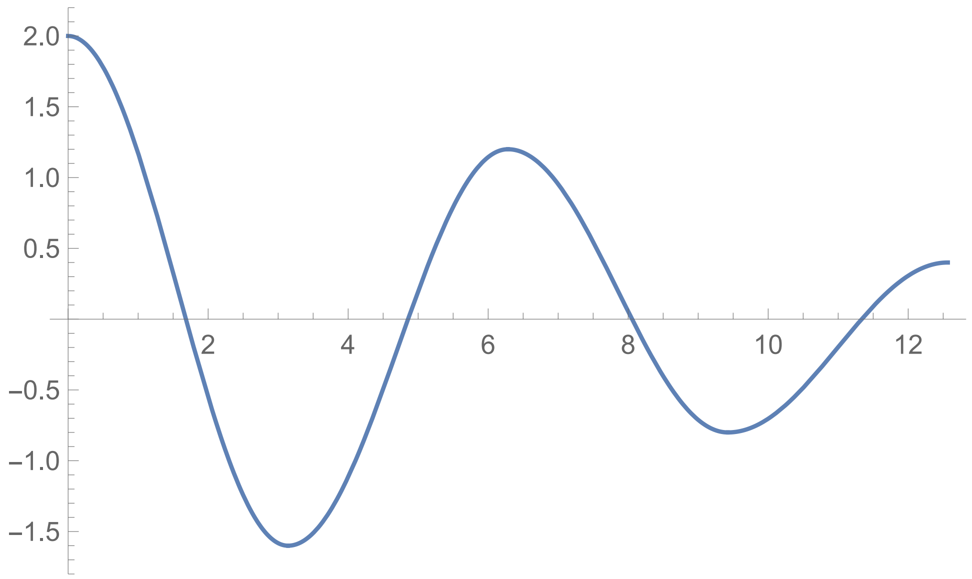
```
In[117]:= xFricDamp[x0_, d_, ω_, t_] :=
            Piecewise[{{(x0 - d) * Cos[ω * t] + d, 0 ≤ t < Pi / ω},
              {(x0 - 3 * d) * Cos[ω * t] - d, Pi / ω ≤ t < 2 * Pi / ω},
              {(x0 - 5 * d) * Cos[ω * t] + d, 2 * Pi / ω ≤ t < 3 * Pi / ω},
              {(x0 - 7 * d) * Cos[ω * t] - d, 3 * Pi / ω ≤ t < 4 * Pi / ω}}]
```

Note that the function **Piecewise** takes one argument, which consists of a list of several sub-lists. Each sub-list includes precisely two elements: the second element specifies a range of t values and the first element gives the functional form for $x(t)$ within that range.

For most purposes, piecewise functions in Mathematica can be treated like regular functions. For example, one can plot them:

```
In[120]:= ωin = 1;
          Plot[xFricDamp[2, 0.2, ωin, t], {t, 0, 4 * Pi / ωin}]
```

Out[121]=



One can also take derivatives of them. For example, to get the velocity $v(t)$ of the frictionally-damped oscillator in the above example, we would enter

```
In[122]:= D[xFricDamp[x0, d, ω, t], t]
```

Out[122]=
$$\begin{cases} \omega\,(-(\mathrm{x0}-d))\sin(t\,\omega) & t \geq 0 \land t - \frac{\pi}{\omega} < 0 \\ \omega\,(-(\mathrm{x0}-3\,d))\sin(t\,\omega) & t - \frac{\pi}{\omega} \geq 0 \land t - \frac{2\pi}{\omega} < 0 \\ \omega\,(-(\mathrm{x0}-5\,d))\sin(t\,\omega) & t - \frac{2\pi}{\omega} \geq 0 \land t - \frac{3\pi}{\omega} < 0 \\ \omega\,(-(\mathrm{x0}-7\,d))\sin(t\,\omega) & t - \frac{3\pi}{\omega} \geq 0 \land t - \frac{4\pi}{\omega} < 0 \end{cases}$$

