HANDS-ON START TO WOLFRAM *MATHEMATICA*®

and Programming with the Wolfram Language™

Cliff Hastings *Kelvin Mischo* *Michael Morrison*

Wolfram Media
Champaign

Table of Contents

CHAPTER 7

Creating Interactive Models with a Single Command

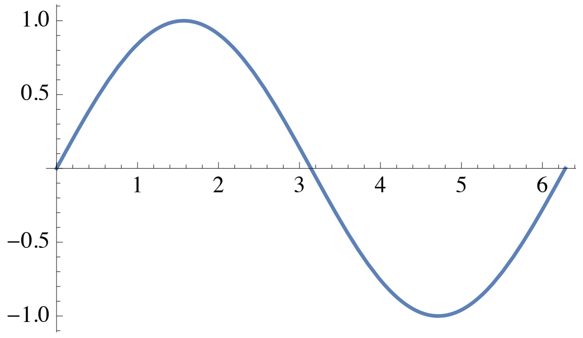
Introduction

One of the most exciting features of *Mathematica* is the ability to create interactive models with a single command called **Manipulate**. The core idea of **Manipulate** is very simple: wrap it around an existing expression and introduce some parameters; *Mathematica* does the rest in terms of creating a useful interface for exploring what happens when those parameters are manipulated. This single command is a powerful tool for learning and teaching about phenomena and for creating models and simulations to support research activities.

Building a First Model

A common workflow is to start with something static, such as a plot, and then to make it interactive using **Manipulate**. Take the following plot as an example, which plots $\sin(x)$ from 0 to 2π.

```
Plot[Sin[x], {x, 0, 2 π}]
```



The goal may be to compare the curve of $\sin(x)$ with the curve of $\sin(2x)$, the curve of $\sin(3x)$, and so on. In other words, to examine the behavior of $\sin(f\,x)$ when f is varied among a large quantity of numbers. **Manipulate** provides an easy way to perform this investigation by constructing an interactive model to explore this behavior.

To begin, it is important to know that using **Manipulate** requires three components:

1. **Manipulate** command
2. Expression to manipulate by changing certain parameters
3. Parameter specifications

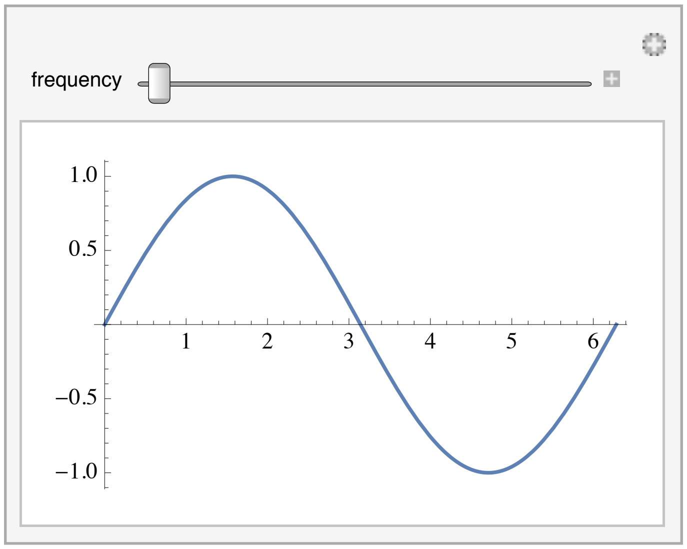
An easy way to keep track of these components is to write commands involving **Manipulate** as follows.

```
Manipulate[
  expression to manipulate,
  parameter specifications]
```

This approach keeps each component on a separate line and provides an easy way to keep track of each separate component.

For the example introduced above, the **Manipulate** command might be as follows.

```
Manipulate[
   Plot[Sin[frequency*x], {x, 0, 2 π}],
   {frequency, 1, 5}]
```



The result is an interactive model with a slider bar that can be clicked and dragged to interactively explore what happens as the value of **frequency** is changed. This specific model can be quite useful for explaining concepts of periodicity and frequency and was built from a single line of code—pretty impressive, and a representative example of the power of **Manipulate**.

The plus icon immediately to the right of the slider bar can be clicked to open an Animation Controls menu for that controller. Animation Controls can be used to animate the model, incrementally step through different values for the parameter, or assign a particular value to the parameter through the use of an input field.

You do not have to follow this multiline convention; you could put a **Manipulate** command on a single line, like:

```
Manipulate[Plot[Sin[f*x],{x,0,2π}],{f,1,5}]
```

To some, it reads more cleanly to have the command on one line; to others, having the components on different lines makes the code more readable. Choose the style that makes the most sense to you.

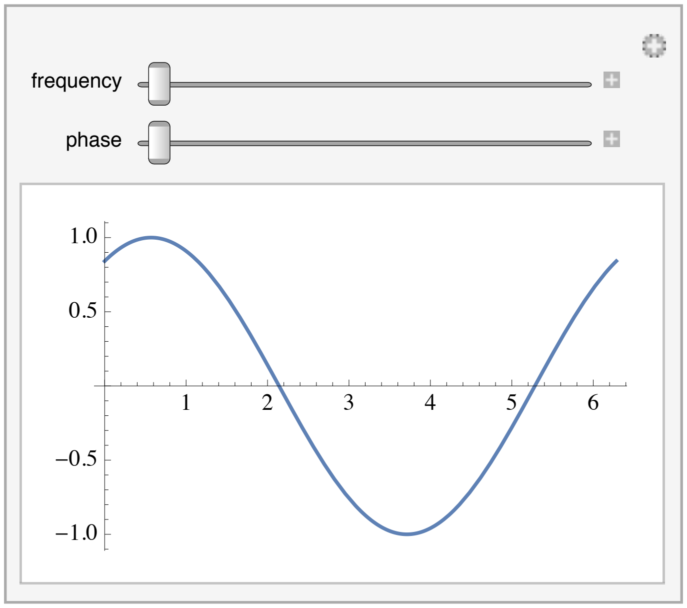
Building Models with Multiple Controls

Manipulate can be used to construct interactive models with an arbitrary number of controllers. To control a model with multiple parameters, simply introduce the new parameters and their corresponding parameter specifications. With two parameters, the basic outline changes to the following.

Manipulate[
 expression to manipulate,
 first parameter specifications,
 second parameter specifications]

The previous example can be expanded by introducing a new parameter, **phase**, along with a range of values for the minimum and maximum of this new parameter. *Mathematica* will automatically create separate controllers for each parameter and label them accordingly.

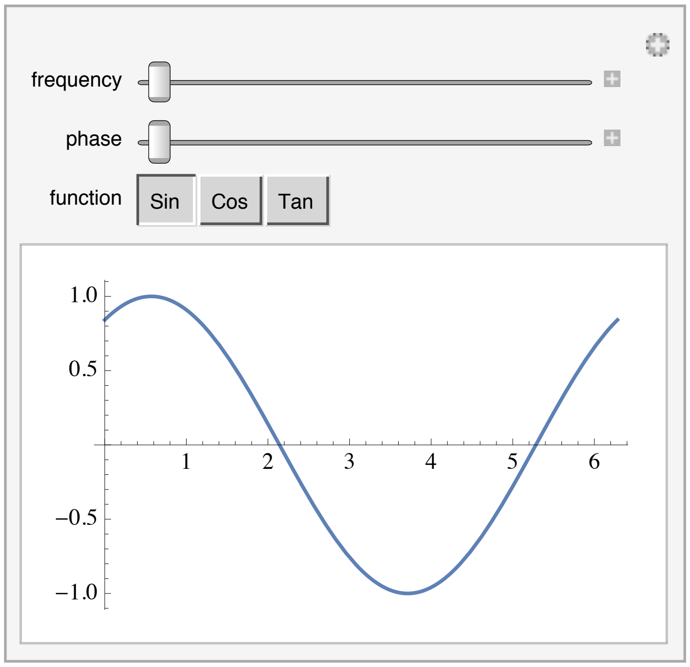
```
Manipulate[
  Plot[Sin[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10}]
```



Manipulate can be used to give parameters a list of discrete choices instead of a continuous range for their values. For example, the **Sin** command can be replaced by a new parameter called **function**, and then a list of choices can be given as the parameter specification for **function**.

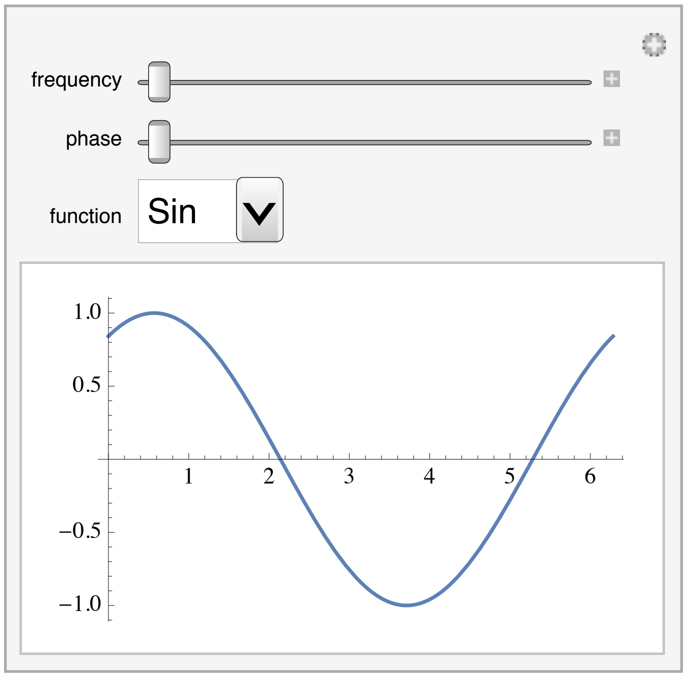
Since curly braces are used to denote lists, this will create a parameter specification with a nested list: the outermost list contains the parameter name and the specification, and the specification itself is a list that contains discrete choices—in this case, **Sin**, **Cos**, and **Tan**—for the parameter to assume.

```
Manipulate[
  Plot[function[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10},
  {function, {Sin, Cos, Tan}}]
```

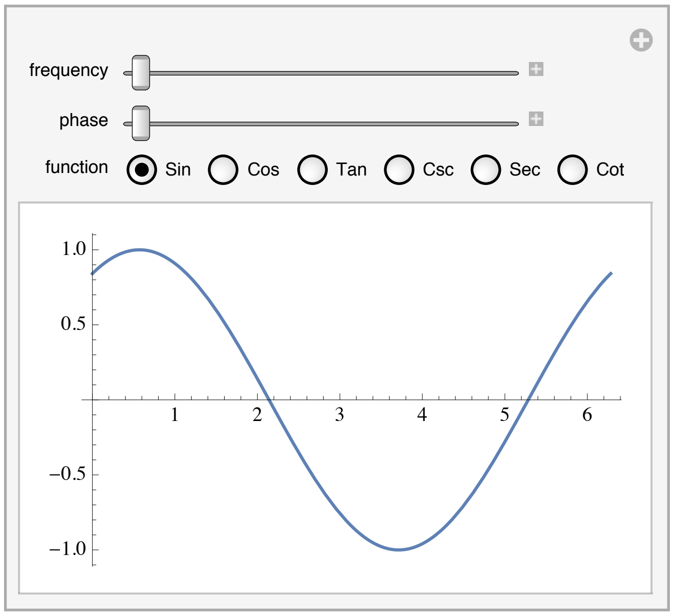


Mathematica has built-in heuristics to select appropriate controller types based on the parameter specifications that have been given. For example, giving a long list of choices causes **Manipulate** to display the controller as a drop-down menu instead of a list of buttons.

```
Manipulate[
  Plot[function[frequency*x + phase], {x, 0, 2 π}],
  {frequency, 1, 5},
  {phase, 1, 10},
  {function, {Sin, Cos, Tan, Csc, Sec, Cot}}]
```