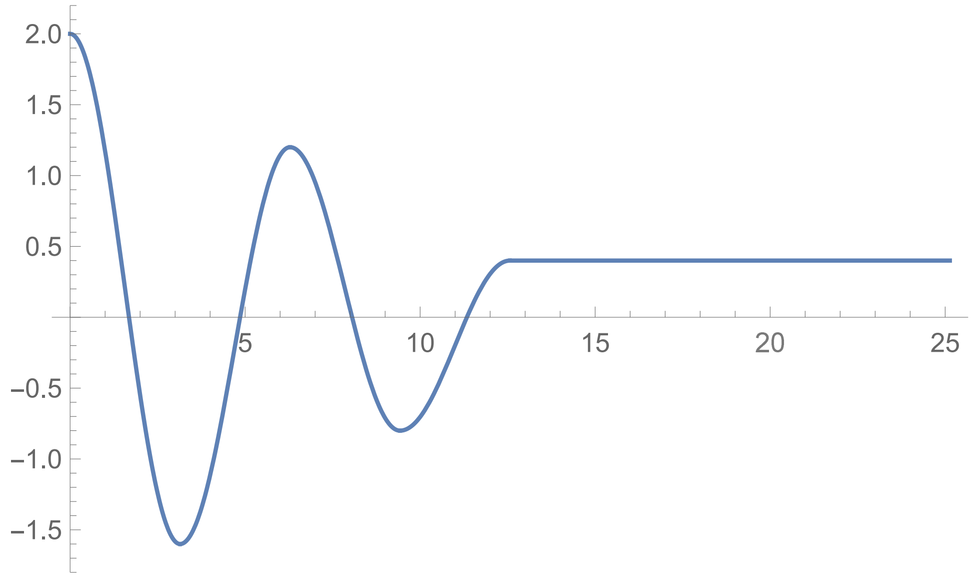
Indeed, the first four lines of this expression are simply the derivatives of $x(t)$ within the four time intervals we've specified. The last line may look a bit strange, but all it is is a default command that Mathematica generates which assigns the value $v(t) = 0$ to $v(t)$ for all values of t which lie outside the intervals we've specified. We can change the default value Mathematica assigns to a piecewise function outside the intervals we've explicitly specified by adding a second, optional argument to **Piecewise**. This argument simple consists of the value we want our function to take outside those intervals. For example, let's say that we knew that the force of static friction will cause our oscillator to get "stuck" at the value $(x_0 - 8d)$ once t reaches $2T$. To build this behavior into our function **xFricDamp**, we could write

```
In[123]:= xFricDamp[x0_, d_, ω_, t_] :=
            Piecewise[{{(x0 - d) * Cos[ω * t] + d, 0 ≤ t < Pi / ω},
               {(x0 - 3 * d) * Cos[ω * t] - d, Pi / ω ≤ t < 2 * Pi / ω},
               {(x0 - 5 * d) * Cos[ω * t] + d, 2 * Pi / ω ≤ t < 3 * Pi / ω},
               {(x0 - 7 * d) * Cos[ω * t] - d, 3 * Pi / ω ≤ t < 4 * Pi / ω}},
             (x0 - 8 * d)]
```

Now, when we plot $x(t)$, we get

```
In[124]:= ωin = 1;
          Plot[xFricDamp[2, 0.2, ωin, t], {t, 0, 8 * Pi / ωin}]
```

Out[125]=



An Introduction to Tables and Lists

Defining four “pieces” of a piecewise function in Mathematica is no problem, but defining too many more than that would be extremely tedious. Fortunately, Mathematica has tools for creating lists which make this task far less time-consuming. Perhaps the most useful of these tools is the **Table** function. To get a sense of how **Table** works, it’s best to start with an example. Let’s say we wanted to generate a list of the squares of the integers from 1 to 10. The syntax for this is

```
In[126]:= Table[n^2, {n, 1, 10}]

Out[126]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

The symbol **n** in this construction functions like an iterator. The second argument of **Table** is a three-element list which specifies the name of this iterator variable, the initial value n_i we want this variable to take, and the final value to which we want the variable. The first element in the list that **Table** generates as its output is simply the first argument of **Table** evaluated at the initial value of the iterator variable n. The next element in that list is the first argument of **Table** evaluated at $n = n_i + 1$, and so on until n reaches the specified final value and the evaluation stops.

In the above example, the first argument of **Table** was a simple function of the iterator variable n, but this argument can take other forms as well. For example, we could set the first argument of **Table** to be a list. Thus, we could generate a set of ordered pairs of integers and their squares by entering

```
In[127]:= Table[{n, n^2}, {n, 1, 10}]
```

Out[127]= $$\begin{pmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \\ 5 & 25 \\ 6 & 36 \\ 7 & 49 \\ 8 & 64 \\ 9 & 81 \\ 10 & 100 \end{pmatrix}$$

We can use this same syntax to create a set of “pieces” to use in defining a piecewise function without too much tedium or hassle. For example, we can define a piecewise solution for the frictionally-damped harmonic oscillator that extends all the way out to $t = 4T$ by first using **Table** to generate a list of functional forms and intervals:

```
In[134]:= Pieces =
           Table[{(x0 - (2 * n - 1) * d) * Cos[ω * t] - (-1) ^ n * d,
             (n - 1) * Pi / ω ≤ t < n * Pi / ω}, {n, 1, 8}]
```

Out[134]=

$$\begin{pmatrix} d + (\mathrm{x0} - d)\cos(t\,\omega) & 0 \le t < \frac{\pi}{\omega} \\ (\mathrm{x0} - 3\,d)\cos(t\,\omega) - d & \frac{\pi}{\omega} \le t < \frac{2\pi}{\omega} \\ d + (\mathrm{x0} - 5\,d)\cos(t\,\omega) & \frac{2\pi}{\omega} \le t < \frac{3\pi}{\omega} \\ (\mathrm{x0} - 7\,d)\cos(t\,\omega) - d & \frac{3\pi}{\omega} \le t < \frac{4\pi}{\omega} \\ d + (\mathrm{x0} - 9\,d)\cos(t\,\omega) & \frac{4\pi}{\omega} \le t < \frac{5\pi}{\omega} \\ (\mathrm{x0} - 11\,d)\cos(t\,\omega) - d & \frac{5\pi}{\omega} \le t < \frac{6\pi}{\omega} \\ d + (\mathrm{x0} - 13\,d)\cos(t\,\omega) & \frac{6\pi}{\omega} \le t < \frac{7\pi}{\omega} \\ (\mathrm{x0} - 15\,d)\cos(t\,\omega) - d & \frac{7\pi}{\omega} \le t < \frac{8\pi}{\omega} \end{pmatrix}$$

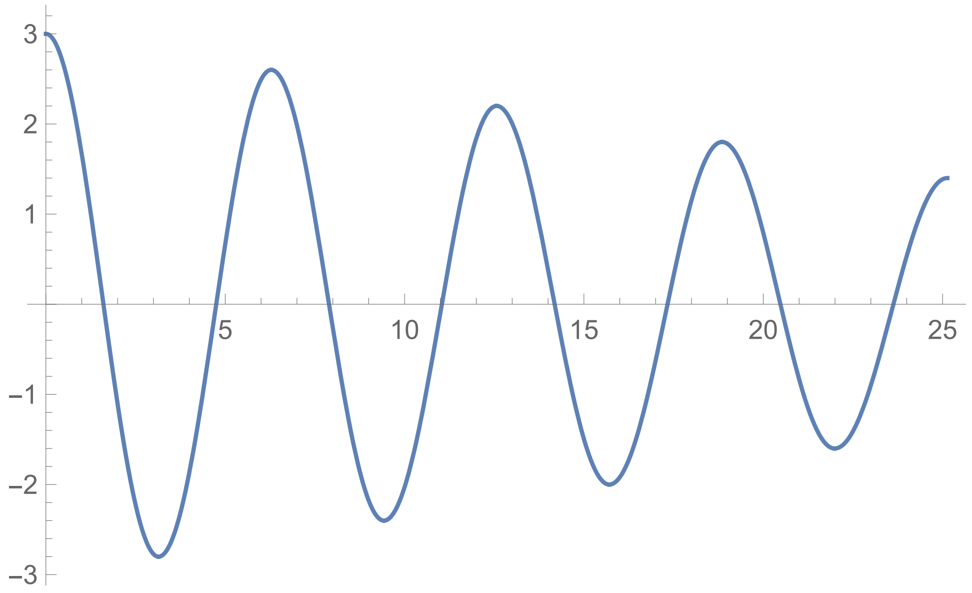
We can now use this list as the argument of **Piecewise** to redefine our piecewise function **xFricDamp** by entering

```
In[132]:= xFricDamp[x0_, d_, ω_, t_] := Evaluate[Piecewise[Pieces]]
```

Note that once again, we need to use **Evaluate** to force Mathematica to assign values to the symbols we've used in the list we fed to the **Piecewise** function. Now we can plot our piecewise solution for $x(t)$ all the way out to $t = 4T$:

```
In[136]:= ωin = 1;
          Plot[xFricDamp[3, 0.1, ωin, t], {t, 0, 8 * Pi / ωin}]
```

Out[137]=



Chapter 3

Linear Algebra

Vectors and Matrices as Lists

In Chapter 2, we saw how to define a list in Mathematica. Lists are used in Mathematica for a variety of things, and one of their most important applications is constructing vectors and matrices. For example, to define a three-component vector $\vec{\mathbf{v}} = 3\hat{\mathbf{i}} - 2\hat{\mathbf{i}} + 2\hat{\mathbf{k}}$, we would write