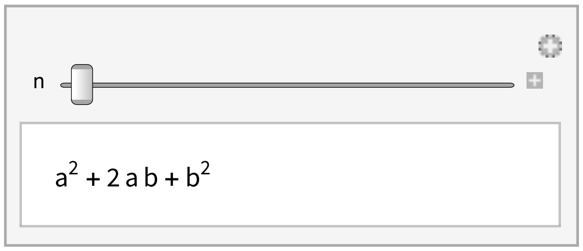


Like most everything in *Mathematica*, the output from commands can be customized through the use of options. If you want to force *Mathematica* to use a particular control type, the **ControlType** option can be used with values such as **Setter**, **Slider**, and **RadioButtonBar**. For example, if you add **ControlType → RadioButtonBar** between the two closing curly braces in the last parameter specification in the preceding example, *Mathematica* will create a row of radio buttons to set the value of **function** instead of giving you a drop-down menu.



The **Manipulate** command is not restricted to graphical manipulation and can be used with any *Mathematica* expression. For example, symbolic expressions can be manipulated just as easily as graphical expressions.

```
Manipulate[
  Expand[(a + b)^n],
  {n, 2, 10, 1}]
```



$a^2 + 2\,a\,b + b^2$

In the preceding example, the range **{n, 2, 10, 1}** was used to restrict the values of **n** to be from 1 to 250 in increments of 1, since exponentiation is not defined for noninteger values.

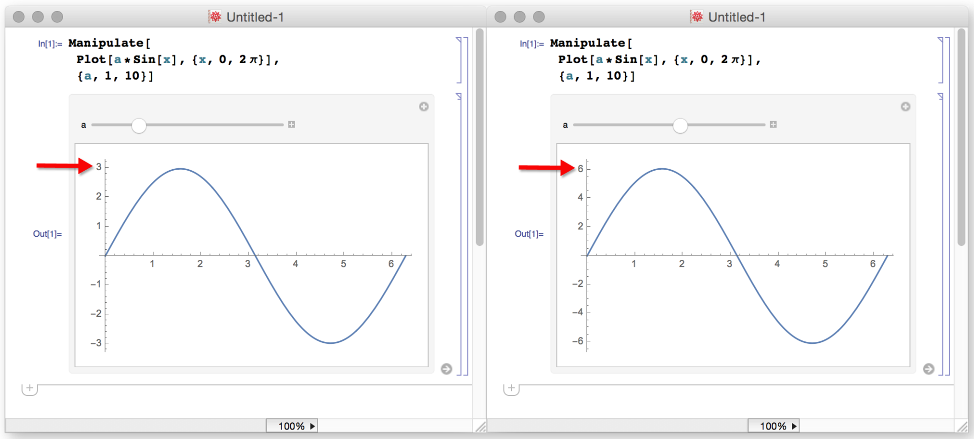
Some Tips for Creating Useful Models

The default results returned by **Manipulate** are generally very useful and do not require any special customization. However, there are a few important points to be aware of, so we will discuss them here in order to help you avoid potential problems.

The Importance of PlotRange

The default behavior of commands like **Plot** is to automatically choose an appropriate viewing window unless a specific range is given. This means that when **Manipulate** is used to change the value of a parameter, which has a resulting effect of changing the appearance of a plot, the plot will immediately be redrawn with a new viewing window. The end result is that manipulating a parameter may appear to change the axes for the plot rather than the plot itself.

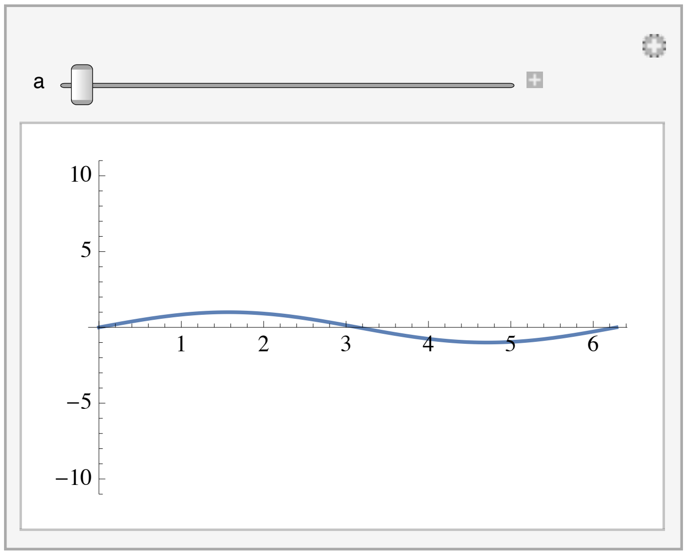
The following screen shot shows an example of this behavior. On the left, the value of the parameter **a** is set to 3, and the plot axes are automatically chosen to fully display the behavior of the plot. On the right, the value of **a** is set to 6, and the plot is drawn accordingly.


```
In[1]:= Manipulate[
          Plot[a * Sin[x], {x, 0, 2 π}],
          {a, 1, 10}]
```

This behavior can be avoided by specifying an explicit range to plot over. This can be accomplished by using the **PlotRange** option for the **Plot** command, which forces the plot to be drawn with the specific plot range the user provides. **PlotRange** takes a list as its argument (remember: lists are enclosed by curly braces), where the first element of the list is the minimum value for the plot range, and the second element of the list is the maximum value for the plot range.