```
In[91]:=  v1 = {3, -1, 2}

Out[91]=  {3, -1, 2}
```

Likewise, we can represent a matrix **M** as a nested list. For example,

```
In[92]:=  M = {{1, 0, 0}, {2, 0, -1}, {-1, 0, 3}}
```

Out[92]=
$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & -1 \\ -1 & 0 & 3 \end{pmatrix}$$

Now that we've seen how to represent vectors and matrices in Mathematica, let's see what we can do with them. First of all, we might want to pick out a particular component of a vector or a particular entry in a matrix. This is simply a matter of picking an element out of the corresponding list or nested list — something we already learned how to do in Chapter 2. In addition, we can also use Mathematica to perform a variety of other vector operations. For example, if we define a second vector $\vec{\mathbf{w}}$ as follows

```
In[94]:=  w = {0, 2, 2}

Out[94]=  {0, 2, 2}
```

we can take the dot product of this new vector with $\vec{\mathbf{v}}$ using the following syntax

```
In[96]:=  v.w

Out[96]=  2
```

The cross product of two three-component vectors can be obtained using the built-in function `Cross`:

```
In[141]:=  Cross[v, w]

Out[141]=  {-6, -6, 6}
```

The same syntax that is used for the dot product of two vectors is also used for matrix multiplication.

For example, if I wanted to multiply the vector $\vec{\mathbf{v}}$ by our matrix $\mathbf{M}$ on the left in order to obtain another vector, I would write

```
In[97]:=  M.v

Out[97]=  {3, 4, 3}
```

Likewise, if I wanted to multiply the matrix $\mathbf{M}$ by itself in order to obtain another matrix, I would write

```
In[138]:=  M.M
```

Out[138]= $\begin{pmatrix} 1 & 0 & 0 \\ 3 & 0 & -3 \\ -4 & 0 & 9 \end{pmatrix}$

Mathematica includes a number of built-in functions useful for manipulating matrices. The most commonly used functions of this sort are listed below.

`Conjugate[x]`	Complex conjugate of a matrix (or a number)
`Transpose[x]`	Transpose of a matrix
`ConjugateTranspose[x]`	Conjugate transpose (*i.e.*, Hermitian Conjugate)
`Det[x]`	Determinant
`Tr[x]`	Trace
`Eigenvalues[x]`	Eigenvalues
`Eigenvectors[x]`	Eigenvectors

The output of most of these functions is fairly self-explanatory; however, the output of both `Eigenvalues` and `Eigenvectors` requires some additional clarification. The function `Eigenvalues` returns a list whose elements are the eigenvalues of the (square) matrix on which it acts. For example,

```
In[145]:=  Eigenvalues[M]

Out[145]=  {3, 1, 0}
```

The function `Eigenvectors` returns a list whose elements are the eigenvectors of the matrix on which it acts. Since each eigenvector is itself a list, the output of `Eigenvectors` is a nested list. By default, Mathematica displays this nested list as a matrix. For example, the eigenvectors of our matrix $\mathbf{M}$ are displayed as follows

```
In[147]:=  Eigenvectors[M]
```

Out[147]= $\begin{pmatrix} 0 & -1 & 3 \\ 2 & 3 & 1 \\ 0 & 1 & 0 \end{pmatrix}$

Each row of this output matrix corresponds to a different eigenvector. The set of numbers appearing in a given row are the components of that eigenvector. If we want to pick out a single eigenvector from this list, we can do so in the usual way. For example, the three individual eigenvectors of M are M as

```
In[152]:= evec1 = Eigenvectors[M][[1]]
          evec2 = Eigenvectors[M][[2]]
          evec3 = Eigenvectors[M][[3]]

Out[152]= {0, -1, 3}

Out[153]= {2, 3, 1}

Out[154]= {0, 1, 0}
```

One important caveat about the `Eigenvectors` function is that the eigenvalues it outputs are in general not normalized:

```
In[156]:= evec1.evec1

Out[156]= 10
```

Thus, if you want a set of normalized eigenvectors for a given matrix, you're going to have to normalize them yourself. For example, if I want to normalize the first eigenvector $\vec{\mathbf{e}}_1$ from the list of eigenvectors of M above, I would write

```
In[164]:= evec1norm = evec1 / Sqrt[evec1.evec1]
```

Out[164]= $\left\{0, -\frac{1}{\sqrt{10}}, \frac{3}{\sqrt{10}}\right\}$

Indeed, we can easily verify that this new eigenvector is properly normalized:

```
In[165]:= evec1norm.evec1norm

Out[165]= 1
```