The arrow (→) in the **PlotRange** option is constructed by using the hyphen (**-**) and the greater-than symbol (**>**), which *Mathematica* then formats into the arrow.

```
Manipulate[
 Plot[a*Sin[x], {x, 0, 2 π},
  PlotRange → {-11, 11}],
 {a, 1, 10}]
```



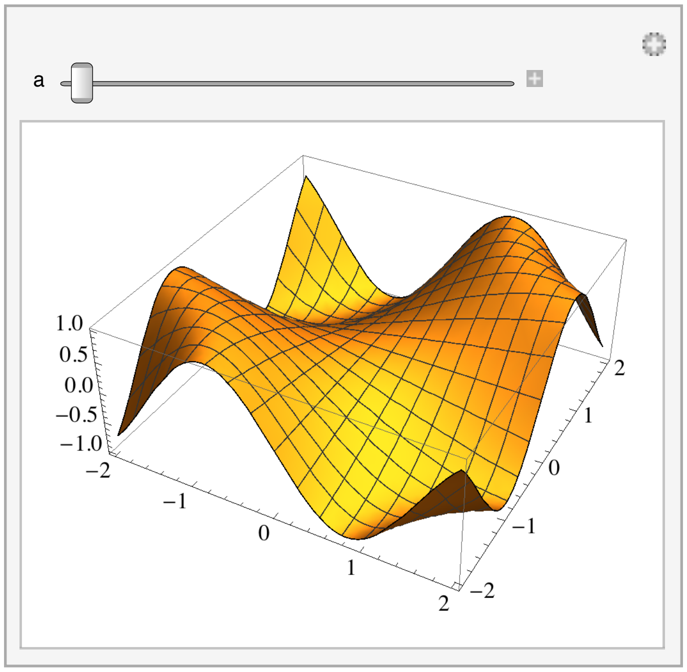
In the preceding example, the **Plot** function now spans two lines as a result of adding the **PlotRange** option. Notice how the **PlotRange** line is nicely indented to show that it is part of the **Plot** statement, while the list with the amplitude parameter is indented to show that it is an argument that belongs with the **Manipulate** command. You should experiment with deleting and adding extra line breaks like this based on your preference for how the code looks.

Since the plot range is now fixed, adjusting **a** appears to stretch or flatten the plot, which may be the desired behavior for this model to show.

Optimizing Performance for 3D Graphics

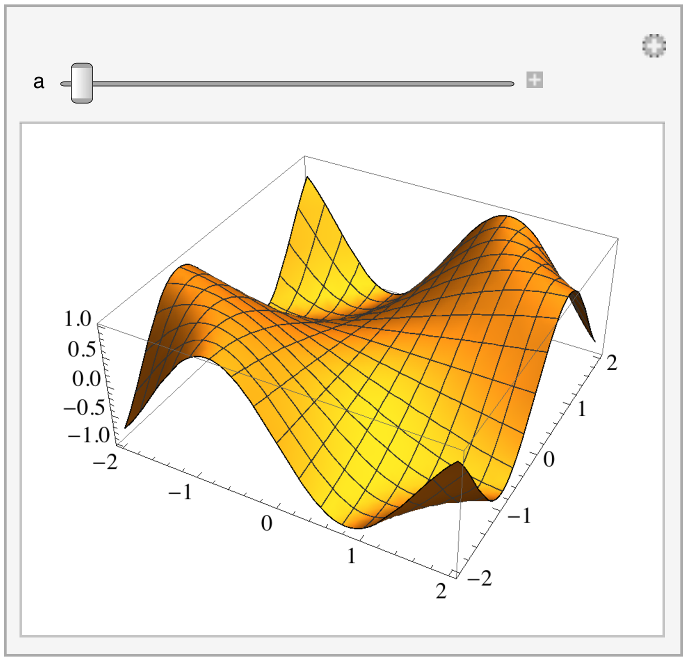
When 3D graphics are manipulated with controllers like slider bars, they may appear jagged while the controllers are being moved, and then smooth again when the controllers are released. The following example shows this behavior in action.

```
Manipulate[
  Plot3D[Sin[a x y], {x, -2, 2}, {y, -2, 2}],
  {a, 1, 5}]
```



Mathematica's default behavior is to optimize the performance while the controller is being moved, and then to optimize the appearance once the controller is released. This allows a fast interaction between users and the controllers, and nicely rendered results when finished. However, if rendering is more important than fast interaction, then the use of options like **PerformanceGoal** can be handy.

```
Manipulate[
  Plot3D[Sin[a x y], {x, −2, 2}, {y, −2, 2}, PerformanceGoal → "Quality"],
  {a, 1, 5}]
```



Now when the slider bar is dragged, the appearance of the plot remains smooth. The trade-off is that the slider bar may be slightly less responsive than it was in the preceding example.

Labeling Controllers and Displaying Current Values

Manipulate creates a unique controller for each parameter that can be manipulated. By default, *Mathematica* will use the name of the parameter when it labels its corresponding controller, so if the parameter is named **frequency**, then "frequency" is what the label for the controller will say.

There are times, though, when it is desirable to name the parameter one thing and to have the controller label display something else. A user might do this to save on keystrokes: use a short variable name, like **f**, for a parameter, but then label the control for **f** with something different, like "frequency," to improve readability of the model.

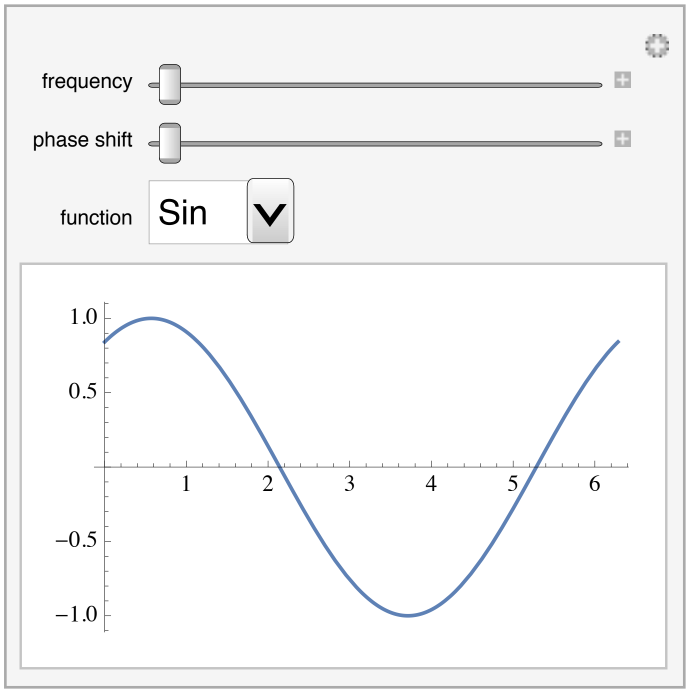
Labeling is also useful in situations where the label is comprised of multiple words. Since a parameter in *Mathematica* has to be a single symbol without spaces, a parameter cannot be named something like **phase shift**. However, a parameter could be named **ps**, and then the label corresponding to the controller for **ps** could be given as "phase shift."

To label a controller, a set of nested braces is used in the parameter specification, and values are entered as follows.

{{parameter, initial value, "parameter label"}, minimum, maximum}

Using this idea, an example from earlier in this chapter could be modified to use different parameter names and labels for each of the controllers.

```
Manipulate[
  Plot[fn[f*x + ps], {x, 0, 2 π}],
  {{f, 1, "frequency"}, 1, 5},
  {{ps, 1, "phase shift"}, 1, 10},
  {{fn, Sin, "function"}, {Sin, Cos, Tan, Csc, Sec, Cot}}]
```

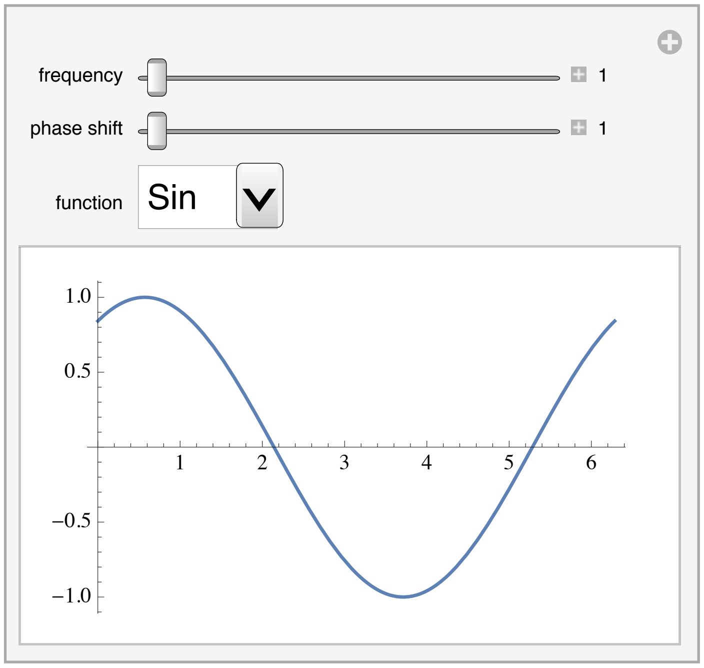


The labels appear to the left of each controller, and the actual names of the parameters—in this case, **f**, **ps**, and **fn**—are not visible in the output at all.

You do not have to make the initial value of the parameter the same as the lower bound of the controller; you can set the initial value to be, say, 3 for a controller that ranges from 1 to 5.

Another useful option to set for the controllers is **Appearance→"Labeled"**, which will display the current value of the parameter to the right of its Animation Controls button. (There is no need to set this option for the **fn** parameter, since the function name is already displayed *within* the controller as part of the buttons.)

```
Manipulate[
  Plot[fn[f*x + ps], {x, 0, 2 π}],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"},
  {{ps, 1, "phase shift"}, 1, 10, Appearance → "Labeled"},
  {{fn, Sin, "function"}, {Sin, Cos, Tan, Csc, Sec, Cot}}]
```

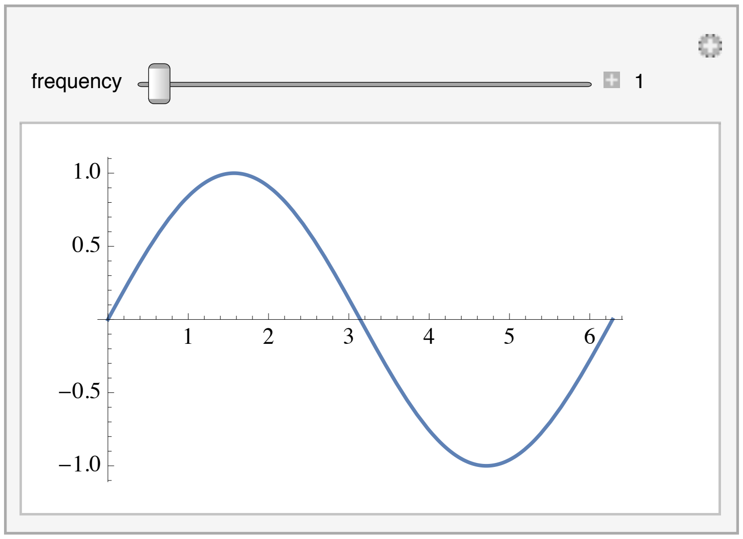


Creating an Interactive Plot Label

While labeling individual controllers in a **Manipulate** can be useful, it can also be desirable to create an interactive plot label that takes all of these labels into consideration and prints a single expression, like the equation of the function being graphed.

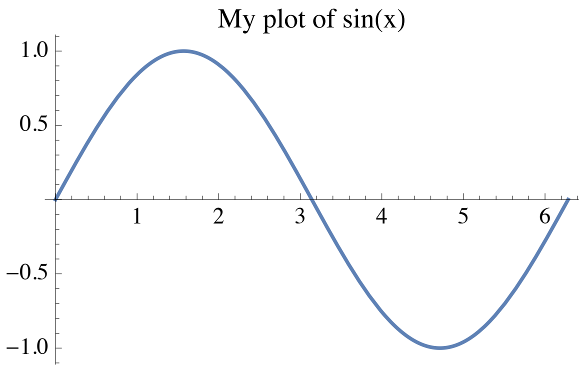
The following example plots **Sin[f*x]**, where **f** is a manipulable parameter. The controller for **f** uses the **Appearance→"Labeled"** option setting to print its values to the right of the controller, which is helpful, but the user is still required to examine the code to ascertain exactly what function is being plotted.

```
Manipulate[
  Plot[Sin[f*x], {x, 0, 2 π}],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"}]
```



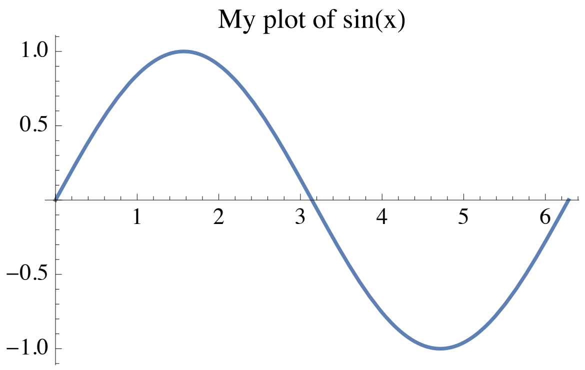
Creating an interactive plot label can make the function being plotted more obvious. First, a quick explanation of the **PlotLabel** option is necessary. **PlotLabel** is an option for **Plot** (and other plotting commands) that prints a label at the top of the plot. **PlotLabel** expects a string to be passed as its option setting. A string in *Mathematica* is enclosed with quotation marks.

```
Plot[Sin[x], {x, 0, 2 π}, PlotLabel → "My plot of sin(x)"]
```



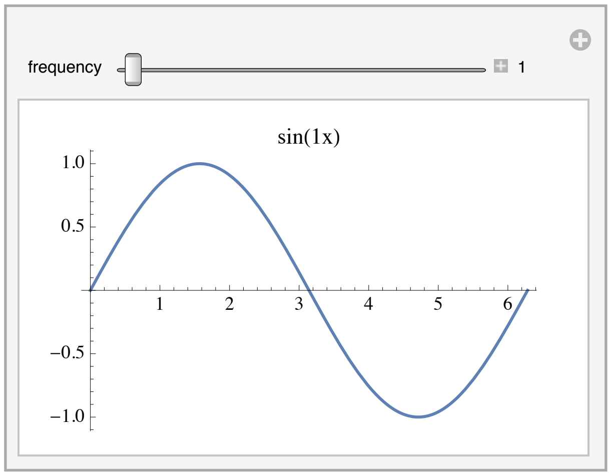
Strings can also be joined together with the **<>** operator. This is useful when constructing a single string from multiple pieces of information that might be coming from different places.

```
Plot[Sin[x], {x, 0, 2 π}, PlotLabel → "My plot of " <> "sin(x)"]
```



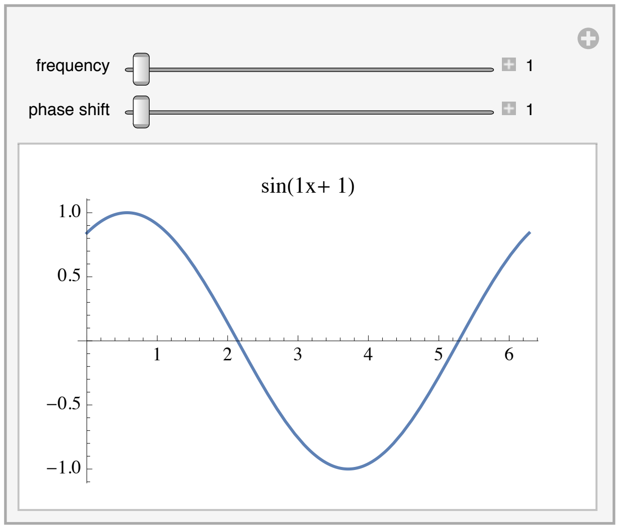
To create an interactive plot label, the **PlotLabel** option has to be hooked up to the same parameters as the **Manipulate** command. By using the same parameter symbol name, when the plot is manipulated, its plot label will simultaneously update. However, the **PlotLabel** option expects a string, and parameters in **Manipulate** commands are generally not strings. A trick is to use the **ToString** command to convert an expression to a string, and then to use **<>** to hook multiple strings together.

```
Manipulate[
  Plot[Sin[f*x], {x, 0, 2 π}, PlotLabel → "sin(" <> ToString[f] <> "x)"],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"}]
```



The same approach can be used to create an interactive plot label that updates based on the values of several manipulable parameters.

```
Manipulate[
  Plot[Sin[f*x + ps], {x, 0, 2 π},
    PlotLabel → "sin(" <> ToString[f] <> "x+ " <> ToString[ps] <> ")"],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"},
  {{ps, 1, "phase shift"}, 1, 6, Appearance → "Labeled"}]
```

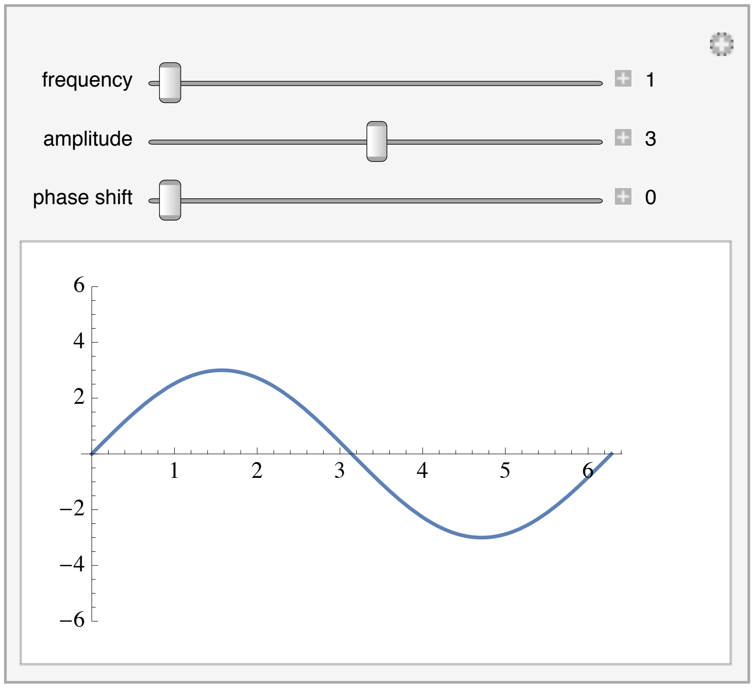


The **<>** symbol is actually shorthand for a command named **StringJoin**, but since it is used so often, the symbolic shorthand form exists. There are other commands like this in *Mathematica* with symbolic shorthand forms, so if you see a symbol you do not recognize, you can search the documentation to find the corresponding formal command name.

Hiding Code

Manipulate commands especially lend themselves to hidden code because the input that created the model is usually not as important as the model itself. Like other situations where hidden input is desirable, simply double-click the cell bracket containing the output (the interactive model created by **Manipulate**) to hide the corresponding input.

```
Manipulate[
  Plot[a*Sin[f*x + ps], {x, 0, 2 π}, PlotRange → 6],
  {{f, 1, "frequency"}, 1, 5, Appearance → "Labeled"},
  {{a, 3, "amplitude"}, 1, 5, Appearance → "Labeled"},
  {{ps, 0, "phase shift"}, 0, 2 π, Appearance → "Labeled"}]
```



Obfuscating code can be taken one step further by deleting the input entirely or by copying and pasting just the output (the interactive model) into a separate notebook. In many cases, the interactive model will still function when the notebook is opened, although it will not be operational if it references a function or data that is no longer available at the time of future reuse. The next section outlines ways to make **Manipulate** statements self-contained so that they include all necessary definitions.

Double-clicking the output to hide the input is much more common than deleting it. If you keep the input intact, you can add minor edits later quite easily. If the input is deleted, you would likely have to start over and recreate the **Manipulate** statement.

Remembering User-Defined Functions

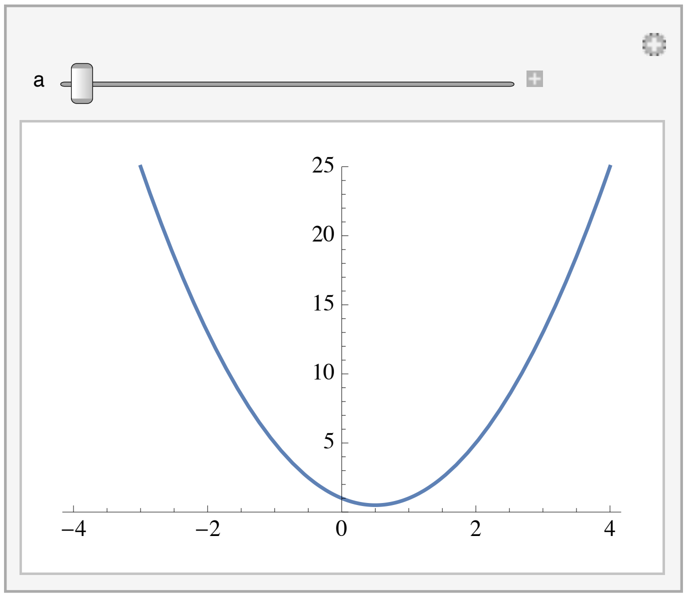
While the examples so far have utilized Wolfram Language functions, the **Manipulate** command can be used with any expression, including user-defined functions. Once a function is defined, then **Manipulate** can operate on it. As an example, the function **f[x]** is defined as follows.

```
f[x_] := 2 x^2 + 2 x + 1
```

Remember, you can typeset an exponent using the Ctrl+6 keyboard shortcut or by using one of the palettes to create a typesetting template.

And now this function can be used with **Manipulate**.

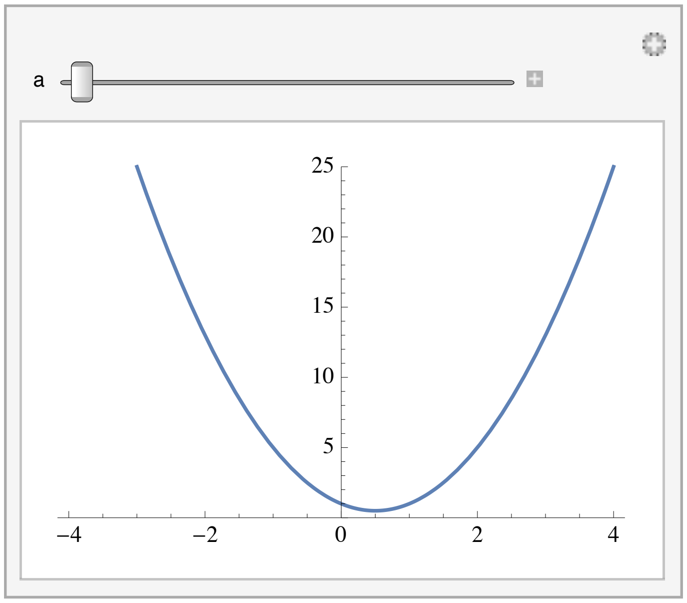
```
Manipulate[
  Plot[f[a*x], {x, -4, 4}, PlotRange → {0, 25}],
  {a, -1, 1}]
```



If the output cell of the above expression—the interactive model created by **Manipulate**—was copied to a new notebook, and the *Mathematica* session was ended, and the new notebook was reopened later, then the interactive model would no longer function because *Mathematica* would not remember the definition of the function **f**.

There are two different strategies that can be employed to use **Manipulate** with user-defined functions. The first is to use **Initialization**, which allows the definition of symbols to be performed when the **Manipulate** command is evaluated. The syntax for **Initialization** uses **RuleDelayed**, which is more commonly input using the escape sequence Esc :> Esc and automatically converted to **:→** as its shorthand form. In the following example, the **Initialization** option setting is placed on its own line for the sake of clarity.

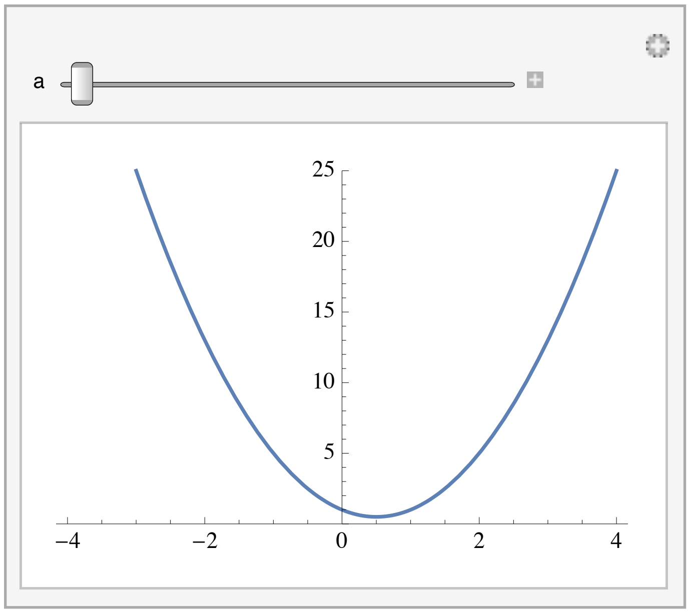
```
Manipulate[
  Plot[f[a*x], {x, -4, 4}, PlotRange → {0, 25}],
  {a, -1, 1},
  Initialization :→ (f[x_] := 2 x^2 + 2 x + 1)]
```



Now if the output (or the input and output) is copied into a new document, saved, and reopened at a later time, the **Manipulate** model will work.

A second approach is to use the **SaveDefinitions** option, which will save the current definitions for every symbol in the **Manipulate** command; these saved definitions will travel with the interactive model, even when copied and pasted to a new notebook.

```
Manipulate[
  Plot[f[a*x], {x, -4, 4}, PlotRange → {0, 25}],
  {a, -1, 1},
  SaveDefinitions → True]
```



As before, if the output is now copied into a new document, saved, and reopened at a later time, the **Manipulate** model will work; it "remembers" the definition for the function **f** since it was told to save the definitions.

In general, using **Initialization** is good if you might want the recipient of your document to see the underlying commands used to construct your interactive model. If you would prefer to hide that information from your audience, then **SaveDefinitions** can be a better approach.

Clear is used to remove all variable and function definitions from this chapter.

```
Clear[f]
```

Conclusion

The use of **Manipulate** to communicate ideas is quite popular with *Mathematica* users, since the results can be immediately understood without the audience having to understand or even see any Wolfram Language commands. A good understanding and generous use of **Manipulate** can go a long way in explaining ideas, illustrating concepts, and simulating phenomena—and all with a single command!

Exercises

1. Create a **Manipulate** statement to vary $x^2 + 1$, where x is an integer ranging from 1 to 10.
2. Similarly to Exercise 1, create a **Manipulate** statement to produce a list of values of the form $\{x,\ x^2 + 1,\ x^3 + 1\}$, where x is an integer ranging from 1 to 10.
3. Create a **Manipulate** statement to show the list of $\{x,\ x^2 + 1,\ x^3 + 1\}$ and then add a fourth element to the list that is an expression that answers whether $x^2 > 2\,x + 1$. As before, use the same integer range of 1 to 10 for the variable x.
4. Use the Wolfram Language to create a plot of $x^2 + 3\,x - 1$ over the domain from -5 to 5.
5. Use **Manipulate** to visualize the behavior of $x^2 + 3\,x - 1$ when a constant c is used to multiply x^2, and where c ranges from 1 to 20.
6. When moving the slider from the example in Exercise 5, remember that *Mathematica* is choosing the optimal plot range as the slider is moved. Use **PlotRange** to introduce a fixed plot range of -5 to 100.
7. Copy the input from Exercise 6 and add a second constant d to change $3\,x$ to $3\,d\,x$, where d also ranges from 0 to 20.
8. Copy the input from Exercise 7 and add another function so that you are visualizing both $c\,x^2 + 3\,d\,x - 1$ and $2\,c\,x^2 - d\,x + 3$. (Reminder: to visualize two functions on the same set of axes, place the functions in a list.)
9. Use the **ThermometerGauge** command to create a gauge illustrating the temperature of 10 on a scale of 0 to 50.
10. Now use **Manipulate** to create a model of the temperature $10\,x$, where x can be changed from 0 to 5.